Subcoinductive Types

Daniëlle Gramsbergen

Supervisors:
Henning Basold and Peter Bruin

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Mathematical Institute (MI)

30/06/2020

Abstract

In the field of type theory, the current technique of constructing subtypes has been proven to be restrictive when it comes to recursive structures, such as most coinductive types (coTs). In this thesis, a new technique of assembling subtypes, of coinductive types especially, is introduced. This technique allows these subcoinductive types (ScoTs) to be built from the ground up without needing to specify a coT as supertype in advance and can be related to a coT later by creating an embedding.

Contents

1 Introduction

Type theory is a an area of computer science and logic that was conceived in order to circumvent some of the paradoxes known to set theory. Today, type systems are used in several ways. Programming languages, for example, use type systems to detect type mismatch at compile time, usually caused by small mistakes made by the programmer [Geu]. The main way in which type theory differs from set theory, is that the terms cannot be introduced without specifying their type. The resulting hierarchy of these types prevents circular propositions and avoids, for instance, Russell's Paradox. However, type theory comes with some downsides as well. Working with constructive type theory, some axioms (law of the excluded middle) and proof techniques (proof by contradiction) are not viable anymore. Furthermore, the elaborate syntax can make understanding underlying concepts difficult at times, making constructive type theory challenging to work with. Nonetheless, axioms can be added when needed and the existence of proof assistants helping to catch mistakes, remedy these problem partially.

Type theory has close links to type checking systems used in programming languages, which enforce well behaved programs. Most data structures can be easily captured by algebraic techniques, but for some structures this has proven to be strenuous. Especially dynamical structures undergoing incessant change turned out impossible to express in an algebraic structure. However, systems that are based on some kind of state, can be described by coalgebraic structures such as coinductive types. Coinductive types (coTs) are defined by the observations that can be made on them. Such observations can be compared as extracting some information from a (dynamical) system along with the system moving to the next state.

Due to the recursive nature of most coinductive types, constructing subtypes in the manner currently known to type theory exhibits limitations. These current subtypes are inhabited by dependent pairs. These pairs consist of a term of a certain type and a proof of this term holding to a certain predicate. To avoid cardinality problems, the said predicates have to conform to limiting rules. Furthermore, this construction forces recursive predicates to be written in an iterative way, which may be impossible for some forms of recursion.

The aim of this thesis is to answer the question: what is a suitable syntax for subtypes of coinductive types? Definitions and techniques will be introduced describing a new way of constructing subtypes, specifically for coinductive types. Tools from both category theory and type theory will be used to accomplish this task, where categorical concepts will be translated into type theoretical concepts. In contrast to the current technique, these subcoinductive types (ScoTs) will not be built out of an existing coT, but will be built from the ground up. After constructing a ScoT, it can be related to a coT by constructing an embedding from the ScoT into the coT being the ScoT's supertype. This thesis is roughly split up into two parts. The first part consists of preliminaries (Chapter 2) and the second part outlines new theory (Chapter 3) and a comparison of this new theory to the existing theory (Chapter 4). The preliminaries act as a reference

to some concepts of type theory, category theory and coinduction, which are used in the rest of the thesis. Readers well versed in category theory and type theory can skip Chapter 2 and refer back to it when needed.

2 Homotopy Type Theory, Category Theory and Coinduction

2.1 Homotopy Type Theory

Homotopy type theory [1] can be seen as an alternative to set theory in the formalisation of mathematics. The main addition of homotopy type theory to general (Martin-Löf) type theory is that types can viewed as spaces in homotopy theory. Whereas (homotopy) type theory itself is made up of rules instead of axioms, univalent type theory is homotopy type theory with one added axiom: univalence. This axiom states that identity is equivalent to equivalence [Esc18], roughly speaking. In this thesis, however, univalance will not be used, so the theory we are working in is regular homotopy type theory.

(Homotopy) type theory is concerned with *types* inhabited by *terms*, which from a homotopical perspective can be regarded as *points* in *spaces*. When introducting a term x of type X, this is written as $x : X$, as one would write $x \in X$ in set theory. Terms cannot be introduced without specifying their type, which is one aspect distinguishing type theory from set theory. Furthermore, in the type theory used in this thesis, terms belong to exactly one type, provided that all free variables are equipped with a type. In set theory, elements are allowed to be contained in many different sets, which is not the case for terms and types in the type theory used here.
Another difference between type theory and set theory is that type theory is *necessarily* constructive, which can be implemented by requiring that a proof of a theorem is given as a terminating program. Types can be regarded as theorems or propositions, with finding a term (or 'witness') of this type being the equivalent of finding a proof for the theorem the type represents. This interpretation is called the Curry-Howard correspondence. Since types are capable of having multiple terms, these proofs contain more information about the theorem than just the statement of it being 'true' or 'false'.

2.1.1 Introducing New Types

The introduction of a new type in type theory in general follows a certain pattern [Pro13]. A new type is defined by:

- *Formation rules*: how to form new types of this kind.
- *Introduction rules*: how to construct terms.
- *Elimination rules*: how to use terms.
- *Computation rules*: how the elimination rules act on the introduction rules.

For the types being discussed in this chapter, some of these rules are remarked upon. However, most of them are not discussed. The interested reader is referred to [Pro13],

[1] This whole section on homotopy type theory is mainly based on and closely follows Chapter 1 of [Pro13]. For sections 2.1.5 and 2.1.9, Chapters 2 and 3 have also bee consulted respectively.

2.1.2 Definitional and Propositional Equality

Two notions of equality exists in homotopy type theory. The first is *definitional* equality ($\equiv$). Definitional equality cannot be proven or negated; terms are definitionally equal simply if they are defined to be equal. Terms (or types) which are definitionally equal may be substituted for each other.

It is however desirable to be able to prove that terms are equal even if they are not defined as equal, hence the second equality: *propositional equality* ($=$). Whereas definitional equality of $x, y : X$, $x \equiv y$ is on the same level in the hierarchy of higher inductive types as the judgement $z : X$, propositional equality operates on a higher level. Propositional equality is provable, which in homotopy type theory means finding a term for the proposition $x = y$, making $x = y$ a new type. From a homotopical perspective, any witness $p : (x = y)$ can be regarded as a path between points x and y in space X. There may be multiple paths between these points, which is to say there can be multiple proofs of x and y being equal. Additionally, it is possible to construct paths between paths, and again paths between those paths and so forth, giving rise to higher inductive types. One equality type is used extensively and has therefore been given a name: reflexivity. For a term $x : X$, reflexivity of x is of the following type:

$$\mathrm{refl}_X\ x : x =_X x$$

These equalities can interact in the following way. Say there is a proof $p : (x = y)$ and by definition $y \equiv z$, then p is also a proof for $x = z$, insomuch that y can be substituted for z due to the definitional equality.

2.1.3 Universes

Types with terms, which are not types themselves, are called 'small types'. However, there also exist types of which the terms are again types. These types are called universes $\mathcal{U}$.

There exist different ways to relate universes to each other. One would be a cumulative approach ('Russel style'):

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots.$$

The main downside is losing uniqueness of type for each term. A possible solution would be 'Tarski style' universes, where terms of smaller universes are mapped to bigger universes in a unique way. Since these problems do not affect us directly in this thesis,

we will not go into the technical details. Furthermore, because the place in the hierarchy of the used universe is not relevant in the theory discussed, the notation $X : Ty$ has been chosen to denote term X is a type itself. Ty does, in fact, mean the same thing as $\mathcal{U}$. The only thing that is relevant in this thesis is that the uniqueness of type for terms is kept. The interested reader is referred to [Pro13], Notes of Chapter 1.

2.1.4 Function Types, Type Families and $\Pi-$Types

In set theory, functions are defined as functional relations. However, in type theory, they are a primitive concept. Formation of a function type is a follows. Given $X, Y : Ty$, a new type $X \to Y$ of functions can be formed. Functions are introduced using $\lambda-$abstraction. Say we want to define a function $f : X \to Y$ which uses a variable $x : X$. This can be written as:

$$f :\equiv \lambda(x : X).\Phi$$

where Φ is expression which may contain $x : X$.
Functions are eliminated by function application and computation is defined for $x' : X$ as:

$$f(x') :\equiv \Phi'$$

where Φ' is the expression for which all instances of x are substituted by instances of x'.

Maps do not restrict to small types, one can also make to, or from, universes. A map mapping terms inhabiting a small type to types inhabiting a universe is called a **type family**. Take for instance type family Y:

$$Y : X \to Ty.$$

For function $f : X \to Y$, the codomain Y is constant. It is nonetheless possible, with some type family, to define a function where the type of the codomain varies. Let $Y : X \to Ty$ again be a type family. Define a type of dependent functions as:

$$\prod_{x:X} Y(x)$$

This type family Y can contain free variables in the same way a non-dependent function can. If Y is a constant type family, then a function $f : \prod_{x:X} Y$ reverts back to a 'regular' function with a constant domain.

Note that the type theoretic operation of taking a product type over all terms of X ($\prod_{x:X}$) can be regarded as the type theoretical version of the logical operation of 'for all' terms of X ($\forall x \in X$). This can be recognised as follows: dependent function

$\prod_{x:X} Y(x)$ gives a term of $Y(x) : Ty$ for every term x of X. Due to the 'propositions as types' view, this term can be seen as a proof of the truth of $Y(x)$. Since for every term x of X such a proof of $Y(x)$ exists, one can infer Y being true for all terms x of X.

2.1.5 Homotopies

Definition 2.1. Let $Y : X \to Ty$ be a type family and $f, g : \prod_{x:X} Y(x)$ be dependent functions over this family. **Homotopy** $f \sim g$ is a dependent function of type:

$$f \sim g :\equiv \prod_{x:X} (f(x) = g(x)).$$

In set theory, (after translating the $\Pi_{x:X}$ into $\forall x \in X$) this would be enough to conclude $f = g$. However, in (homotopy) type theory, this is not necessarily true. Two functions f and g could be defined differently, but give the same output, which in (Martin Löf) type theory does not make them equal. In other words: a path between all the outputs of the functions does not imply there is a path between the functions. To make these functions equal in type theory, an extra axiom has to be introduced called **function extensionality**:

$$\text{funext} : \Big(\prod_{x:X} (f(x) = g(x)) \Big) \to (f = g).$$

Function extensionality follows from univalence. In this thesis, the univalence axiom will not be adopted, as it did not appear to be required in the theory developed in Chapter 3. Consequently, function extentionality will not hold. The homotopical interpretation, however, will be utilised in Section 3.5.

2.1.6 Product Types and $\Sigma-$Types

The product type $X \times Y$ is formed from types $X, Y : Ty$. Terms are introduced by taking $x : X$ and $y : Y$, making a pair $\langle x, y \rangle : X \times Y$. In order to access one of the terms in a pair, projections

$$\pi_L : X \times Y \to X$$
$$\pi_R : X \times Y \to Y$$

are the elimination rules of this type, where π_L returns the left term and π_R returns the right term. Tuples containing more than two terms also have projections, usually denoted as $\pi_0, \pi_1, \pi_2, \ldots \pi_{n-1}$ where each projection returns the $(n-1)^{\text{th}}$ term. In this thesis, counting starts from 0, but since this is not the case across all the literature, L and R are used for pairs to avoid confusion.

Similar to dependent functions, it is possible to define a type of dependent pairs,

or $\Sigma-$types. For $\Sigma-$types the type of the second term of the pair depends on the first term of the pair. In other words, for a type family $Y : X \to Ty$, a type of dependent pairs is written as:

$$\sum_{x:X} Y(x),$$

where pairs $\langle x, y\rangle : \sum_{x;X} Y(x)$ are constructed out of terms $x : X$ and $y : Y(x)$.
Just as with product types, elimination is given by projections, one of which is a dependent projection.

$$\pi_L : \Big(\sum_{x:X} Y(x)\Big) \to X$$

$$\pi_R : \prod_{w:\sum_{x:X} Y(x)} Y(\pi_L(w))$$

Again, for Y a constant family: $X \times Y = \sum_{x:X} Y$

Like the operation for depenedent functions, the type theoretic operation for taking a dependent pair for some term x of X ($\sum_{x:X}$) also has a corresponding logical operation. In this case, the operation is the 'exists' for x of X operation ($\exists x \in X$). This can be seen in the following way. A proof of $\sum_{x:X} Y(x)$ needs only one pair $\langle x,\ y\rangle$, with y a term of $Y(x)$, which means there has to be at least one term x of X for which there exists a proof for $Y(x)$. In other words, there *exists* a term x of X such that $Y(x)$ is true.

2.1.7 Currying

Something that has not been discussed yet is functions taking multiple variables, or functions with a product type as its domain. Such a function $f : X \times Y \to Z$ can be defined as:

$$f(\langle x, y\rangle) :\equiv g\ (x)\ (y)$$

where $g : X \to Y \to Z$ (right associative) is function taking a variable of type X and returning another function only dependent a variable of type Y. This can be generalised to n-tuples, where the process of each step taking one variable and returning a function dependent on $n-1$ variables is called currying. Due to the lack of ambiguity, parentheses are often omitted, meaning g can be written in the following way:

$$g\ (x)\ (y) :\equiv g\ x\ y.$$

This notation is left associative, meaning $g\ x$ is applied first, returning another function taking only y as variable.

2.1.8 Coproduct Types

With types $X, Y : Ty$, a new type can be formed: the coproduct type $X + Y$. With $x : X, y : Y$, term $[x, y] : X + Y$ can be introduced. This coproduct type can be related to the disjoint union in set theory. There are two maps into the coproduct type

$$\kappa_L : X \to X + Y$$
$$\kappa_R : Y \to X + Y.$$

These maps can be regarded as 'inclusion maps'. The coproduct type can be seen as a sort of dual to the product type, since it is defined by maps *into* the coproduct, whereas product are defined by maps going *out of* the product (projections).

Coproduct types also come with a useful elimination principle, which is of type:

$$\mathrm{el}_+ : (X \to Z) \to (Y \to Z) \to (X + Y \to Z).$$

Elimination takes two maps as arguments and returns a map which domain is the coproduct of the domains of its argument. Now both introduction and elimination are defined, it is possible to specify computation. Map em_+ returns the image of the first argument if X is inhabited and the image of the second argument if Y is inhabited. Let $f : X \to Z$, $g : Y \to Z$ be existing maps. Elimination on these maps is defined thus defined as follows.

$$\mathrm{el}_+(\kappa_L\ x) :\equiv f(x)$$
$$\mathrm{el}_+(\kappa_R\ y) :\equiv g(y)$$

2.1.9 Contractibility

There are some special types for which all terms are considered to be the same. These types are said to be contractible.

Definition 2.2. Type X is **contractible** if there exists a term $x : X$, such that for all $y : X$, $x = y$ holds.

If one regards type X as a proposition, contractibility would mean all proofs of this proposition can be equated. Types which act as sets can be defined using this contractibility.

Definition 2.3. Type X is a **set** if for all $x, y : X$ and all paths $p, q : x = y$, $p = q$ holds.

Even though sets are easier to work with, in this thesis we will stick to regular types which are not necessarily sets. The notion of contractibility, however, will be used when examining an alternative definition of subtype in Chapter 4.

