A Realisability Model of Dependent Type Theory with Sizes

MSc Thesis *(Afstudeerscriptie)*

written by

Bastiaan Laarakker
(born September 2nd, 2001 in Telki, Hungary)

under the supervision of **dr. Benno van den Berg** and **Daniël Otten MSc**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
August 26, 2025	dr. Benno van den Berg (supervisor)
	dr. Malvin Gattinger (chair)
	dr. Tom de Jong
	Daniël Otten MSc (supervisor)
	dr. Niels van der Weide

Institute for Logic, Language and Computation

Abstract

Many proof assistants are based on dependent type theories, and extend the theory with support for defining inductive and coinductive types. For their underlying theory to remain consistent and for type checking to be decidable, proof assistants ensure termination of recursive functions and productivity of corecursive functions using syntactic guardedness conditions. This syntactic approach is often too restrictive, limiting the expressiveness of the language. Sized types are a type-based alternative to determining termination or productivity of (co)recursive functions. With sized types, (co)inductive types can be annotated with additional size information, and a well-founded ordering on these sizes is used to prove termination or productivity.

This work investigates the syntax and semantics of sized types. We construct a realisability model for a dependent type theory with a type of sizes and an impredicative universe. We make use of a form of internal parametricity via axioms, describing how sizes should be treated uniformly. These are validated by the model, thereby also showing how realisability models allow for parametric reasoning. Impredicativity lets us derive initial algebras and terminal coalgebras for all polynomial functors via existential and universal quantification over suitable size-indexed (co)algebras, improving on previous work on encoding (co)inductive types using sized types. We also discuss what is needed for a consistent and robust implementation of sized types, and note some connections with guarded type theory, another type-based approach to productivity checking.

Contents

Chapter 1

Introduction

1.1 Dependent Type Theory and Totality

Dependent type theory can be viewed from different perspectives, one of which exhibits type theory as an alternative foundation to set theory for formalising mathematics. By the Curry-Howard correspondence, we interpret types as logical propositions, and terms of types as proofs of such propositions. A key desirable property of dependent type theory is that type checking, checking whether a given term has a certain type, is decidable. This allows for a different perspective on dependent type theory, namely as a dependently-typed programming language, where dependent types allow for more expressivity in the typing system. Since type checking is decidable in these languages, proofs of propositions can be automatically verified when implemented in such dependently-typed languages, as this corresponds exactly to type checking.

Many important structures are defined as *inductive types* in type theory, which we may view as types that are freely generated by a set of constructors. In dependent type theory, such inductive types come with an elimination principle, specifying how terms of inductive types are used, and thus how functions from an inductive type may be defined. Dependently-typed languages such as Agda and Rocq often include more features in their language, and instead allow for functions to be defined using recursion, from which elimination principles can be derived.

However, allowing for unrestricted recursion would lead to non-terminating functions being definable in the language, which could lead to an inconsistent underlying theory and undecidable type checking. Clearly, this would interfere with the role of such dependently-typed programming languages as a proof assistant. Thus, to serve as a proof assistant, a dependently-typed language should be *total*: all recursive functions must be terminating, where termination refers to normalisation with respect to the reduction rules of the theory. Since it is generally undecidable whether a recursive program terminates, termination checkers must approximate this problem. This leads to a balance between accepting as many recursive functions as possible, without accepting any non-terminating programs. Thus, the requirement of totality necessarily restricts the expressiveness of such languages, since there will exist programs that are terminating but which will not be accepted by the termination checker.

More recently, support for *coinductive types*, which represent infinite data, has also been developed in several proof assistants. Dually to how recursive equations are defined on inductive types, corecursive equations are used to construct coinductive data. Such functions, constructing infinite data, are not terminating by their nature. However, proof assistants still require a restriction on corecursive functions, namely that they are *productive*: any finite prefix of the solution to the equation must be generated in finite time. Here the same issue occurs; there will exist corecursive programs that are productive, but which will not be accepted by the productivity checker. Thus, developing good methods for termination- and productivity checking are important for increasing the expressiveness of such dependently-typed languages, while ensuring that type checking remains decidable and the theory consistent.