2.1.10 Types $\mathbb{0}$, $\mathbb{1}$ and $\mathbb{2}$

The three types $\mathbb{0}, \mathbb{1}$ and $\mathbb{2}$ are all important to type theory and will be used extensively throughout this thesis. Therefore, a short description will be given of each.

The *empty type* $\mathbb{0}$ is the type without inhabitants. It is the neutral element of the coproduct.

The *unit type* $\mathbb{1}$ is the type with one term, namely $\star : \mathbb{1}$. $\star$ can be regarded as the empty product. Every contractible type is logically equivalent to $\mathbb{1}$ [Pro13].

The *type of booleans* $\mathbb{2}$ has to terms: $0_{\mathbb{2}}$ and $1_{\mathbb{2}}$ (usually simply written as 0 and 1 if the type is clear from context). This type can be regarded as the coproduct of unit types: $\mathbb{1} + \mathbb{1}$.

2.2 Category Theory

In this thesis, we will mostly be working with type theory, however, there is a close connection between type theory and category theory [LS88] [2]. Some concepts are more comprehensible if looked at through a categorical lens, which makes it useful to very briefly introduce the basic concepts in order to provide a better explanation for some concepts later on.

2.2.1 Categories and Functors

Category theory and type theory are closely linked. For most of this thesis, we will make use of categorical interpretations due to their clarity. In this section, basic categorical definitions are given.

Definition 2.4. A **category** $\mathscr{C}$ is a collection of objects $\mathrm{ob}(\mathscr{C})$ together with for each $X, Y \in \mathrm{ob}(\mathscr{C})$ a collection of arrows/morphisms $\mathscr{C}(X, Y)$ between X and Y. These objects and morphims obey the following properties:

- For each X, Y and Z, a *composition function*:

$$\circ : \mathscr{C}(X, Y) \times \mathscr{C}(Y, Z) \to \mathscr{C}(X, Z)$$

$$(f, g) \mapsto g \circ f$$

- For each $X \in \mathrm{ob}(\mathscr{C})$, there is the *identity* on X:

$$1_X \in \mathscr{C}(X, X).$$

  Sometimes the identity written as id_X or simply id if the object is clear from the context.

- For each $f \in \mathscr{C}(X, Y)$, we have *identity laws*:

$$f \circ 1_X = f = 1_Y \circ f$$

- For each $f \in \mathscr{C}(X, Y),\ g \in \mathscr{C}(Y, Z)$ and $h \in \mathscr{C}(Z, W)$, we have *associativity*:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

To clarify this definition, some examples are provided using familiar structures.

[2] Definitions from Sections 2.2.1 and 2.2.3 are based on [Lei14] and definitions in Section 2.2.2 is based on [Jac99], Chapter 1.

Example 2.1. *$\mathscr{E}$ is the category of sets. Class* $\mathrm{ob}(\mathscr{E})$ *contains all sets and the class of morphisms consists of functions between those sets. Formally, this is to say for all* $X, X' \in \mathrm{ob}(\mathscr{E})$*, $\mathscr{E}(X, X')$ is the class of all functions with domain X and codomain X'. It is easy to check there is an identity for all $X \in \mathrm{ob}(\mathscr{E})$ following the identity laws. Moreover, function composition and associativity are also easily shown. This is omitted since it is not very interesting to write out in full here.*

Example 2.2. *$\mathscr{G}$ is a category of groups. Class* $\mathrm{ob}(\mathscr{G})$ *contains all groups and the class of morphisms consists of group homomorphisms between those groups. Formally, this is to say for all* $X, X' \in \mathrm{ob}(\mathscr{G})$*, $\mathscr{G}(X, X')$ is the class of all group homomorphisms from X to X'.*

Example 2.3. *$\mathscr{O}$ is a category of topological spaces. Class* $\mathrm{ob}(\mathscr{O})$ *contains all topological spaces and the class of morphisms consists of continuous maps between those topological spaces. Formally, this is to say for all* $X, X' \in \mathrm{ob}(\mathscr{O})$*, $\mathscr{G}(X, X')$ is the class of all continuous functions from X to X'.*

Example 2.4. *$\mathscr{T}$ is a category of types. Class* $\mathrm{ob}(\mathscr{T})$ *contains all types and the class of morphisms consists of maps between those types. Formally, this is to say for all* $X, X' \in \mathrm{ob}(\mathscr{T})$*, $\mathscr{G}(X, X')$ is the class of all functions with domain X and codomain X'. Note due to the lack of function extensionality, some technical difficulties arise when trying to precisely define this category. Because these technicalities do not matter for the scope of this thesis, readers interested in the details are referred to [Jac99], chapter 2.*

There is also the possibility of constructing maps *between* categories, by mapping both the objects and the arrows.

Definition 2.5. Let $\mathscr{C}$ and $\mathscr{D}$ be categories. A **functor** $F : \mathscr{C} \to \mathscr{D}$ is a map between categories with:

- A *map between objects*:
$$F : \mathrm{ob}(\mathscr{C}) \to \mathrm{ob}(\mathscr{D})$$
$$X \mapsto F(X)$$
- For each $X, Y \in \mathrm{ob}(\mathscr{C})$ a *map between arrows*:
$$F : \mathscr{C}(X, Y) \to \mathscr{D}(F(X), F(Y))$$
$$f \mapsto F(f),$$

satisfying the following identities:

- For $X \in \mathrm{ob}(\mathscr{C})$:
$$F(1_X) = 1_{F(X)}$$

- For $f \in \mathscr{C}(X, Y),\ g \in \mathscr{C}(Y, Z)$:

$$F(g \circ f) = F(g) \circ F(f).$$

Example 2.5. *For categories $\mathscr{G}$ and $\mathscr{E}$, the map $F : \mathscr{G} \to \mathscr{E}$ mapping a group to its underlying set is a functor. It is easily checked that identity and associativity still hold after application of F.*

Definition 2.6. A **small category** is a category for which its objects and morphisms are contained in proper sets instead of in classes.

2.2.2 Fibrations and Substitution Functors

One can think of fibrations in an informal manner as a way to index one category over another. Fibrations give rise to substitution functors, a concept that will be useful later in this thesis.

Definition 2.7. A morphism $f : X \to Y$ is **Cartesian** over, or a **Cartesian lifting** of, $u : I \to J$ in $\mathscr{C}$ if both $p(f) = u$ and every $g : Z \to Y$ in $\mathscr{D}$ for which $p(g) = u \circ w$ holds for some $w : p(Z) \to I$, uniquely determines a $h : Z \to X$ in $\mathscr{D}$ above w with $f \circ h = g$.

Definition 2.8. A functor $p : \mathscr{D} \to \mathscr{C}$ is a **fibration** if for every $Y \in \mathrm{ob}(\mathscr{D})$ and $u : I \to p(Y)$ in $\mathscr{C}(I, p(Y))$, there is a Cartesian morphism $f : X \to Y$ in $\mathscr{D}(X, Y)$ above u.

Due to the technicality of these definitions, it might be useful to examine them in terms of the diagram in Figure 1 below.

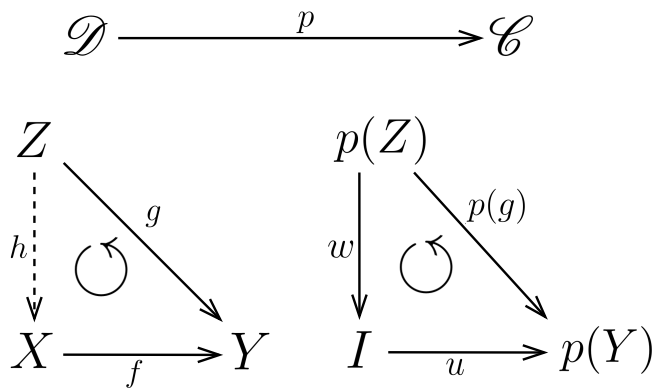

Figure 1: The diagrams belonging to p being a fibration.

Assume there exists some fibration $p : \mathscr{D} \to \mathscr{C}$. This means one can now say something useful about the *fibres* of $\mathscr{D}$.

Definition 2.9. Fibre $\mathscr{D}_I = p^{-1}(I)$ over I (for category $\mathscr{D}$), is a category with $X \in \mathrm{ob}(\mathscr{D}_I)$ if $X \in \mathrm{ob}(\mathscr{D})$ and $p(X) = I$. Furthermore, $f \in \mathscr{D}_I(X, Y)$ if $f \in \mathscr{D}(X, Y)$ and $p(f) = 1_I$ in $\mathscr{C}$.

The fibration p comes with a Cartesian lifting $f : X \to Y$ over $u : I \to p(Z)$ (in $\mathscr{C}(I, p(Y))$) for all $X, Y \in \mathrm{ob}(\mathscr{D})$. Using the Axiom of Choice, one can now choose a specific lifting for each $u : I \to p(Z)$, which will be noted as:

$$\overline{u}(f) : u^*(X) \to X$$

With such a specific lifting for each $X \in \mathrm{ob}(\mathscr{D})$, it it is possible to determine the following functor between fibres of $\mathscr{D}$ as shown below.

$$u^* : \mathscr{D}_J \to \mathscr{D}_I$$

Definition 2.10. Let $u : I \to J$ be a morphism in $\mathscr{C}$. Furthermore, let $\overline{u}(X) : u^*(X) \to X$ be a Cartesian lifting with for fibres $\mathscr{D}_J$ and $\mathscr{D}_I$ of $\mathscr{D}$: $X \in \mathrm{ob}(\mathscr{D}_j)$ and $u^*(X) \in \mathrm{ob}(\mathscr{D}_I)$. Let $f : X \to Y$ be another morphism in $\mathscr{C}$. Functor $u^* : \mathscr{D}_J \to \mathscr{D}_I$ is called a **substitution functor**, which makes the diagram in $\mathscr{D}$ commute.

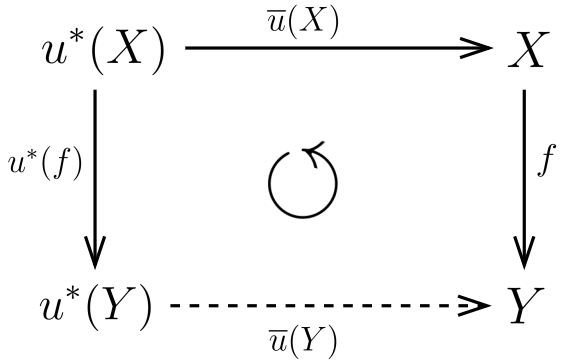

Figure 2: Diagram over $\mathscr{D}$ with substitution functor u^*.

Example 2.6. *Let's look at a possible definition of the category of predicates $\mathscr{P}$ used in [Jac99]. Class* $\mathrm{ob}(\mathscr{P})$ *contains pairs $\langle I, X\rangle$, where X is a type family indexed over I and can thus be seen as a predicate. $i : I$ is a free variable in X. For $\langle I, X\rangle, \langle J, Y\rangle \in \mathrm{ob}(\mathscr{P})$, a morphism is a apair of maps $\langle u, v\rangle$ with types $u : I \to J$ and $v : \prod_{i:I} X(i) \to Y(u(i))$..*

The functor $p : \mathscr{P} \to \mathscr{E}$ defined as $\langle I, X \rangle \mapsto I$, sending a predicate to the underlying type is an example of a fibration. This fibration gives for each $u : I \to J$ rise to substitution functor $u^* : \mathscr{P}_J \to \mathscr{P}_I$.

In this thesis, however, type theory is used, which means this useful substitution functor cannot be used directly. In type theory, predicates are defined as being of type:

$$Pred : \sum_{X:Ty} X \to Ty.$$

Due to the lack of function extensionality and the lack of functor axioms, the map

$$p : Pred \to Ty$$

cannot be proven to be a fibration. The concept of the substitution functor, however, will be used. In the case of type theory, this substitution map (as it is not a functor) will be defined explicitly where needed.

Example 2.7. *Let*

$$f : X \to Y$$

be a map between types $X, Y : Ty$. One can explicitly define the substitution map:

$$f^* : Pred_Y \to Pred_X$$

$$f^*\ P :\equiv \lambda x.P\ (f\ x).$$

This substitution map, inspired by the concept of the substitution functor in category theory, will prove to be convenient in defining a coiteration principle in Section 3.2 and defining bisimulation in Section 3.4.

2.2.3 Limits

In mathematics, the term limit is often associated with some kind of 'finality' of an object, like is the case for limits in analysis. For categories, a limit is something with just enough structure to say something relevant about a diagram, without unnecessary structure getting in the way.
To make this precise, some definitions are needed. Consider a 'diagram' for some category, which is to say a graphical representation of objects and morphisms between them. Take for example a category $\mathscr{D}$ with $D_1, D_2, D_3 \in \mathrm{ob}(\mathscr{D})$, $d_{12} \in \mathscr{D}(D_1, D_3)$ and $d_{13} \in \mathscr{D}(D_2, D_3)$ (the identities are left out for the sake of clarity). The graphical representation of this category is shown below in Figure 3.

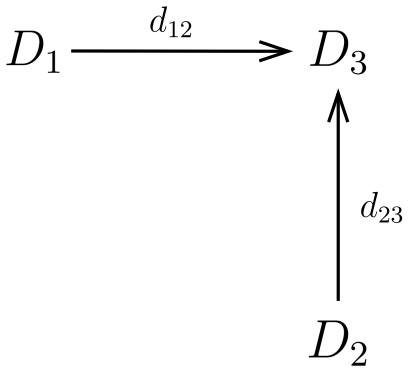

Figure 3: A graphical representation of category $\mathscr{D}$.

One can imagine the general shape of this graphical respresentation, without specifying specific objects and morphisms to be:

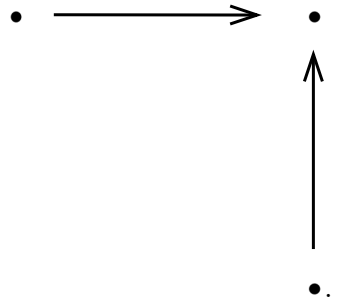

Figure 4: The general shape of the category $\mathscr{D}$.

This is a useful way to think about the concept informally, but for it to be useful a small category needs to be chosen to take the place of this generalised shape. The idea is that this small category has objects which can be placed on the dots of Figure 4 in a way that the morphisms still hold. With this idea of a generalised shape, diagrams, cones and limits can be defined formally.

Definition 2.11. Let $\mathscr{C}$ be a category and $\mathscr{I}$ be a a small category. A functor

$$D : \mathscr{I} \to \mathscr{C}$$

is called a **diagram** over **shape** $\mathscr{I}$.

Definition 2.12. Let $\mathscr{C}$ be a category and $D : \mathscr{I} \to \mathscr{C}$ be a diagram. An object $C \in \mathrm{ob}(\mathscr{C})$ together with a family

$$(f_I : C \to D(I))_{I \in \mathscr{I}}$$

is called a **cone** if for all maps $u : I \to J$ the diagram below commutes.

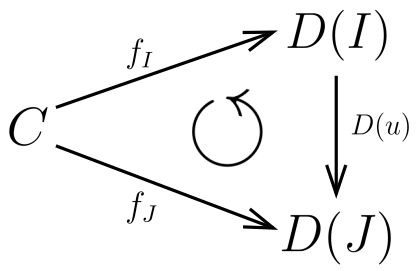

Figure 5: A cone of object C and family $(f_I : C \to D(I))_{I \in \mathscr{I}}$.

Definition 2.13. A cone

$$(p_I : L \to D(I))_{I \in \mathscr{I}}$$

is a **limit** of D if there exists for each cone

$$(f_I : C \to D(I))_{I \in \mathscr{I}}$$

a map

$$f : C \to L$$

such that for all $I \in \mathscr{I}$, this f satisfies :

$$p_I \circ f = f_I.$$

To clarify these definitions, the diagram below will be put into context.

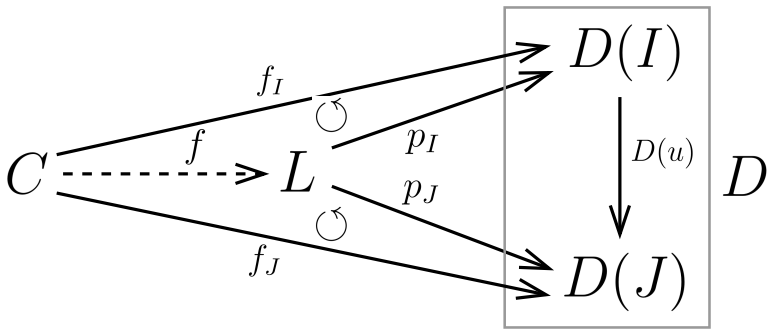

Figure 6: Object L and family $(p_I : L \to D(I))_{I \in \mathscr{I}}$ together are a limit of D.

A cone C has certain information about diagram D. Every such cone factors through a 'universal cone' of L, which can be seen as having all the *necessary* information about diagram D. Since these are very abstract concepts (like most concepts in category theory), two examples are discussed in short for clarification. Those familiar with category theory may already have recognised the limit over the shape of Figure 4 to be a pullback.

Example 2.8. *Let for some category $\mathscr{C}$, $X, Y \in \mathrm{ob}(\mathscr{C})$. A product P, together with its projections $p_L : P \to X$, $p_R : P \to X$ is a limit over the category containing only objects X and Y (without any morphisms between them).*

Example 2.9. *Let $\mathscr{R}$ be the category of rings. The ring of p-adic numbers $\mathbb{Z}_p$ is a limit [Sut13] over the diagram:*

$$\cdots \to \mathbb{Z}/(p^3) \to \mathbb{Z}/(p^2) \to \mathbb{Z}/(p).$$

In Chapter 3 $\mathbb{Z}_p$ will be shown to be in fact a subcoinductive type.

2.3 (Co)Induction and (Co)Inductive Types

In mathematics, induction is a familiar concept, but the dual concept of coinduction is less heard of [3]. Induction in type theory describes in which way terms of a(n inductive) type can be analysed, whereas coinduction describes how terms of a (coinductive) type are introduced. The proof techniques of induction and coinduction make use of the specific structure of inductive and coinductive objects respectively. Inductive types are defined by their constructors, which specify how to build an inductive object. Coinductive types, on the other hand, are defined by their observations, which specify how a coinductive type is broken down. The duality of these concepts mentioned before, is in categorical sense. In this section, inductive and conductive type will therefore be examined from a categorical perspective. Inductive and coinductive objects will be considered in general by regarding initial algebras and final coalgebras over any category.

2.3.1 Inductive Types

The concept of induction is familiar to all mathematicians. The proof method of mathematical induction, especially, is often used in the whole of mathematics. In mathematical induction, the initiality of the natural numbers is used. Here, general inductive types are defined as being initial algebras over an endofunctor in a general category. To make this an inductive type specifically, one only needs to choose the mentioned category to be the category of types $\mathscr{T}$.

Definition 2.14. For a category $\mathscr{C}$, an **algebra** over an endofunctor $F : \mathscr{C} \to \mathscr{C}$, is a pair (X, l), where X is an object of category $\mathscr{C}$ and $l : F(X) \to X$ a morphism.

Definition 2.15. An algebra is **initial** if for any algebra (Y, l'), there is a unique morphism $f : X \to Y$ which makes the following diagram commute:

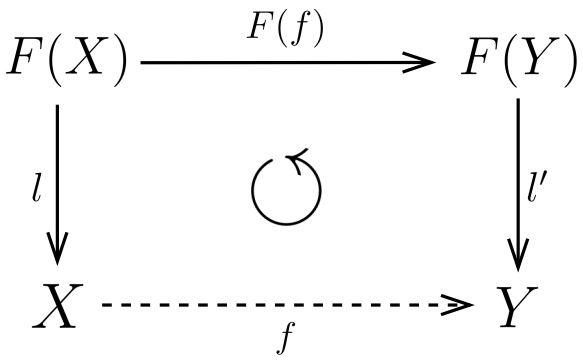

Figure 7: Initial algebra (X, l)

[3] This section is based on [JR97]

To clarify this definition, an example is used which is very familiar: the natural numbers $\mathbb{N}$. Note for the rest of this thesis, it is considered that 0 is s natural number.

Example 2.10. *The natural numbers* $\mathbb{N} \in \mathrm{ob}(\mathscr{E})$ *(including* 0*) are a classic example of an initial algebra.*
Let $\mathrm{null} : \mathbb{1} \to \mathbb{N}$ *and* $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$ *be the constructors of* $\mathbb{N}$*, where morphism* null *can be seen as the base term* 0 *and* succ *is the map which maps a natural number to its successor. Then* $(\mathbb{N}, [\mathrm{null}, \mathrm{succ}])$ *forms an algebra when defining for all* $X \in \mathrm{ob}(\mathscr{E})$ *the endofunctor* $F : \mathscr{E} \to \mathscr{E}$ *in the following way:*

$$F(X) :\equiv \mathbb{1} + X, \quad F(f) :\equiv \mathrm{id} + f.$$

Where in this case id *is the identity morphism on* $\mathbb{1}$*. Let* $(Y, [\mathrm{nl}, \mathrm{s}])$*, with* $\mathrm{nl} : \mathbb{1} \to Y$*,* $\mathrm{s} : Y \to Y$*, be another algebra.*
We define $f : \mathbb{N} \to Y$ *as:*

$$f(\mathrm{null}) :\equiv \mathrm{nl}, \quad f(\mathrm{succ}(n)) :\equiv \mathrm{s}(f(n))$$

The defined f *and* F *make the diagram below commute.*

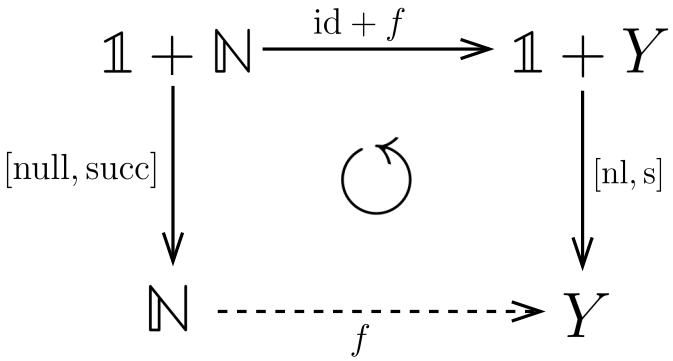

Figure 8: Natural numbers

The uniqueness of f *is fairly easy to check for the category of sets* $\mathscr{E}$ *(assume there is another map* g *with the same properties and show this is in fact the same map as* f *[JR97]). Uniqueness also follows for the category of types* $\mathscr{T}$ *when* $\mathscr{T}$ *is defined carefully, making* $\mathbb{N}$ *an* ***inductive type****. However, as mentioned before, the technicalities of formally defining this category in the right way is beyond the scope of this thesis.*

2.3.2 Coinductive Types

Coinductive types differ from inductive types, as they are defined by *observations* on terms, instead of constructors on terms. The categories of inductive objects and coinductive objects are, in fact, each other's dual. This duality states the objects in the diagram remain, but the sources and the targets of the morphisms switch.

Definition 2.16. For a category $\mathscr{C}$ a **coalgebra** over an endofunctor $F : \mathscr{C} \to \mathscr{C}$, is a pair (X, o) where X is an object of category $\mathscr{C}$ and $o : X \to F(X)$ a morphism.

Definition 2.17. A coalgebra is **final** if for any coalgebra (Y, o') there is a unique morphism $g : Y \to X$ which makes the following diagram commute.

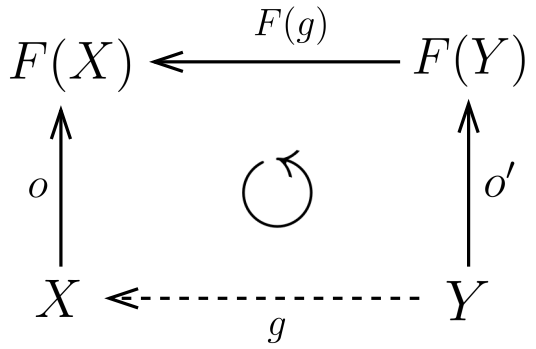

Figure 9: Final coalgebra (X, o)

Comparing the the diagrams in Figures 7 and 9, the main difference between them is that the arrows are pointing in the reverse direction.

Definition 2.18. Every category $\mathscr{C}$ has a *dual* category $\mathcal{C}^{op}$ where $\text{ob}(\mathscr{C}) = \text{ob}(\mathscr{C}^{op})$ and for all $X, Y \in \text{ob}(\mathscr{C})$, we have $\mathscr{C}(X, Y) = \mathscr{C}^{op}(Y, X)$ [Lei14].

This definition makes precise what it means for inductive and coinductive objects to be each others dual. An inductive object in $\mathscr{C}$ is a coinductive object in $\mathscr{C}^{op}$ and vice versa. The standard example for coinductive types is the type of *streams* over some type A. A stream over A can be seen as an infinite list of terms of A. On this stream, written as A^ω, two observations can be made: head : $A^\omega \to A$ giving the first term of the list and tail : $A^\omega \to A^\omega$ giving the rest of the list, excluding the first term (the 'system' in the next 'state'). The example of the type of (ascending) streams of natural numbers, consisting of terms of the form $(n, n+1, n+2, \dots) : \mathbb{N}^\omega$ for any $n : \mathbb{N}$, will be elaborated below.

Example 2.11. *Let $\mathscr{E}$ be the category of sets. The set of streams of natural numbers $\mathbb{N}^\omega \in \mathrm{ob}(\mathscr{E})$ is an example of a coinductive object.*
Let $\mathrm{head} : \mathbb{N}^\omega \to \mathbb{N}$ *and* $\mathrm{tail} : \mathbb{N}^\omega \to \mathbb{N}^\omega$ *be observations of $\mathbb{N}^\omega$, where* head *gives the first term of the stream and* tail *gives the stream without the first term. It is clear* $(\mathbb{N}^\omega, (\mathrm{head}, \mathrm{tail}))$ *is a coalgebra for the endofunctor $F : \mathscr{E} \to \mathscr{E}$ defined as:*

$$F(X) :\equiv \mathbb{N} \times X, \quad F(g) :\equiv \mathrm{id} \times g$$

Where id *is the identity morphism of $\mathbb{N}$. Let $(Y, (\mathrm{h}, \mathrm{t}))$ be another coalgebra. We will look at the specific case where $Y = \mathbb{N}$.*
We define $g : \mathbb{N} \to \mathbb{N}^\omega$ by coiteration, which means g returns an infinite tuple of numbers increasing with one every step, starting at the number n which g takes as an argument.

$$g(n) :\equiv (n, n+1, n+2, \dots)$$

Functions h *and* t *as:*

$$\mathrm{h}(n) := n$$

$$\mathrm{t}(n) := n + 1$$

F and g defined above make the diagram below commute. Uniqueness of g is again easily derived in $\mathscr{E}$ and can also be shown when defining $\mathscr{T}$ carefully. Working in category T would make this final coalgebra a ***coinductive type****.*

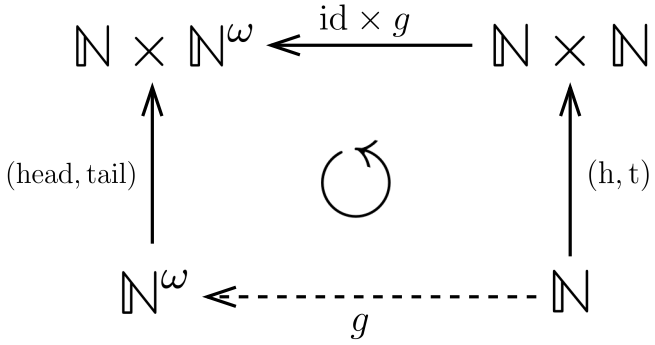

Figure 10: Streams of natural numbers

3 Subcoinductive Types

3.1 Examples

Before defining subcoinductive types (ScoTs) formally, it might be useful to look at some examples which will fit this definition. The properties of these ScoTs are described informally and will be elaborated at several points later on in this chapter. There will be five examples which will be used throughout this chapter to illustrate concepts such as coiteration, computation, bisimilarity and coinduction.

3.1.1 One-after-zero Boolean Streams

Take for example the type of Boolean streams where every zero is followed by a one, written as $2^\omega_{0,1}$. Streams over type A are infinite lists of terms of A. In this case, where $A :\equiv 2$, one gets Boolean streams which are streams of ones and zeros.

$$2^\omega_{0,1} : Ty, \quad \langle \mathrm{h}, \mathrm{t} \rangle : 2^\omega_{0,1} \to 2 \times 2^\omega_{0,1}, \quad \mathrm{o_p} : \prod_{x:2^\omega_{0,1}} \mathrm{h}\ x = 0 \to \mathrm{h}\ (\mathrm{t}\ x) = 1$$

This type is a subtype of the coinductive type (coT) of Boolean streams 2^ω. A type is a subtype of another type, if an embedding can be given in a natural way. This will be elaborated in Section 3.6. The type of Boolean streams is formed by the following observation.

$$2^\omega : Ty, \quad \langle \mathrm{head}, \mathrm{tail} \rangle : 2^\omega \to 2 \times 2^\omega$$

Like in set theory, an injective function from the subtype to its supertype (a coT of which the ScoT is a subtype) is desired. Take a map ι, which maps terms of $2^\omega_{0,1}$ to the corresponding terms in 2^ω, which is of the following type.

$$\iota : 2^\omega_{0,1} \to 2^\omega$$

Informally, ι maps a term of a ScoT to the 'equivalent' term of a coT. The precise definition of such a map is given by coiteration, described in Section 3.2. To prove injectivity of such a ι, it needs to be shown that $\iota\ x = \iota\ y$ implies $x = y$ for all $x, y : 2^\omega_{0,1}$. Take terms $x, y : 2^\omega_{0,1}$ and suppose $\iota\ x = \iota\ y$. Streams are infinitely constructed, so proving them to be equal poses a greater difficulty compared to finitely constructed structures. It is however possible to prove terms to be bisimilar, which in our definition of ScoTs (Section 3.3) will imply equality. In this case the following can be written for all $x, y : 2^\omega_{0,1}$ for which $\iota\ x = \iota\ y$ holds.

$$\mathrm{h}\ x = \mathrm{head}\ (\iota\ x) = \mathrm{head}\ (\iota\ y) = \mathrm{h}\ y$$

$$\iota\ (\mathrm{t}\ x) = \mathrm{tail}\ (\iota\ x) = \mathrm{tail}\ (\iota\ y) = \iota\ (\mathrm{t}\ y).$$

Note this is not enough to imply $x = y$ and thus prove injectivity of ι. However, this can be remedied by constructing a relation between x and y which equates the heads and relates the tails recursively. Relating the tails is saying the heads of these tails are equal and the tails of the tails are again related, and so on. Such relations will be defined and illustrated in Section 3.4. Since relating two streams is not quite the same, ι will not exactly be an injection, but an embedding. This embedding is constructed by coiteration on the observation on data of the one-after-zero Boolean streams. Section 3.2 will address coiteration and Section 3.6 will address this embedding.