In the dependently-typed language Agda, which will be our main example, both termination and productivity are generally enforced via syntactic checks on (co)recursive equations. In the simplest method of ensuring totality, recursive equations on inductive data must be *guarded-by-destructors*, which means that recursive calls must be made on direct subterms [Gim95]. Dually, productivity is guaranteed by enforcing that corecursive equations are *guarded-by-constructors*, meaning that every corecursive call must occur directly within a data constructor. In fact, the termination checker for Agda makes use of a variant of the size-change principle [LJB01], which allows more general structural recursion where recursive calls may be on any structurally smaller term. This principle works by representing function calls in size-change graphs, describing which parameters decrease according to the subterm relation. A functional program is then represented by a possibly infinite sequence of such graphs. If such an infinite sequence contains an infinitely descending chain, the program must terminate, since the subterm relation is well-founded.

However, syntactic methods suffer from several issues. Programs with more complex patterns of (co)recursion are not easily defined in a way that satisfies the guardedness condition. Moreover, syntactic methods are sensitive to slight reformulations of programs, and do not work well with higher-order programs. Another method worth mentioning for guaranteeing termination is via structural recursion on user-defined well-founded orderings [Ler24], or even weaker orderings such as in [VCW12]. Although flexible, these methods pollute the program and associated proofs by requiring it to include a certification of termination.

It is clear that having a robust method, which is also modular and compositional, for ensuring termination and productivity is important. This work is concerned with a type-based approach to termination checking, sized types.

1.2 Sized Types

Sized types [Abe06; AVW17; HPS96] offer an alternative approach to both termination- and productivity checking, in which they are guaranteed through typing derivations. In this approach, both inductive and coinductive types can be annotated with additional size information. We may think of sizes as *ordinals*, which thus come with a *well-founded ordering*. For simple inductive types, the size annotation given to a term can be thought of as an ordinal bounding the height of the data structure. For coinductive types, the size can be viewed as an ordinal specifying the depth up to which a term may be inspected, i.e. the observation depth. Sizes give the termination checker more information for tracking descent along some well-founded ordering, and thus integrate well with the size-change principle. We begin by giving some examples of sized types.

1.2.1 Working with Sized Types

In this section, we provide a practical perspective on sized types by giving some examples on how they aid termination checking. This will also lead us to some of the limitations of sized types. Since there currently only exists an implementation of sized types in Agda [Abe12], we will use an Agda-like syntax to convey the examples informally.

A typical concrete example is the type of size-indexed lists. Recall that for any type A, we may define an inductive type $\mathsf{List}\,A$ with constructors $\mathsf{nil} : \mathsf{List}\,A$ and $\mathsf{cons} : A \to \mathsf{List}\,A \to \mathsf{List}\,A$. Now suppose we have some type Size of sizes together with a well-founded ordering $<$ on Size and a successor operation $\uparrow$ such that $s < \uparrow s$. Then we may define an Size-indexed type of lists $\mathsf{SList}\,A\,s$, where $s : \mathsf{Size}$. Following the intuition that annotating an inductive type with a size s means that all terms of the type should be of *at most* size s, we introduce the following constructors:

$$\begin{aligned} \mathsf{nil} &: \mathsf{SList}\,A\,s \\ \mathsf{cons} &: \forall r < s.\ A \to \mathsf{SList}\,A\,r \to \mathsf{SList}\,A\,s \end{aligned}$$

Thus, to construct a list of size at most s, the list may be empty or we may extend a list of size $r < s$ by adding an element to the front of the list. In the case of lists, "size" can simply be read as length.

The benefit of annotating these types with sizes is that the ordering $<$ on sizes is well-founded. This allows for well-founded recursion on sizes: to show that a recursive function on a size-indexed inductive type terminates, it suffices to show that the size decreases in each recursive call. This removes the need for a syntactic check on terms in recursive functions and delegates the proof of termination to the type system; the termination certificate is given by the typing derivation.