3.1.2 Pairs of Natural Numbers with an Even Sum

Another example of a ScoT would be the type of pairs of natural numbers, for which the separate elements of the pairs sum up to an even number. This ScoT is written as $(\mathbb{N} \times \mathbb{N})_e$.

$$(\mathbb{N} \times \mathbb{N})_e : Ty, \quad \langle \pi_l, \pi_r \rangle : (\mathbb{N} \times \mathbb{N})_e \to \mathbb{N} \times \mathbb{N}, \quad \mathrm{o_p} : \prod_{x:(\mathbb{N}\times\mathbb{N})_e} \sum_{m:\mathbb{N}} \pi_l\ x + \pi_r\ x = m + m$$

This type is a subtype of the product type (that can be defined coinductively) over the natural numbers $\mathbb{N} \times \mathbb{N}$, which has the usual left and right projections one is used to for products, namely

$$\pi_{left} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$\pi_{right} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}.$$

Again, an injective function ι from the subtype to the supertype is desired.

$$\iota : (\mathbb{N} \times \mathbb{N})_e \to \mathbb{N} \times \mathbb{N}$$

Let for $x, y : (\mathbb{N} \times \mathbb{N})_e$, $\iota\ x = \iota\ y$. For all such $x, y : (\mathbb{N} \times \mathbb{N})_e$, the following equalities hold:

$$\pi_l\ x = \pi_{left}\ (\iota\ x) = \pi_{left}\ (\iota\ y) = \pi_l\ y$$
$$\pi_r\ x = \pi_{right}\ (\iota\ x) = \pi_{right}\ (\iota\ y) = \pi_r\ y,$$

Note there is a big difference between $(\mathbb{N}\times\mathbb{N})_e$ and $2^{\omega}_{0,1}$. $(\mathbb{N}\times\mathbb{N})_e$ is not defined recursively, so injectivity can be shown directly. The equations above imply, since $x = \langle \pi_l\ x, \pi_r\ x \rangle$ and that $y = \langle \pi_l\ y, \pi_r\ y \rangle$, ι is injective.

3.1.3 Non-terminating Computations

The type of computations over $\mathbb{N}$, written as $\mathbb{D}\mathbb{N}$, is a coT with one observation [Bas17]. Informally, this observation either returns a natural number when terminating, or continues looping by returning another computation. This behaviour of either terminating or continuing, but not both, can be captured by defining an observation into the coproduct

$$\mathbb{N} + \mathbb{D}\mathbb{N}.$$

The coT $\mathbb{D}\mathbb{N}$ is formed by this one observation.

$$\mathbb{D}\mathbb{N} : Ty, \quad \text{step} : \mathbb{D}\mathbb{N} \to \mathbb{N} + \mathbb{D}\mathbb{N}$$

Now take the type of *non-terminating* computations, written as $\mathbb{D}^\infty\mathbb{N}$. These computations will loop forever.

$$\mathbb{D}^\infty\mathbb{N} : Ty, \quad \text{st} : \mathbb{D}^\infty\mathbb{N} \to \mathbb{N} + \mathbb{D}^\infty\mathbb{N}, \quad \text{o}_\text{p} : \prod_{x:\mathbb{D}^\infty\mathbb{N}} \sum_{d:\mathbb{D}^\infty\mathbb{N}} \text{st}\ x = \kappa_R\ d$$

The ScoT of non-terminating computations is a subtype of the coT of computations by constructing an embedding between them. Like for the other examples, this will be shown later in this chapter.

3.1.4 P-adic Numbers

Let p be a prime number and

$$p_k : \mathbb{Z}/(p^{k+1}) \to \mathbb{Z}/(p^k)$$

$$p_k\ x :\equiv\ x \bmod p^k$$

be a map. The ScoT of p-adic numbers, written as $\mathbb{Z}_p$ is formed in the following way.

$$\mathbb{Z}_p : Ty, \quad \overline{\pi}_n : \mathbb{Z}_p \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n). \quad \text{o}_\text{p} : \prod_{x:\mathbb{Z}_p} \prod_{k:\mathbb{N}} \overline{\pi}_k\ x = p_k \circ \overline{\pi}_{k+1}\ x$$

This is a subtype of $\prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$, which can be understood in the following way. An n-tuple $z : \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$ can be constructed where every subsequent term in the tuple 'adds information' about the p-adic number. In other words, for $w : \mathbb{Z}_p$ the following holds.

$$z = \langle w \bmod p^0,\ w \bmod p^1,\ w \bmod p^2,\ \ldots \rangle = \langle \overline{\pi}_0\ w, \overline{\pi}_1\ w, \overline{\pi}_2\ w,\ \ldots \rangle$$

Again, a map ι can be made:

$$\iota : \ \mathbb{Z}_p \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$$

which can be shown to be injective. Let again for $x, y : \mathbb{Z}_p, \ \ \iota\ x = \iota\ y$. For all such $x, y : \mathbb{Z}_p$ and for all $k : \mathbb{N}$:

$$\overline{\pi}_k\ x = \pi_k\ (\iota\ x) = \pi_k\ (\iota\ y) = \overline{\pi}_k\ y$$

which shows $\mathbb{Z}_p$ is a subtype of $\prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$. Note again, by the lack of recursion, injectivity can be shown directly. .

3.1.5 Full Binary Non-well-founded Trees

Non-well-founded trees, or M-types, are trees representing infinite data structures [BAS15]. Non-well-founded trees are the dual of well-founded trees, or W-types [Pro13]. Among W-types are inductive types such as the natural numbers, whereas among M-types are coinductive types such as streams. These two examples of natural numbers and streams both have the property that each node in the tree only sprouts one branch. Full binary non-well-founded trees are characterised by the property of having exactly two branches sprouting out of each node.

Let $A : Ty, \ B : A \to Ty$. Take the type of full binary non-well-founded trees, written as MAB_2, is formed in the following way.

$$MAB_2 : Ty, \ \ \mathrm{i} : MAB_2 \to \sum_{a:A} (B\ a \to MAB_2), \ \ \mathrm{o_p} : \prod_{x:MAB_2} B\ (\pi_L\ (i\ x)) = 2$$

Terms $a : A$ represent the nodes of the tree and type $B\ a$ represents the 'branching factor' for each node a. The type of full binary non-well-founded trees is a subtype of the type of non-well-founded trees, formed as follows.

$$MAB : Ty, \ \ \mathrm{in} : MAB \to \sum_{a:A} (B\ a \to MAB)$$

An embedding can be constructed between these types, which will be elaborated in Section 3.6.

3.2 Towards a General Syntax for Subcoinductive Types

As discussed in Section 2.1.1, there is a certain pattern to introducing a new type. In this section, rules for formation, introduction, elimination and computation will we given. Together, these rules will define a ScoT. Before it is possible to describe these rules, the coiteration steps for data and predicates, respectively, need to be specified.

First, a mapping $F : Ty \to Ty$ needs to be defined for every (S)coT. This mapping has functor-like properties, like for every map $f : X \to Y$ between types X and Y, there exists a term $F\ f$. These terms are called monotonicity witnesses [MP08]. Furthermore, for reasons that will become apparent in Section 3.6, F needs to hold to the following functor-like rules:

$$F : Ty \to Ty$$

$$F\ f : F\ X \to F\ Y$$

$$F_\circ : \prod_{X,Y,Z:Ty} \prod_{f:X\to Y} \prod_{g:Y\to Z} F(g \circ f) = F\ g \circ F\ f$$

$$F_{id} : \prod_{X:Ty} F\ id_X = id\ (F\ X)$$

A map F abiding to these rules will from this point be called a **type-functor**.
For recursion to be possible, $F\ X$ may contain type X as a free variable.

Example 3.1. *Throughout this section, ScoT $2^\omega_{0,1}$ will be used to illustrate introduced definitions at various points. In this case, F is defined as*

$$F\ X :\equiv 2 \times X.$$

Note that due to X being contained in the type $2 \times X$, $F\ X$ can be fed into itself recursively. One can write, for example:

$$F(F(F\ X))) = 2 \times 2 \times 2 \times X.$$

This type functor F makes it possible to coiterate over the objects of a ScoT. However, for ScoTs it is also required to be able to coiterate over predicates. For this we first define two new types, one relating to coalgebras and one relating to predicates over $F\ X$ for some type X.

$$coAlg(F, X) :\equiv X \to F\ X$$

$$Pred_{FX} :\equiv F\ X \to Ty$$

The type *coAlg* is a function type which sends terms of type X to terms of types $F\ X$. Informally, such a function can be viewed as taking a coiterative step in type X. The type $Pred_{FX}$ is a function type which sends terms of $F\ X$ to types. Such a type can informally be interpreted as having a witness for a predicate over a term of $F\ X$. Having

this witness shows that there is proof of the truth of $Pred_{FX}(x)$ for a term x of $F\ X$, or in other words, the predicate holds for x.
The aim is to have a certain predicate to be true for every term of the given coalgebra. In type theoretic terms this results in a map of the following type.

$$G : \prod_{X:Ty} coAlg(F, X) \to Pred_{FX}$$

Example 3.2. *For ScoT $2^{\omega}_{0.1}$, map G is defined as*

$$G : \prod_{X:Ty} (X \to 2 \times X) \to Pred_{2\times X}$$

$$(G\ X\ d)\ (y) :\equiv \pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y)) = 1.$$

Note this G describes the core of what this predicate expresses, independent of which type term y belongs to and what coalgebra d does. If one feeds type $2^{\omega}_{0,1}$ and its observation $\langle h, t\rangle$ into this G, it results in the predicate that of the first term of the stream is a zero, the second should be a one.

$$(G\ 2^{\omega}_{0,1}\ \langle h, t\rangle)\ (\langle h, t\rangle(y)) :\equiv \pi_L(\langle h, t\rangle(y)) = 0 \to \pi_L\ (d\ (\pi_R\ (\ (\langle h, t\rangle(y)))))$$

$$= (\mathrm{h}\ y = 0 \to \pi_L\ (d\ (\mathrm{t}\ y)) = 1)$$

$$= (\mathrm{h}\ y = 0 \to \mathrm{h}\ (\mathrm{t}\ y) = 1)$$

The examples discussed in Section 3.1 exhibit a general pattern. ScoTs are formed with *observations* on both their data and their predicates separately. Furthermore, these ScoTs can be related to their supertype by relating the observations of the ScoTs to observations of the coT via a mapping. Note that in the examples of Section 3.1, the observations of the subtype and the supertype are indeed very similar.

A subcoinductive type $\nu(F, G)$ is **formed** by the type functor F and dependent map G described above and result in a ScoT together with its observations on data and predicates $\mathrm{o_d}$ and $\mathrm{o_p}$ respectively.

$$\frac{F : Ty \to Ty \quad G : \prod_{X:Ty}\ coAlg(F, X) \to Pred_{FX}}{\nu(F, G) : Ty \quad \mathrm{o_d} : \nu(F, G) \to F(\nu(F, G)) \quad \mathrm{o_p} : \prod_{x:\nu(F,G)}(G\ \nu(F, G)\ \mathrm{o_d})\ (\mathrm{o_d}\ x)}\ \mathrm{F}_{\nu(F,G)}$$

Example 3.3. *ScoT $2^{\omega}_{0,1}$ is formed by the following formation rule*

$$\frac{F : Ty \to Ty \qquad G : \prod_{X:Ty}(X \to 2 \times X) \to Pred_{2\times X}}{2^{\omega}_{0,1} : Ty \quad \langle \mathrm{h}, \mathrm{t}\rangle : 2^{\omega}_{0,1} \to 2 \times 2^{\omega}_{0,1} \quad \mathrm{o_p} : \prod_{x:2^{\omega}_{0,1}} \mathrm{h}\ x = 0 \to \mathrm{h}\ (\mathrm{t}\ x) = 1}\ \mathrm{F}_{2^{\omega}_{0,1}}$$

Here, h *denotes the map which returns the first term of a given stream and* t *denotes the map which provides the given stream with the first term removed. The predicate* $\mathrm{o_p}$ *states that for all streams of* $2^\omega_{0,1}$*, if the first term is a zero, then the term following this zero is a one. Note that since* t *returns another stream in* $2^\omega_{0,1}$*, this predicate is checked recursively over the whole stream.*

Inhabitants of coTs are introduced by coiteration. Coiteration can be thought of, in a categorical sense, as the map from a coalgebra into a final coalgebra.
For general coTs, the only thing requiring specification is what happens to terms under application of observation and the coiteration scheme. For ScoTs, however, what happens to its predicates needs to be specified also.
For coiteration of data, the coiteration principle for general coTs [Bas18] only has to be modified slightly.

Coiteration for a **coinductive type** $\nu(F)$ is defined, given some type X and an observation on X, $d :\equiv \langle d_1, d_2, \ldots d_n \rangle$, by the following scheme:

$$\frac{X : Ty \quad d : X \to FX}{coiter^{\nu(F)}\ d : X \to \nu(F).}\ \mathrm{I}_{\nu(F)}$$

Example 3.4. *For coT* 2^ω*, a supertype of ScoT* $2^\omega_{0,1}$*, coiteration is given by*

$$\frac{X : Ty \quad d : X \to 2 \times X}{coiter^{2^\omega}\ d : X \to 2^\omega.}\ \mathrm{I}_{2^\omega}$$

Coiteration states that given some type X and a (recursive) observation on X, a map *into* $\nu(F)$ can be made which is dependent on observation d, namely $coiter^{\nu(F)}\ d$. For coiteration of data of ScoTs, coiteration on coTs can be borrowed. However, for a complete definition of coiteration on ScoTs, one needs to find a way to define coiteration on predicates as well. Using map G, general coiteration on ScoTs can be given by:

$$\frac{X : Ty \quad d : X \to FX \quad q : \prod_{x:X}(G\ X\ d)\ (d\ x)}{coiter^{\nu(F,G)} d\ q : X \to \nu(F,G)}$$

$$\frac{X : Ty \quad d : X \to FX \quad q : \prod_{x:X}(G\ X\ d)\ (d\ x)}{coiter^{\nu(F,G)}_{Pred} : \prod_{x:X}(G\ X\ d)\ (d\ x) \to (G\ \nu(F,G)\ \mathrm{o_d})\ (\mathrm{o_d}\ (coiter^{\nu(F,G)} d\ q\ (x)))}$$

Coiteration for the data and the predicates are defined by separate maps, but they are both conclusions of the same prerequisites.

Example 3.5. *For ScoT $2^{\omega}_{0,1}$, coiteration can be given by*

$$\frac{X : Ty \quad d :\equiv \langle d_1, d_2\rangle : X \to 2 \times X \quad q : \prod_{x:X} d_1\ x = 0 \to d_1\ d_2\ x = 1}{coiter^{2^{\omega}_{0,1}}\ d\ q\ :\ X \to 2^{\omega}_{0,1}}$$

$$\frac{X : Ty \quad d :\equiv \langle d_1, d_2\rangle : X \to 2 \times X \quad q : \prod_{x:X} d_1\ x = 0 \to d_1\ d_2\ x = 1}{coiter^{2^{\omega}_{0,1}}_{Pred}\ d\ q : \prod_{x:X}(d_1\ x = 0 \to d_1\ d_2\ x = 1) \to (\mathrm{h}\ (coiter^{2^{\omega}_{0,1}}\ d\ q\ (x)) = 0 \to \mathrm{h}\ \mathrm{t}\ (coiter^{2^{\omega}_{0,1}}\ d\ q\ (x)) = 1)}$$

Note that at the moment map G takes an argument in $F\ X$. For a definition of coiteration, it would be preferable if G took an argument in X and to make d a part of the map instead of the argument. This would make the definition of q more straightforward. This modification can be made easily by noticing that for maps

$$d : X \to FX,$$

$$\mathrm{o_d} : \nu(F, G) \to F(\nu(F, G))$$

and

$$coiter^{\nu(F,G)}\ d\ q : X \to \nu(F, G)$$

it is possible to explicitly define substitution maps (see example 2.7):

$$d^* : Pred_{FX} \to Pred_X,$$

$$\mathrm{o_d}^* : Pred_{F(\nu(F,G))} \to Pred_{\nu(F,G)}$$

and

$$(coiter^{\nu(F,G)}\ d\ q)^* : Pred_{\nu(F,G)} \to Pred_X$$

respectively. These substitution functors can be used to rewrite G.

$$(G\ X\ d)\ (d\ x) = d^*\ (G\ X\ d)\ x$$

$$(G\ \nu(F, G)\ \mathrm{o_d})\ (\mathrm{o_d}\ ((coiter^{\nu(F,G)}\ d\ q)\ x)) = \mathrm{o_d}^*\ (G\ \nu(F, G)\ \mathrm{o_d})((coiter^{\nu(F,G)}\ d\ q)\ x)$$
$$= (coiter^{\nu(F,G)}\ d\ q)^*(\mathrm{o_d}^*(G\ \nu(F, G)\ \mathrm{o_d}))\ x$$

Example 3.6. *For ScoT $2^{\omega}_{0,1}$, coiteration is given by*

$$X : Ty \quad d :\equiv \langle d_1, d_2\rangle : X \to 2 \times X$$

$$\frac{q : \prod_{x:X} d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y))) = 1)\ x}{coiter^{2^{\omega}_{0,1}}\ d\ q\ :\ X \to 2^{\omega}_{0,1},\quad \langle \mathrm{h}, \mathrm{t}\rangle : 2^{\omega}_{0,1} \to 2 \times 2^{\omega}_{0,1}}\ \mathrm{I}_{2^{\omega}_{0,1}}$$

$$X : Ty \quad d :\equiv \langle d_1, d_2 \rangle : X \to 2 \times X$$

$$\frac{q : \prod_{x:X} d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y))) = 1)\ x}{coiter^{2^\omega_{0,1}}_{Pred}\ d\ q : \prod_{x:X} d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y)) = 1))\ x \to} \ \mathrm{I}_{2^\omega_{0,1}}$$

$$(coiter^{2^\omega_{0,1}}\ d\ q)^*\langle \mathrm{h}, \mathrm{t}\rangle^*(\lambda y : 2 \times 2^\omega_{0,1}.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y)) = 1))\ x.$$

Note that only the types of the coiteration maps are provided by coiteration, what these maps actually *do* is not clear yet. The specification of the actual behaviour of the coiteration maps will be given by computation rules. In Section 2.1.1, it is mentioned that computation rules specify how elimination acts on the introduction rules. In the case of (S)coTs, elimination is given by observation(s) $\mathrm{o_d}$ (and $\mathrm{o_p}$). The computation rules for data and predicates respectively is as follows:

$$\mathrm{o_d}\ (coiter^{\nu(F,G)}\ d\ q\ x) :\equiv F\ (coiter^{\nu(F,G)}\ d\ q)\ (d\ x)$$

$$coiter^{\nu(F,G)}_{Pred}\ d\ q\ x :\equiv \mathrm{o_p}\ (coiter^{\nu(F,G)}\ d\ q\ x).$$

Example 3.7. *For ScoT $2^\omega_{0,1}$, the computation rules are defined as follows:*

$$\langle h, t\rangle(coiter^{2^\omega_{0,1}}\ d\ q\ x) :\equiv \langle \mathrm{h}(coiter^{2^\omega_{0,1}}\ d\ q\ x),\ \mathrm{t}(coiter^{2^\omega_{0,1}}\ d\ q\ x)\rangle$$

$$coiter^{2^\omega_{0,1}}_{Pred}\ d\ q\ x\ (q\ x) :\equiv \mathrm{h}\ (coiter^{2^\omega_{0,1}}\ d\ q\ x) = 0 \to \mathrm{h}\ (\mathrm{t}\ (coiter^{2^\omega_{0,1}}\ d\ q\ x)) = 1.$$

What this computation states is that data is still part of the ScoT after the coiteration step and that after a term of ScoT has gone through coiteration, the given predicate still holds on this term after the mapping.

3.3 General Syntax

The combination of the rules for formation, introduction, elimination and computation, discussed in Section 3.2 give a complete definition of the new type called a subcoinductive type.

Definition 3.1. A **subcoinductive type** is defined by its rules of **formation**

$$\frac{F : Ty \to Ty \qquad G : \prod_{x:X} coAlg(F, X) \to Pred_{FX}}{\nu(F,G) : Ty \quad \mathrm{o_d} : \nu(F,G) \to F(\nu(F,G)) \quad \mathrm{o_p} : \prod_{x:\nu(F,G)} \mathrm{o_d}^*(G\ \nu(F,G)\ \mathrm{o_d})\ x}\ \mathrm{F}_{\nu(F,G)},$$

rules of **elimination** of type

$$\mathrm{o_d} : \nu(F,G) \to F(\nu(F,G))$$

$$\mathrm{o_p} : \prod_{x:\nu(F,G)} \mathrm{o_p}^*(G\ \nu(F,G)\ \mathrm{o_d})\ x,$$

rules of **coiteration**

$$\frac{X : Ty \quad d : X \to FX \quad q : \prod_{x:X} d^*(G\ X\ d)\ x}{coiter^{\nu(F,G)} d\ q : X \to \nu(F,G)}\ \mathrm{I}_{\nu(F,G)}$$

$$\frac{X : Ty \quad d : X \to FX \quad q : \prod_{x:X} d^*(G\ X\ d)\ x}{coiter^{\nu(F,G)}_{Pred}\ d\ q : \prod_{x:X} d^*(G\ X\ d)\ x \to (coiter^{\nu(F,G)}\ d\ q)^*(\mathrm{o_d}^*(G\ \nu(F,G)\ \mathrm{o_d}))\ x}\ \mathrm{I}_{\nu(F,G)}$$

and rules of **computation**

$$\mathrm{o_d}\ (coiter^{\nu(F,G)}\ d\ q\ x) :\equiv F\ (coiter^{\nu(F,G)}\ d\ q)\ (d\ x)$$

$$coiter^{\nu(F,G)}_{Pred}\ d\ q\ x\ (q\ x) :\equiv \mathrm{o_p}\ (coiter^{\nu(F,G)}\ d\ q\ x).$$

All the components of this definition will be illustrated on the examples of Section 3.1.

3.3.1 One-after-zero Boolean Streams

Since this type was used as the illustrative example throughout Section 3.2, only a brief overview will be provided.
ScoT $2^{\omega}_{0,1}$ is formed by the observation consisting of taking the head and tail of the a

Boolean stream and the predicate that every zero in the stream should be followed by a one.

$$\frac{F : Ty \to Ty \qquad G : \prod_{X:Ty}(X \to 2 \times X) \to Pred_{2\times X}}{2^\omega_{0,1} : Ty \quad \langle \mathrm{h}, \mathrm{t}\rangle : 2^\omega_{0,1} \to 2 \times 2^\omega_{0,1} \quad \mathrm{o_p} : \prod_{x:2^\omega_{0,1}} \mathrm{h}\ x = 0 \to \mathrm{h}\ (\mathrm{t}\ x) = 1}\ \mathrm{F}_{2^\omega_{0,1}}$$

Here:

$$F\ X :\equiv 2 \times X$$
$$(G\ X\ d)(y) :\equiv \pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y))) = 1.$$

and the rules of elimination are of type

$$\langle \mathrm{h}, \mathrm{t}\rangle : 2^\omega_{0,1} \to 2 \times 2^\omega_{0,1}$$
$$\mathrm{o_p} : \prod_{x:2^\omega_{0,1}} \mathrm{h}\ x = 0 \to \mathrm{h}\ (\mathrm{t}\ x) = 1.$$

Coiteration is given by

$$\frac{\begin{array}{c} X : Ty \quad d :\equiv \langle d_1, d_2\rangle : X \to 2 \times X \\ q : \prod_{x:X} d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y)) = 1)\ x \end{array}}{coiter^{2^\omega_{0,1}}\ d\ q\ :\ X \to 2^\omega_{0,1},\ \ \langle \mathrm{h}, \mathrm{t}\rangle : 2^\omega_{0,1} \to 2 \times 2^\omega_{0,1}}\ \mathrm{I}_{2^\omega_{0,1}}$$

$$\frac{\begin{array}{c} X : Ty \quad d :\equiv \langle d_1, d_2\rangle : X \to 2 \times X \\ q : \prod_{x:X} d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y))) = 1)\ x \end{array}}{\begin{array}{l} coiter^{2^\omega_{0,1}}_{Pred}\ d\ q : \prod_{x:X}(d^*(\lambda y : 2 \times X.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y))) = 1))\ x \to \\ (coiter^{2^\omega_{0,1}}\ d\ q)^*(\langle \mathrm{h}, \mathrm{t}\rangle^*(\lambda y : 2 \times 2^\omega_{0,1}.\pi_L\ y = 0 \to \pi_L\ (d\ (\pi_R\ y)) = 1)))\ x. \end{array}}\ \mathrm{I}_{2^\omega_{0,1}}$$

The only thing left to define is the way observations $\langle h, t\rangle$ and d act on coiteration. This is expressed by its computation rules.

$$\langle h, t\rangle(coiter^{2^\omega_{0,1}}\ d\ q\ x) :\equiv \langle \mathrm{h}(coiter^{2^\omega_{0,1}}\ d\ q\ x),\ \mathrm{t}(coiter^{2^\omega_{0,1}}\ d\ q\ x)\rangle$$

$$coiter^{2^\omega_{0,1}}_{Pred}\ d\ q\ x\ (q\ x) :\equiv \mathrm{h}\ (coiter^{2^\omega_{0,1}}\ d\ q\ x) = 0 \to \mathrm{h}\ (\mathrm{t}\ (coiter^{2^\omega_{0,1}}\ d\ q\ x)) = 1$$

3.3.2 Pairs of Natural Numbers with an Even Sum

ScoT $(\mathbb{N} \times \mathbb{N})_e$ is formed by two projections mapping the left and right terms of the pairs onto a natural number. The predicate on the data of this ScoT, is that the outputs of these projections must sum up to an even natural number. Since sums are easier to define than products in type $\mathbb{N}$ [Pro13], the choice has been made to write $m + m$ instead of $2m$.

$$\frac{F : Ty \to Ty \qquad G : \prod_{x:X}(X \to \mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N} \to Ty)}{(\mathbb{N} \times \mathbb{N})_e : Ty \quad \langle \pi_l, \pi_r \rangle : (\mathbb{N} \times \mathbb{N})_e \to \mathbb{N} \times \mathbb{N}} \ \mathrm{F}_{(\mathbb{N}\times\mathbb{N})_e}$$

$$\mathrm{o_p} : \prod_{x:(\mathbb{N}\times\mathbb{N})_e} \sum_{m:\mathbb{N}} \pi_l \ x + \pi_r \ x = m + m$$

Here:

$$F \ X :\equiv \mathbb{N} \times \mathbb{N}$$
$$(G \ X \ d) \ (y) :\equiv \sum_{m:\mathbb{N}} \pi_L \ y + \pi_R \ y = m + m,$$

with elimination rules of type:

$$\langle \pi_l, \pi_r \rangle : (\mathbb{N} \times \mathbb{N})_e \to \mathbb{N} \times \mathbb{N}$$
$$\mathrm{o_p} : \prod_{x:(\mathbb{N}\times\mathbb{N})_e} \sum_{m:\mathbb{N}} \pi_l \ x + \pi_r \ x = m + m.$$

Coiteration for $(\mathbb{N} \times \mathbb{N})_e$ is given by:

$$\frac{X : Ty \quad d :\equiv \langle d_1, d_2 \rangle : X \to \mathbb{N} \times \mathbb{N} \qquad q : \prod_{x:X} d^*(\lambda y : \mathbb{N} \times \mathbb{N}. \sum_{m:\mathbb{N}} \pi_L \ y + \pi_R \ y = m + m) \ x}{coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q : X \to (\mathbb{N} \times \mathbb{N})_e, \quad \langle \pi_l, \pi_r \rangle : (\mathbb{N} \times \mathbb{N})_e \to \mathbb{N} \times \mathbb{N}} \ \mathrm{I}_{(\mathbb{N}\times\mathbb{N})_e}$$

$$\frac{\begin{array}{c} X : Ty \quad d :\equiv \langle d_1, d_2 \rangle : X \to \mathbb{N} \times \mathbb{N} \\ q : \prod_{x:X} d^*(\lambda y : \mathbb{N} \times \mathbb{N}. \sum_{m:\mathbb{N}} \pi_L \ y + \pi_R \ y = m + m) \ x \end{array}}{\begin{array}{l} coiter^{(\mathbb{N}\times\mathbb{N})_e}_{Pred} \ d \ q : \prod_{x:X} d^*(\lambda y : \mathbb{N} \times \mathbb{N}. \sum_{m:\mathbb{N}} \pi_L \ y + \pi_R \ y = m + m) \ x \to \\ (coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q)^*(\langle \pi_l, \pi_r \rangle^*(\lambda y : \mathbb{N} \times \mathbb{N}. \sum_{m:\mathbb{N}} \pi_L \ y + \pi_R \ y = m + m)) \ x, \end{array}} \ \mathrm{I}_{(\mathbb{N}\times\mathbb{N})_e}$$

which makes our computation rules look like:

$$\langle \pi_l, \pi_r \rangle \ (coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q \ x) :\equiv \langle \pi_l \ (coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q \ x), \pi_r(coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q \ x) \rangle$$
$$(coiter^{(\mathbb{N}\times\mathbb{N})_e}_{Pred} \ d \ q \ x \ (q \ x)) :\equiv \sum_{m:\mathbb{N}} \pi_L \ (coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q \ x) + \pi_R \ (coiter^{(\mathbb{N}\times\mathbb{N})_e} \ d \ q \ x) = m+m.$$