It is instructive to see an example of a recursive function on a sized inductive type. To do so, first recall that we can define a lmap function on the ordinary list type, which maps a function f over the elements of the list:

$$\begin{aligned}
&\mathsf{lmap} : (A \to B) \to \mathsf{List}\, A \to \mathsf{List}\, B \\
&\mathsf{lmap}\, f\, \mathsf{nil} = \mathsf{nil} \\
&\mathsf{lmap}\, f\, (\mathsf{cons}\, a\, as) = \mathsf{cons}\, (f a)\, (\mathsf{lmap}\, f\, as)
\end{aligned}$$

Note that this definition is structurally recursive, because the recursive call on lmap occurs on a direct subterm of $\mathsf{cons}\, a\, as$. Thus, sized types are not needed to convince the termination checker that this function terminates. Now let us consider a more complex inductive type, namely that of rose trees, where nodes may have any finite number of children. The type $\mathsf{Rose}\, A$ has a single constructor

$$\mathsf{node} : A \to \mathsf{List}\, (\mathsf{Rose}\, A) \to \mathsf{Rose}\, A$$

Intuitively, we could also define a map function for rose trees by reusing the map function for lists as follows:

$$\begin{aligned}
&\mathsf{rmap} : (A \to B) \to \mathsf{Rose}\, A \to \mathsf{Rose}\, B \\
&\mathsf{rmap}\, f\, (\mathsf{node}\, a\, as) = \mathsf{node}\, (f a)\, (\mathsf{lmap}\, (\mathsf{rmap}\, f)\, as)
\end{aligned}$$

However, the termination checker of Agda will not accept this definition, because it is not structurally recursive, since the recursive call of rmap is not directly on a subterm of $\mathsf{node}\, a\, as$. To alleviate this problem, we introduce a size-indexed variant of rose trees, $\mathsf{SRose}\, s\, A$, with the following constructor:

$$\mathsf{node} : \forall r < s.\ A \to \mathsf{List}\, (\mathsf{Rose}\, A\, r) \to \mathsf{Rose}\, A\, s$$

The intuition behind the constructor here is: if each child of a node has height at most r for some $r < s$, then the node itself has height at most s. Now we can redefine the function rmap on the type of sized rose trees as:

$$\begin{aligned}
&\mathsf{rmap} : (s : \mathsf{Size}) \to (A \to B) \to \mathsf{SRose}\, A\, s \to \mathsf{SRose}\, B\, s \\
&\mathsf{rmap}\, s\, f\, (\mathsf{node}\, r\, a\, as) = \mathsf{node}\, (f\, a)\, (\mathsf{lmap}\, (\mathsf{rmap}\, r\, f)\, as)
\end{aligned}$$

Informally, $\mathsf{rmap}\, s$ is well-defined and terminating because lmap is known to be terminating, and the recursive call of rmap is now on r, where $r < s$. Thus, by well-foundedness of sizes, we cannot have an infinite sequence of recursive calls.

As an example for coinductive types, we might consider the type $\mathsf{Stream}\, A$ of streams over A. To work with size-indexed coinductive types, we will informally make use of copatterns [Abe+13], which allow a coinductive type to be defined by specifying its destructors rather than its constructors. For the type $\mathsf{Stream}\, A$, the destructors are

$$\begin{aligned}
&\mathsf{head} : \mathsf{Stream}\, A \to A \\
&\mathsf{tail} : \mathsf{Stream}\, A \to \mathsf{Stream}\, A
\end{aligned}$$

As with lists, we may define a sized type of streams $\mathsf{SStream}\, A\, s$, with the following destructors:

$$\begin{aligned}
&\mathsf{head} : \mathsf{SStream}\, A\, s \to A \\
&\mathsf{tail} : \mathsf{SStream}\, A\, s \to \forall r < s.\ \mathsf{SStream}\, A\, r
\end{aligned}$$