3.3.3 Non-terminating Computations

ScoT $\mathbb{D}^\infty\mathbb{N}$ is formed by one observation which takes a step in the computation. This step either returns the output of the computation, which in this case is a natural number, or it keeps looping by returning a new computation. The observation on predicates states that a computation of type $\mathbb{D}^\infty\mathbb{N}$ will be looping forever.

$$\frac{F : Ty \to Ty \quad G : \prod_{x:X}(X \to \mathbb{N} + X) \to (\mathbb{N} + X \to Ty)}{\mathbb{D}^\infty\mathbb{N} : Ty \quad \mathrm{st} : \mathbb{D}^\infty\mathbb{N} \to \mathbb{N} + \mathbb{D}^\infty\mathbb{N} \quad \mathrm{o_p} : \prod_{x:\mathbb{D}^\infty\mathbb{N}} \sum_{w:\mathbb{D}^\infty\mathbb{N}} \mathrm{st}\ x = \kappa_R\ w}\ \mathrm{F}_{\mathbb{D}^\infty\mathbb{N}}$$

Here:

$$F\ X :\equiv \mathbb{N} + X$$
$$(G\ X\ d)(y) = \sum_{w:\mathbb{D}^\infty\mathbb{N}} y = \kappa_R\ w,$$

with elimination rules of type :

$$\mathrm{st} : \mathbb{D}^\infty\mathbb{N} \to \mathbb{N} + \mathbb{D}^\infty$$
$$\mathrm{o_p} : \prod_{x:\mathbb{D}^\infty\mathbb{N}} \sum_{w:\mathbb{D}^\infty\mathbb{N}} \mathrm{st}\ x = \kappa_R\ w.$$

Coiteration for $\mathbb{D}^\infty\mathbb{N}$ is given by:

$$\frac{X : Ty \quad d : X \to \mathbb{N} + X \quad q : \prod_{x:X} d^*(\lambda y : \mathbb{N} + X.\, \sum_{w:\mathbb{D}^\infty\mathbb{N}} y = \kappa_r\ w)\ x}{coiter^{\mathbb{D}^\infty\mathbb{N}} d\ q : X \to \mathbb{D}^\infty\mathbb{N}, \quad \mathrm{st} : \mathbb{D}^\infty\mathbb{N} \to \mathbb{N} + \mathbb{D}^\infty\mathbb{N}}\ \mathrm{I}_{\mathbb{D}^\infty\mathbb{N}}$$

$$\frac{X : Ty \quad d : X \to \mathbb{N} + X \quad q : \prod_{x:X} d^*(\lambda y : \mathbb{N} + X.\, \sum_{w:\mathbb{D}^\infty\mathbb{N}} y = \kappa_r\ w)\ x}{coiter^{\mathbb{D}^\infty\mathbb{N}}_{Pred}\ d\ q : \prod_{x:X} d^*(\lambda y : \mathbb{N} + X.\, \sum_{w:\mathbb{D}^\infty\mathbb{N}} y = \kappa_r\ w)\ x \to (coiter^{\mathbb{D}^\infty\mathbb{N}}\ d\ q)^*(\mathrm{st}^*(\lambda y : \mathbb{N} + \mathbb{D}^\infty\mathbb{N}.\, \sum_{w:\mathbb{D}^\infty\mathbb{N}} y = \kappa_r\ w))\ x,}\ \mathrm{I}_{\mathbb{D}^\infty\mathbb{N}}$$

which makes our computation rules look like:

$$\mathrm{st}(coiter^{\mathbb{D}^\infty\mathbb{N}}\ d\ q\ x) :\equiv \mathrm{st}(coiter^{\mathbb{D}^\infty\mathbb{N}}\ d\ q\ x)$$
$$coiter^{\mathbb{D}^\infty\mathbb{N}}_{Pred}\ d\ q\ x\ (q\ x) :\equiv \sum_{w:\mathbb{D}^\infty\mathbb{N}} (coiter^{\mathbb{D}^\infty\mathbb{N}}\ d\ q\ x) = \kappa_R\ w$$

It is clear that specifying the first coiteration step here is not necessary, since $\mathrm{o_d}$ consists of only one observation. It is written anyway, to keep consistency between the different examples.

3.3.4 P-adic Numbers

ScoT $\mathbb{Z}_p$ is formed by one observation mapping a p-adic number to an infinite tuple, where every consecutive number in the tuple is a more precise approximation of the mentioned p-adic number. The predicate on this ScoT states that the n^{th} approximation in this list is the same as the $(n-1)^{\text{th}}$ approximation modulo p^n.

$$\frac{F : Ty \to Ty \qquad G : \prod_{x:X}(X \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)) \to (\prod_{n:\mathbb{N}} \mathbb{Z}/(p^n) \to Ty)}{\mathbb{Z}_p : Ty \quad \overline{\pi}_n : \mathbb{Z}_p \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n) \quad \mathrm{o_p} : \prod_{x:\mathbb{Z}_p} \prod_{k:\mathbb{N}} \overline{\pi}_k\ x = p_k \circ \overline{\pi}_{k+1}\ x.}\ \mathrm{F}_{\mathbb{Z}_p}$$

Here:

$$F\ X :\equiv \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$$
$$(G\ X\ d)(y) :\equiv \prod_{k:\mathbb{N}} \pi_k\ y = p_k \circ \pi_{k+1}\ y,$$

with elimination rules of type:

$$\overline{\pi}_n : \mathbb{Z}_p \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$$
$$\mathrm{o_p} : \prod_{x:\mathbb{Z}_p} \prod_{k:\mathbb{N}} \overline{\pi}_k\ x = p_k \circ \overline{\pi}_{k+1}\ x$$

Coiteration for $\mathbb{Z}_p$ is given by:

$$\frac{X : Ty \quad d : X \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n) \qquad q : \prod_{x:X} d^*(\lambda y : \prod_{n:\mathbb{N}} \mathbb{Z}_p.\ \prod_{k:\mathbb{N}} \pi_k\ y = p_k \circ \pi_{k+1}\ y)\ x}{coiter^{\mathbb{Z}_p} d\ q : X \to \mathbb{Z}_p, \quad \overline{\pi}_n : \mathbb{Z}_p \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)}\ \mathrm{I}_{\mathbb{Z}_p}$$

$$\frac{\begin{array}{c} X : Ty \quad d : X \to \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n) \\ q : \prod_{x:X} d^*(\lambda y : \prod_{n:\mathbb{N}} \mathbb{Z}_p.\ \prod_{k:\mathbb{N}} \pi_k\ y = p_k \circ \pi_{k+1}\ y)\ x \end{array}}{\begin{array}{c} coiter^{\mathbb{Z}_p}_{Pred}\ d\ q : \prod_{x:X} d^*(\lambda y : \prod_{n:\mathbb{N}} \mathbb{Z}_p.\ \prod_{k:\mathbb{N}} \pi_k\ y = p_k \circ \pi_{k+1}\ y)\ x \to \\ (coiter^{\mathbb{Z}_p}\ d\ q)^*(\overline{\pi}^*_n(\lambda y : \prod_{n:\mathbb{N}} .\ \prod_{k:\mathbb{N}} \pi_k\ y = p_k \circ \pi_{k+1}\ y))\ x, \end{array}}\ \mathrm{I}_{\mathbb{Z}_p}$$

which makes our computation rules look like:

$$\overline{\pi}_n(coiter^{\mathbb{Z}_p}\ d\ q\ x) :\equiv \overline{\pi}_n(coiter^{\mathbb{Z}_p}\ d\ q\ x)$$
$$(coiter^{\mathbb{Z}_p}_{Pred}\ d\ q\ x\ (q\ x)) :\equiv \prod_{k:\mathbb{N}} \pi_k\ (coiter^{\mathbb{Z}_p}\ d\ q\ x) = p_k \circ \pi_k\ (coiter^{\mathbb{Z}_p}\ d\ q\ x)$$

3.3.5 Full Binary Non-well-founded Trees

ScoT MAB_2 is formed by the observation that for each node, a map from each branch coming from said node to a new tree is made. The observation on predicates tells us that the number of branches needs to be 2, always.

$$\frac{F : Ty \to Ty \qquad G : \prod_{x:X}(X \to \sum_{a:A}(B\ a \to X)) \to (\sum_{a:A}(B\ a \to X) \to Ty)}{\begin{array}{c} MAB_2 : Ty \quad \mathrm{i} : MAB_2 \to \sum_{a:A}(B\ a \to MAB_2) \\ \mathrm{o_p} : \prod_{x:MAB_2} B\ (\pi_L\ (i\ x)) = 2 \end{array}}\ \mathrm{F}_{MAB_2}$$

Here:

$$F\ X :\equiv \sum_{a:A}(B\ a \to X)$$
$$(G\ X\ d)(y) :\equiv B(\pi_L(y)) = 2,$$

with elimination rules of type:

$$\mathrm{i} : MAB_2 \to \sum_{a:A}(B\ a \to MAB_2)$$
$$\mathrm{o_p} : \prod_{x:MAB_2} B\ (\pi_L\ (i\ x)) = 2$$

Coiteration for MAB_2 is given by:

$$\frac{\begin{array}{c} X : Ty \quad d : X \to \sum_{a:A}(B\ a \to X) \\ q : \prod_{x:X} d^*(\lambda y : \sum_{a:A}(B\ a \to X).B(\pi_L(y)) = 2)\ x \end{array}}{coiter^{MAB_2} d\ q\ : X \to MAB_2,\ \mathrm{i} : MAB_2 \to \sum_{a:A}(B\ a \to MAB_2)}\ \mathrm{I}_{MAB_2}$$

$$\frac{\begin{array}{c} X : Ty \quad d : X \to \sum_{a:A}(B\ a \to X) \\ q : \prod_{x:X} d^*(\lambda y : \sum_{a:A}(B\ a \to X).B(\pi_L(y)) = 2)\ x \end{array}}{\begin{array}{c} coiter^{MAB_2}_{Pred} d\ q : \prod_{x:X} d^*(\lambda y : \sum_{a:A}(B\ a \to X).B(\pi_L(y)) = 2)\ x \to \\ (coiter^{MAB_2}\ d\ q)^*(\mathrm{i}^*(\lambda y : \sum_{a:A}(B\ a \to MAB_2).B(\pi_L(y)) = 2))\ x, \end{array}}\ \mathrm{I}_{MAB_2}$$

which makes our computation rules look like:

$$\mathrm{i}(coiter^{MAB_2}\ d\ q\ x) :\equiv \mathrm{i}(coiter^{MAB_2}\ d\ q\ x)$$
$$(coiter^{MAB_2}_{Pred}\ d\ q\ x\ (q\ x)) :\equiv \prod_{x:MAB_2} B\ (\pi_L\ (i\ ((coiter^{MAB_2}\ d\ q)\ x))) = 2.$$

3.4 Bisimilarity

Due to their infinite construction, it is difficult to directly determine whether or not two (S)coTs are equal. Nevertheless, to be able to compare (S)coTs, a concept called *bisimulation* is borrowed from the field of transition systems. Two terms of a (S)coT are to be called bisimilar if a bisimulation can be made between them. Such a bisimulation is a relation between (S)coTs relating their behaviour to each other. First, one needs a type-theoretic definition of a relation.

Definition 3.2. The type of **relations** $\mathcal{R}_X$ on type X is defined as

$$\mathcal{R}_X :\equiv X \to X \to Ty.$$

In order to make this relation a bisimulation, the relation has 'survive' the coiteration step. One way of achieving this is relating the relation before the application of d and after the application of d. This can be done by something called relation lifting.

Definition 3.3. A **lifting** of a type functor F to another type functor $\overline{F}$, is defined as a dependent map

$$\overline{F} : \prod_{X:Ty} \mathcal{R}_X \to \mathcal{R}_{FX}.$$

For this thesis, it is required to supply such a lifting together with the functor F of the ScoT being introduced. For many functors, there are techniques to construct such a lifting [Sta11]. However, these existing techniques do not readily apply to type theory. The conception of such a technique is beyond the scope of this thesis.

Using map $d : X \to F\ X$, the following substitution map can be defined

$$d^{\#} : \mathcal{R}_{FX} \to \mathcal{R}_X$$

$$d^{\#}\ R :\equiv \lambda x.\lambda y.R\ (d\ x)\ (d\ y)$$

due to which the following statements can be equated.

$$\overline{F}\ \nu(F,G)\ R\ (d\ x)\ (d\ y) = d^{\#}\ (\overline{F}\ \nu(F,G)\ R)\ x\ y$$

Note

$$d^{\#} \circ (\overline{F}\ X) : \mathcal{R}_X \to \mathcal{R}_X$$

were $\overline{F}$ is independent of the coalgebra d.

Definition 3.4. Define a map which takes as arguments two relations on the same $X : Ty$ and returns another type. Note this can be seen as defining a relation on two relations over X.

$$\sqsubseteq : \mathcal{R}_X \to \mathcal{R}_X \to Ty$$
$$R \sqsubseteq S :\equiv \prod_{x,y:X} R\ x\ y \to S\ x\ y$$

$R : \mathcal{R}_{\nu(F,G)}$ is a **bisimulation** if there exists a term b of the following type

$$b : R \sqsubseteq d^{\#}(\overline{F}\ \nu(F,G)\ R).$$

Define $Bisim_{(X,d)} : Ty$, with (X, d) a coalgebra, as the type containing pairs of a relation over X and the proof of this relation being a bisimulation.

$$Bisim_{(X,d)} :\equiv \sum_{R:\mathcal{R}_X} R \sqsubseteq d^{\#}(\overline{F}\ \nu(F,G)\ R)$$

This $\overline{F}$ will be provided for the examples in Section 3.1 and for $2^{\omega}_{0,1}$, a specific example for a bisimulation will be provided.

Example 3.8. *For ScoT $2^{\omega}_{0,1}$, with $F\ X :\equiv 2 \times X$, $\overline{F}$ is given by:*

$$\overline{F} : \prod_{X:Ty} \mathcal{R}_X \to \mathcal{R}_{2\times X}$$

$$\overline{F}\ X\ R\ u\ v :\equiv \langle \pi_L\ u = \pi_L\ v, R\ (\pi_R\ u)\ (\pi_R\ v)\rangle$$

Let us look at two streams u and v of $2^{\omega}_{0,1}$, which are defined differently. u is defined by coiteration, such that:

$$\mathrm{h}\ u :\equiv 0$$
$$\mathrm{h}\ (\mathrm{t}\ u) :\equiv 1$$
$$\mathrm{t}\ (\mathrm{t}\ u) :\equiv u$$

v will be a stream which is made out of combining two other streams x and y of $2^{\omega}_{0,1}$, where

$$\mathrm{head}\ x :\equiv 0$$
$$\mathrm{tail}\ x :\equiv x$$

$$\mathrm{head}\ y :\equiv 1$$
$$\mathrm{tail}\ y :\equiv y.$$

Note that x can be seen as the stream of only ones and y can be seen as the stream of only zeros. Now the following relation is defined.

$$S\ x\ y :\equiv \prod_{n:\mathbb{N}} \text{head}\ (\text{tail}^n\ x) = 0 \to \text{head}\ (\text{tail}^n\ y) = 1$$

Furthermore, with this relation S, a function zip *can be defined. This function alternates in taking terms of two streams forms a new stream.*

$$\text{zip} : \prod_{x,y:2^\omega_{0,1}} S\ x\ y \to 2^\omega_{0,1}$$

$$\text{h}\ (\text{zip}\ x\ y\ p) :\equiv \text{head}\ x$$

$$\text{h}\ (\text{t}\ (\text{zip}\ x\ y\ p)) \equiv \text{head}\ y$$

$$\text{t}\ (\text{t}\ (\text{zip}\ x\ y\ p)) :\equiv \text{zip}\ (\text{tail}\ x)\ (\text{tail}\ y)\ (\lambda\ n.p\ (n+1)).$$

Now, v is defined as

$$v :\equiv \text{zip}\ x\ y\ (\lambda\ n.\text{refl}_{2^\omega})$$

These streams $u, v : 2^\omega_{0,1}$ can now be related by the following relation.

$$R\ u\ v :\equiv (\text{h}\ u = \text{h}\ v)\ \times\ (\text{h}\ (\text{t}\ u) = \text{h}\ (\text{t}\ v))\ \times\ (\text{t}\ (\text{t}\ u) = u)\ \times\ (\text{t}\ (\text{t}\ v) = v)$$

To prove this is a bisimulationm it needs to be shown that

$$\prod_{u,v:2^\omega_{0,1}} R\ u\ v \to \langle \pi_L\ u = \pi_L\ v, R\ (\pi_R\ u)\ (\pi_R\ v)$$

This will be illustrated briefly. The left part of $R\ u\ v$, already ensures $\text{h}\ u = \text{h}\ v$. *The right part shows that the heads of the tails are again equal, and that the tails of the tails are again related, so $R\ (\text{t}\ u)\ (\text{t}\ v))$ holds.*

Example 3.9. *For $(\mathbb{N} \times \mathbb{N})_e$, with $F\ X :\equiv \mathbb{N} \times \mathbb{N}$, $\overline{F}$ is given by:*

$$\overline{F} : \prod_{X:Ty} \mathcal{R}_X \to \mathcal{R}_{\mathbb{N}\times\mathbb{N}}$$

$$\overline{F}\ X\ R\ u\ v :\equiv \langle \pi_l\ u = \pi_l\ v, \pi_r\ u = \pi_r\ v\rangle$$

Example 3.10. *For $\mathbb{D}^\infty\mathbb{N}$, with $F\ X :\equiv \mathbb{N} + X$, $\overline{F}$ is given by:*

$$\overline{F} : \prod_{x:Ty} \mathcal{R}_X \to \mathcal{R}_{\mathbb{N}+X}$$

Since a relation on a coproduct is required, elimination and computation of coproducts (see Section 2.1.8) need to be used. First, the following maps are defined:

$$f : \mathbb{N} \to \mathbb{N} + X \to Ty$$

$$f\ n :\equiv \mathrm{el}_+\ (\lambda m.n = m)\ (\lambda x.\mathbb{0})$$

$$g : X \to \mathbb{N} + X \to Ty$$

$$g\ x :\equiv \mathrm{el}_+\ (\lambda n.\mathbb{0})\ (\lambda y.R\ x\ y)$$

Map f enforces relation between computations only if they return the same natural number upon termination and map g enforces relation between computation only if both computations loop the same number of times. With these maps, elimination and computation for coproducts can be used again to define the desired lifting.

$$\overline{F}\ X\ R\ u\ v :\equiv (\mathrm{el}_+\ f\ g)\ u\ v$$

Example 3.11. *For $\mathbb{Z}_p$, with $F\ X :\equiv \prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)$, $\overline{F}$ is given by:*

$$\overline{F} : \prod_{x:Ty} \mathcal{R}_X \to \mathcal{R}_{\prod_{n:\mathbb{N}} \mathbb{Z}/(p^n)}$$

$$\overline{F}\ X\ R\ u\ v :\equiv \prod_{n:\mathbb{N}} \overline{\pi}_n\ u = \overline{\pi}_n\ v$$

Example 3.12. *For MAB_2, with $F\ X :\equiv \sum_{a:A}(B\ a \to X)$, $\overline{F}$ is given by:*

$$\overline{F} : \prod_{x:Ty} \mathcal{R}_X \to \mathcal{R}_{\sum_{a:A}(B\ a \to X)}$$

$\overline{F}\ X\ R\ u\ v :\equiv (\pi_L\ (\mathrm{i}\ (u)) = \pi_L\ (\mathrm{i}\ (v))) \times R\ (\pi_R\ \mathrm{i}\ (u)\ (B\ (\pi_L\ (\mathrm{i}\ (u)))))\ (\pi_R\ (\mathrm{i}\ (v))\ (B\ (\pi_L\ (\mathrm{i}\ (v)))))$.

It would be useful to add some concept that describes the 'largest' bisimulation on some ScoT. In set theory, this would be done by defining every bisimulation to be included in the largest bisimulation. In type theory, however, proofs of inclusion can differ. Instead, the definition of bisimilarity is formulated in terms of finality of coalgebras.

Definition 3.5. Bisimilarity is a final coalgebra over a type functor of type

$$\mathrm{o_d}^{\#} \circ (\overline{F}\ \nu(F, G)) : \mathcal{R}_{\nu(F,G)} \to \mathcal{R}_{\nu(F,G)}.$$

Note this definition makes bisimilarity a coinductive type.

3.5 Coinduction principle

In order to avoid the requirement for ScoTs to be coTs themselves directly, another requirement is added called *coinduction*. Coinduction requires every bisimulation is a bisimilarity, implying uniqueness of coiteration.

Definition 3.6. Let $Bisim_{(\nu(F,G),\mathrm{o_d})}$ be the type of bisimulations (as defined in section 3.4) on some ScoT $\nu(F,G)$. **Coinduction** is defined by the dependent map:

$$coind : \prod_{R:Bisim_{(\nu(F,G),\mathrm{o_d})}} \pi_L\ R\ \sqsubseteq\ =$$

Note this requirement states that there exists a map from any bisimulation on a ScoT to bisimilarity. With the assumption of coinduction, coiteration will be shown to be unique for two examples from section 3.1. One example with a recursive structure and one without.

Example 3.13. *Let* $f :\equiv coiter^{2^\omega_{0,1}}\ d\ q : X \to 2^\omega_{0,1}$ *and* $f' : X \to 2^\omega_{0,1}$ *both be coalgebra morphisms. This means:*

$$\mathrm{h}\ (f\ x) = d_1\ x, \quad \mathrm{t}\ (f\ x) = f\ (d_2\ x)$$

$$\mathrm{h}\ (f'\ x) = d_1\ x, \quad \mathrm{t}\ (f'\ x) = f'\ (d_2\ x).$$

Take the following relation on $2^\omega_{0,1}$.

$$R\ u\ v :\equiv \sum_{x:X} (u = f\ x)\ \times\ (v = f'\ x)$$

Evidently, $R\ (f\ x)\ (f'\ x)$ *holds, since*

$$\langle X, \mathrm{refl}_{2^\omega_{0,1}}, \mathrm{refl}_{2^\omega_{0,1}} \rangle : R\ (f\ x)\ (f'\ x)$$

is a proof of that particular statement. For elaboration on $\mathrm{refl}_{2^\omega_{0,1}}$*, see Section 2.1.2. Furthermore,* R *is a bisimulation. Recall*

$$d^\#\ (\overline{F}\ 2^\omega_{0,1}\ R)\ u\ v = (\overline{F}\ 2^\omega_{0,1}\ R)\ (d\ u)\ (d\ v)$$

$$= \langle \pi_L\ (d\ u) = \pi_L\ (d\ v),\ R\ (\pi_R\ (d\ u))\ (\pi_R\ (d\ v)) \rangle$$

$$= \langle \mathrm{h}\ u = \mathrm{h}\ v, R\ (\mathrm{t}\ u)\ (\mathrm{t}\ v) \rangle$$

The proof for this is in finding a term for proposition:

$$\pi_R\ Bisim_{(2^\omega_{0,1},\langle \mathrm{h},\mathrm{t}\rangle)}\ R \equiv \prod_{u,v:2^\omega_{0,1}} R\ u\ v \to \mathrm{h}\ u = \mathrm{h}\ v\ \times\ R\ (\mathrm{t}\ u)\ (\mathrm{t}\ v)\rangle$$

It is already established f and f' are homomorphisms, which means the following terms exist:

$$hom_f^{h,d} : \prod_{x:X} \mathrm{h}\ (f\ x) = d_1\ x$$
$$hom_{f'}^{h,d} : \prod_{x:X} \mathrm{h}\ (f'\ x) = d_1\ x$$
$$hom_f^{t,d} : \prod_{x:X} \mathrm{t}\ (f\ x) = f\ (d_2\ x)$$
$$hom_{f'}^{t,d} : \prod_{x:X} \mathrm{t}\ (f'\ x) = f'\ (d_2\ x)$$

which leads to the definition of the map

$$r\ (f\ x)\ (f'\ x)\ (\langle X, \mathrm{refl}_{2^\omega_{0,1}}, \mathrm{refl}_{2^\omega_{0,1}}\rangle) :\equiv \langle p_1, p_2\rangle$$

with:

$$p_1 :\equiv (hom_f^{h,d}) \circ (hom_{f'}^{h,d})^{-1}$$
$$p_2 :\equiv \langle d_2\ X, hom_f^{t,d}, hom_{f'}^{t,d}\rangle.$$

and

$$\langle R, \langle p_1, p_2\rangle\rangle : Bisim_{(2^\omega_{0,1},\mathrm{o_d})}\ R.$$

This can be seen as follows, p_1 shows for all $x : X$:

$$\mathrm{h}\ (f\ x) = d_1\ x = \mathrm{h}\ (f'\ x)$$

and p_2 shows for all $x : X$:

$$\mathrm{t}\ (f\ x) = f\ (d_2\ x)$$
$$\mathrm{t}\ (f'\ x) = f'\ (d_2\ x)$$

which proves:

$$R\ (\mathrm{t}\ (f\ x))\ (\mathrm{t}\ (f'\ x))$$

with:

$$R : d_2\ X \to d_2\ X \to Ty.$$

Since $f\ x$ and $f'x$ are bisimular for all $x : X$, coinduction states:

$$\prod_{x:X} f\ x = f'\ x$$

which implies:

$$f \sim f'.$$

This states coiteration is unique up to homotopy.

Example 3.14. *Let $f :\equiv coiter^{(\mathbb{N}\times\mathbb{N})_e}\ d\ q : X \to (\mathbb{N} \times \mathbb{N})_e$ and $f' : X \to (\mathbb{N} \times \mathbb{N})_e$ be coalgebra morphisms. Take again the relation:*

$$R\ u\ v :\equiv \sum_{x:X} (u = f\ x)\ \times\ (v = f'\ x),$$

with the proof for $R\ (f\ x)\ (f'\ x)$ given by $\langle X, \mathrm{refl}_{(\mathbb{N}\times\mathbb{N})_e}, \mathrm{refl}_{(\mathbb{N}\times\mathbb{N})_e}\rangle$. To show R is a bismulation, the following term has to be constructed.

$$r : \pi_R(Bisim_{(\mathbb{N}\times\mathbb{N})_e}\ R)$$

This term is given by:

$$r\ (f\ x)\ (f'\ x)\ (\langle X, \mathrm{refl}_{(\mathbb{N}\times\mathbb{N})_e}, \mathrm{refl}_{(\mathbb{N}\times\mathbb{N})_e}\rangle) :\equiv \langle p_1, p_2\rangle$$

with :

$$p_1 :\equiv (hom_f^{\pi_l,d}) \circ (hom_{f'}^{\pi_l,d})^{-1}$$
$$p_2 :\equiv (hom_f^{\pi_r,d}) \circ (hom_{f'}^{\pi_r,d})^{-1}$$

and

$$hom_f^{\pi_l,d} : \prod_{x:X} \pi_l\ (f\ x) = d_1\ x$$
$$hom_{f'}^{\pi_l,d} : \prod_{x:X} \pi_l\ (f'\ x) = d_1\ x$$
$$hom_f^{\pi_r,d} : \prod_{x:X} \pi_r\ (f\ x) = d_2\ x$$
$$hom_{f'}^{\pi_r,d} : \prod_{x:X} \pi_r\ (f'\ x) = d_2\ x$$

following from the fact f and f' are homomorphisms. Because for all $x : X$, $f\ x$ and $f'\ x$ are bisimular, by coinduction it can be concluded that:

$$f \sim f',$$

which makes coiteration unique up to homotopy.

For the rest of the examples, a proof will not be provided, since they are long and all use the same principle. All the proofs boil down to defining the specific relation on $\nu(F, G)$:

$$R\ u\ v :\equiv \sum_{x:X} (u = f\ x)\ \times (v = f'\ x)$$

and proving this relation to be a bisimulation. The generality of this technique makes it easy to apply to other ScoTs.

3.6 Embedding

The properties described in sections above do make ScoTs types with special properties, but they are not yet subtypes. The only part that has not been addressed is the 'sub' part. For a ScoT to really be a subtype of a coT, one needs to be able to construct an embedding between the first and the second, analogous to how an injective map can be constructed between subsets and their supersets.
It is desired for the defined ScoT to be the *final* subobject (with reference to its predicates) of a coT, in the same way a coT is a final coalgebra.

Theorem 3.1. *Let Ty be the category of types.*
Let

$$F : Ty \to Ty$$

be a type functor and

$$G : \prod_{x:X} (X \to F\ X) \to (F\ X \to Ty)$$

be a dependent function mapping a coalgebra to a predicate on $F\ X$.
Furthermore, let $\nu(F, G) : Ty$ be a ScoT and $\nu(F) : Ty$ be a coT. By coiteration on o_d, the following embedding can be made of type:

$$\mathrm{em} : \nu(F, G) \to \nu(F).$$

This embedding is unique by coinduction. Let $X : Ty$ be some type with $d : X \to F\ X$ and $q : \prod_{x:X} d^*(G\ X\ d)\ x$.
Then for every coalgebra morphism

$$f_c : X \to \nu(F)$$

there is an unique coalgebra morphism

$$f_s : X \to \nu(F, G)$$

for which

$$f_c = \mathrm{em} \circ f_s$$

holds, making the diagram below commute.

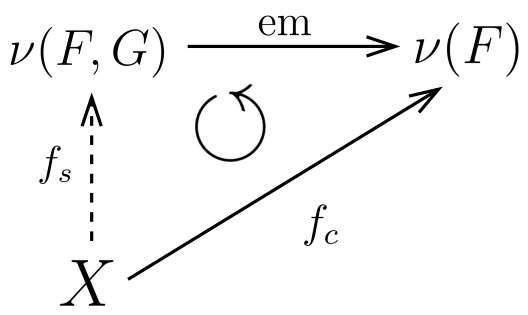

Figure 11: Commuting diagram, embedding relation a ScoT and a coT

This diagram has a familiar form. It may look like $\nu(F, G)$ together with morphism em forms a limit (see Section 2.2.3) over the shape containing only $\nu(F)$, but this is not the case. The limit over a shape with only one object it the object itself.

Proof. Let $\nu(F, G) : Ty$ and $\mathrm{o_d} : \nu(F, G) \to F(\nu(F, G))$. By coiteration on $\mathrm{o_d}$ there exists a coalgebra morphism:

$$\mathrm{em} : \nu(F, G) \to \nu(F).$$

By coinduction, this map em is unique.

Let $X : Ty$ and $d : X \to F\ X$. By coiteration there exists a coalgebra morphism:

$$f_c : X \to \nu(F).$$

By coinduction, f_c is unique.

Let $X : Ty$ and $d : X \to F\ X$. By coiteration there exists a morphism:

$$f_s : X \to \nu(F, G).$$

By coinduction, f_s is unique.

The theorem can be proven by proving the diagram below commutes.

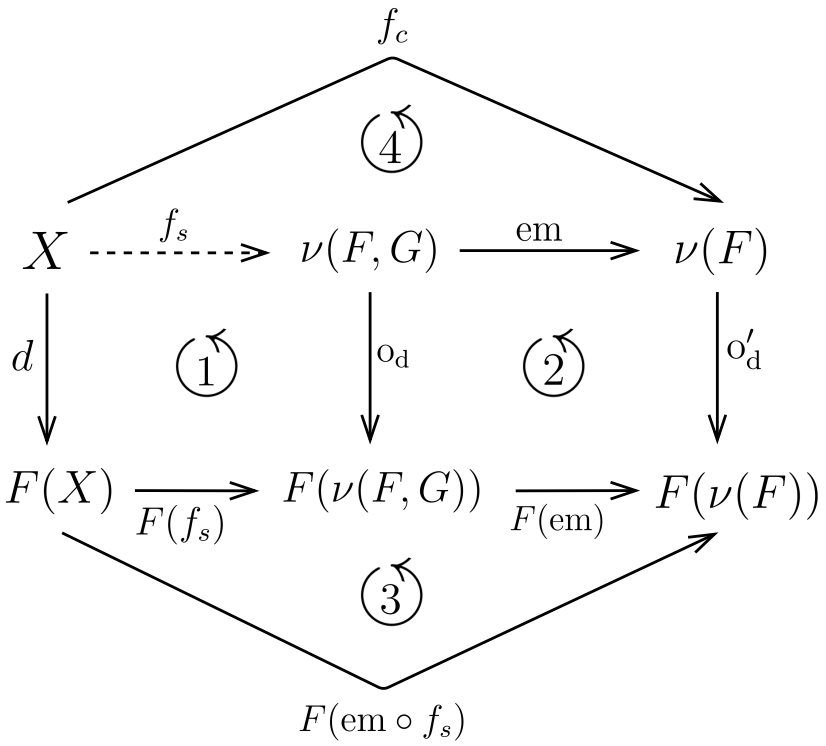

Figure 12: Diagram which if commutes, proves Theorem 3.1.

1. This square commutes because the way f_s is constructed by coiteration on d, making f_s a homomorphism.

2. This square commutes because the way em is constructed by coiteration on $\mathrm{o_d}$, making em a homorphism.

   The combination of these first and second squares also makes $\mathrm{em} \circ f_s$ a homomorphism.

3. This triangle commutes because of the functor properties F has (see Section 3.2).

   The outer edges of the diagram make f_c a homorphism, namely

   $$\mathrm{o_d}' \circ f_c = F(\mathrm{em} \circ f_s) \circ d = F(f_c) \circ d$$

4. Coinduction holds for $\nu(F)$, so it has to be the case that

   $$f_c = \mathrm{em} \circ f_s.$$

   Note because one is using coinduction on coTs, not ScoTs, this is an actual equation, not a homotopy. This means this last triangle also commutes (up to homotopy) and the theorem holds for ScoTs.

□

4 Comparison to Σ-Types over Truncated Propositions

The concept of subtypes is not a new one. As is to be expected, other definitions of subtypes already exist. One of these definitions involves Σ-types over truncated propositions. In summary, terms of these subtypes are pairs consisting of first a term of its supertype and second a proof the predicate (or proposition) that the first term fulfils.

4.1 Alternative Subtypes

In set theory, a subset $B \subseteq A$ could be defined as all the elements of A which satisfy a certain condition. In type theory, there is a similar construction.
Subtype B of $A : Ty$ could be defined as the Σ-type $\sum_{a:A} P(a)$ where $P : A \to \mathcal{U}$ is a type family. This would mean B contains terms $\langle a, p_a \rangle$ for those $a : A$ which meet the predicate described by P, so $p_a : P(a)$.

Example 4.1. *Let $Prime : Ty$ be the type of prime numbers [Fri14]. Let $\mathbb{N}_p$ be the type of natural numbers which can be written as the sum of two primes. This subtype of $\mathbb{N}$ can be written as the following Σ-type:*

$$\mathbb{N}_p :\equiv \sum_{n:\mathbb{N}} \sum_{p,q:Prime} p + q = n.$$

This example shows some flaws in the way subtypes are defined at present. For any $a : A$, the type $P(a)$ could have more than one inhabitant, giving more than one term in B for the corresponding term in A. This could result in subtypes having a cardinality greater than their supertype, a property undesired for a definition of a subtype.

4.2 Mere Propositions and Propositional Truncation

In type theory, wherein types can be seen as propositions or theorems, a proof is able to contain more information than just the theorem being 'true' or 'false'. To work with type theory, as one would with classical logic, it might be tempting to define a type P to be 'true' when it has an inhabitant and 'false' if it does not. However, this leads to contradiction [Pro13]. A better way to achieve this, is defining all types of P to be 'equal', enforcing proof irrelevance. This means the specific term proving P used, does not matter.

Definition 4.1. Type P is a **mere proposition** if for all $x, y : P$, $x = y$. In other words, type $\prod_{x,y:P}(x = y)$ has an inhabitant. Note this makes every mere proposition contractible (see Section 2.1.9).

It seems that requiring the type family P in section 4.1 to be a mere proposition would be a solution to the cardinality problem. This would, however, restrict many possible subtypes from being real subtypes. Instead of requiring P to be a mere proposition to begin with, it could be 'transformed' into a mere proposition, keeping the information of 'having an inhabitant', but forgetting all additional information about the specific inhabitants. This can be done using propositional truncation [Pro13].

Definition 4.2. Let $P : Ty$ be a type. The **propositional truncation** $\|P\|$ of P is built by two constructors and one elimination rule:

- For every $p : P$ there is a $|p| : \|P\|$.
- For every $q, r : \|P\|$, $q = r$ holds
- For $f : X \to Y$, there is an induced $g : \|X\| \to Y$, such that for all $x : X$: $g(\|x\|) \equiv f(x)$.

4.3 Definition Subtype using Σ-Types

With propositional truncation, a definition for subtypes can be given with the desired properties.

Definition 4.3. Type $B : Ty$ **subtype** of $A : Ty$ is defined as the dependent pair:

$$B :\equiv \sum_{a:A} \|P(a)\|$$

Where $P : A \to \mathcal{U}$ is a type family and $\|P\|$ its propositional truncation.

Since for every $a : A$, $\|P(a)\|$ is a mere proposition, every $b : B$ has exactly one correspondent in A.

Example 4.2. *We revisit the previous example, subtype $\mathbb{N}_p$ of $\mathbb{N}$:*

$$\mathbb{N}_p :\equiv \sum_{n:\mathbb{N}} \sum_{p,q:Prime} p + q = n.$$

Note that both:

$$\langle 10, \langle 5, 5\rangle\rangle : \mathbb{N}_p$$

and

$$\langle 10, \langle 3, 7\rangle\rangle : \mathbb{N}_p,$$

which is a problem. If instead $\mathbb{N}_p$ *is defined as:*

$$\mathbb{N}_p :\equiv \sum_{n:\mathbb{N}} \left\| \sum_{p,q:Prime} p + q = n \right\|,$$

there would only be one pair of the form

$$\langle 10, p_{10} \rangle : \mathbb{N}_p$$

with some $p_{10} : \|P(10)\|$*. It does not matter 'which' term of* $\|P(10)\|$ *since for* $|\langle 5, 5\rangle|, |\langle 3, 7\rangle| : \|P(10)\|$ *the following holds:* $|\langle 5, 5\rangle| = |\langle 3, 7\rangle|$*.*

4.4 Comparison

With an existing definition for subtypes, it may seem superfluous to make another one just for coTs. The definition from Chapter 3, however, lends itself better to recursive structures, which is very useful for coTs. Moreover, in the definition of Chapter 3, ScoTs are built from the ground up, not from its supertype. This lends more flexibility, since the supertype does not have to be specified in advance.

Below, most of the examples from Section 3.1 will be written as $\Sigma-$types over truncated propositions, to illustrate the usefulness of the definition of Chapter 3.

Example 4.3. *Consider ScoT* $2^{\omega}_{0,1}$*. This could also be written as:*

$$2^{\omega}_{0,1} :\equiv \sum_{x:2^{\omega}} \left\| \prod_{n:\mathbb{N}} \text{head tail}^n\ x = 0 \to \text{head tail}^{n+1}\ x = 1 \right\|$$

Note that the proposition in this case already is a mere proposition, so truncation is not actually needed, but it is kept there to be consistent with the definition.

Example 4.4. *Consider ScoT* $(\mathbb{N} \times \mathbb{N})_e$*. Since this ScoT is not recursive, the alternative formulation works pretty well.*

$$(\mathbb{N} \times \mathbb{N})_e :\equiv \sum_{x:\mathbb{N}\times\mathbb{N}} \left\| \sum_{m:\mathbb{N}} \pi_L\ x + \pi_R\ x = m + m \right\|$$

Note that again, the propositional truncation here is unnecessary because $\sum_{m:\mathbb{N}} \pi_L\ x + \pi_R\ x = m + m$ *is also already a mere proposition.*

Example 4.5. *Consider the ScoT* $\mathbb{D}^\infty\mathbb{N}$*. Defining this ScoT is again harder in the alliterative way, since it does not lend itself well for recursion.*

$$\mathbb{D}^\infty\mathbb{N} :\equiv \sum_{x:\mathbb{DN}} \left\| \prod_{n:\mathbb{N}} \sum_{y:\mathbb{DN}} y = (\kappa_r \ \mathrm{step})^n \ x \right\|$$

Example 4.6. *Consider the ScoT* MAB_2*. Since the recursion for trees is more complicated than the recursion in the previous examples, we will first work out a few steps:*

1. $B(\pi_L(\mathrm{in}(x))) = \mathbb{2}$
2. $B((\pi_L(\mathrm{in}((\pi_R(\mathrm{in}(x)))(B(\pi_L(\mathrm{in}(x))))))))) = \mathbb{2}$
3. $B((\pi_L(\mathrm{in}((\pi_R(\mathrm{in}((\pi_R(\mathrm{in}(x))))(B(\pi_L(\mathrm{in}(x))))))(B(\pi_L(\mathrm{in}((\pi_R(\mathrm{in}(x)))$
   $(B(\pi_L(\mathrm{in}(x)))))))))))) \ = \ \mathbb{2}$

Note that in every step, x *is replaced by* $(\pi_R \ (\mathrm{i}(x)))(B(\pi_L(i(x))))$*. This means at step* n*, the equation contains* 2^n x*'s, as is to be expected when 'building' a (full) binary tree. This growth of the number of variables on which recursion takes place makes this example different from the others, where every* x *'produces' at most one other* x*. This is also the reason why reworking this type to a non-recursive form is very hard, maybe even impossible, which makes the* $\sum$*-definition of subtypes unsuitable.*

The last example presents the necessity of the new subtype definition for coTs. Recursion is present in most coTs and may get too complicated to write in an iterative way, making it impossible to write subtypes as a Σ-type over a truncated proposition.

5 Conclusion and Discussion

In this thesis, a new technique for constructing subtypes for coTs was introduced and shown to be more useful for dealing with recursion compared to the technique already known. Furthermore, useful categorical concepts, such as substitution functors, have been translated into type theoretical concepts.

First, a general syntax for ScoTs was conceived, being ScoTs are types defined by their observations and the predicates on their terms. Furthermore, part of the general pattern of introducing a new type was followed, as rules for introduction, elimination and computation were introduced. Bisimilation and bisimilarity have been formulated and a coinduction principle has been provided, which states bisimulation in ScoTs implies equality. A theorem was formulated making embedding of ScoTs into coTs precise, where the ScoT together with this embedding form a limit over the coT acting as the supertype. Finally, these new ScoTs where compared to conventional subtypes over coTs and it has been shown ScoTs are more suitable in describing subtypes for coTs since they are better equipped to deal with recursion.

There are some ways in which this theory can be improved. At the moment, neither observation or computation on predicates is formulated recursively regardless of the recursive nature of the ScoT for which these conditions hold. It would be interesting to reformulate these conditions in a way that the recursive nature of the ScoT shines through.
Furthermore, every new subtype has to be supplied with a lifting $\overline{F}$ for the given functor F (Section 3.4). For future work, it would be interesting to extend this theory by developing a technique which gives such a lifting $\overline{F}$ for any given type functor F.
Finally, the current technique only allows ScoTs to be subtypes of coTs over the same coalgebra. The current definition does not allow, for example, an appropriate embedding between $2^{\omega}_{0,1}$ and $\mathfrak{Z}^{\omega}$. Extending the definition of ScoTs with some transformation map allowing these kinds of embeddings would be an interesting addition to the theory.
Another thing to look into would be limits, more specifically equalisers, in the category of types. We suspect that a ScoT, together with an embedding into a coT can act as an equaliser.

References

[BAS15] Paolo Capriotti Benedikt Ahrens and Régis Spadotti. Non-wellfounded trees in homotopy type theory. *International Conference on Typed Lambda Calculi and Application*, 13:17–31, 2015.

[Bas17] Henning Basold. Type theory based on dependent inductive and coinductive types. Workshop on Coinduction in Type Theory Université Savoie, Bouget du Lac, 2017.

[Bas18] Henning Basold. *Mixed Inductive-Coinductive Reasoning*. PhD thesis, Radboud Universiteit Nijmegen, 2018.

[Esc18] Martín H. Escardó. A self-contained, brief and complete formulation of voevodsky's univalence axiom. 2018.

[Fri14] Tobias Fritz. Elementary number theory in type theoretic foundations. unpublished notes, 2014.

[Geu] Herman Geuvers. Introduction to type theory.

[Jac99] Bart Jacobs. *Catagorical Logic and Type Theory*. Elsevier, 1999.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:220–259, 1997.

[Lei14] Tom Leinster. *Basic Category Theory*. Cambridge University Press, 2014.

[LS88] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1988.

[MP08] Favio E. Miranda-Perea. Two extentions of system f with (co)iteration and primitive (co)recursion principles. *RAIRO - Theoretical Informatics and Applications*, 43:703–766, 2008.

[Pro13] The Univalent Foundations Program. *Homotopy Type Theory*. Instutite for Advanced Study, 2013.

[Sta11] Sam Staton. Relating coalagebraic notions of bisimulation. *Logical Methods in Computer Science*, 7:1–21, 2011.

[Sut13] Andrew V. Sutherland. Lecture notes introduction to arithmetic geometry, September 2013.