The tail destructor may yield any stream of depth less than s, since a term of $\mathsf{Stream}\, A\, s$ may only be safely inspected up to depth s. We may define a zipWith function on size-indexed streams by specifying the destructors as follows:

$$\begin{aligned}
&\mathsf{zipWith} : (s : \mathsf{Size}) \to (A \to B \to C) \to \\
&\qquad \mathsf{SStream}\, A\, s \to \mathsf{SStream}\, B\, s \to \mathsf{SStream}\, C\, s \\
&\mathsf{head}\, (\mathsf{zipWith}\, s\, f\, x\, y) = f\, (\mathsf{head}\, x)\, (\mathsf{head}\, y) \\
&\mathsf{tail}\, (\mathsf{zipWith}\, s\, f\, x\, y)\, r = \mathsf{zipWith}\, r\, f\, ((\mathsf{tail}\, x)\, r)\, ((\mathsf{tail}\, y)\, r)
\end{aligned}$$

We can use the type of size-indexed streams and our definition of zipWith to define the Fibonnaci sequence:

$$\begin{aligned}
&\mathsf{fib} : (s : \mathsf{Size}) \to \mathsf{SStream}\, \mathbb{N}\, s \\
&\mathsf{head}\, (\mathsf{fib}\, s) = 0 \\
&\mathsf{head}\, (\mathsf{tail}\, (\mathsf{fib}\, s)\, r) = 1 \\
&\mathsf{tail}\, (\mathsf{tail}\, (\mathsf{fib}\, s)\, r)\, t = \mathsf{zipWith}\, t\, (+)\, (\mathsf{fib}\, t)\, (\mathsf{tail}\, (\mathsf{fib}\, r)\, t)
\end{aligned}$$

This will accepted by the productivity checker of Agda, whereas omitting the size annotations will cause Agda to reject the definition. This is because the definition is not guarded, and its productivity is dependent on the productivity of zipWith. The typing of zipWith tells us that it is size-preserving, allowing us to reuse it in the definition of fib. Note that the typing of fib tells us that the result may be inspected up to *any* depth, analogously to a regular unsized stream.

Thus, sized types allow us to define functions with more complex patterns of (co)recursion with only minor additional annotations. It also gives a more modular method of termination checking, as the typing of functions contains information on how they affect the size of their arguments, which works especially well with higher-order programs. More involved practical applications of sized types have been explored, including [Dan17; FPS20].

1.2.2 Limitations of Sized Types

Although sized types have proven to be useful both when working with complex recursive functions and when working with coinductive types, several limitations exist in both the theory and implementation of sized types. In fact, the current implementation of sized types in Agda is inconsistent, meaning that we can derive a term of the empty type.

The culprit of this inconsistency is the addition of a closure ordinal ∞ in the type of sizes. Such a closure size is added as a syntactic representation of unbounded usability or a "full" (co)inductive type, such as a list of arbitrary length or a stream of arbitrary depth. This allows the user to work with full (co)inductive types while still using sizes, treating $\mathsf{T}\, \infty$ as the entire (co)inductive type whenever T is a sized type. It allows for typing functions where the size of the result cannot easily be given by the operations for sizes. An example of this is concatenation of sized lists: the concatenation of a sized list bounded by size r with one bounded by s should be bounded by size $r + s$. Since Agda does not provide addition on sizes, this type cannot be given precisely, so instead the return type would be $\mathsf{SList}\, A\, \infty$, i.e. a list with an arbitrary unspecified size.

The issue with the closure size ∞ in Agda is that it does not combine well with the intuition of sizes as bounds on the size or observation depth of a type. In order for it to allow for terms of any size via the constructors we gave for sized lists and sized streams, the rule $\infty < \infty$ needs to be introduced. This contradicts well-foundedness of the ordering on sizes and leading to inconsistencies. Alternative syntactic treatments of the closure size exist, but they are unsatisfactory and depend on the shape of sized type. Thus, we may wonder whether we could do without the closure size. Yet we would still like to reason about regular (co)inductive types using its sized variant, so we require another method of constructing (co)inductive types from sized types.

The intuitive insight is that a term of a full inductive type such as $\mathsf{List}\,A$ has *some* size [Vez15]. Thus, we we might represent the inductive type $\mathsf{List}\,A$, using lists only as an example, instead as $\Sigma_{(s:\mathsf{Size})}\mathsf{SList}\,A\,s$. Dually, the intuition for a term of a coinductive type is that it can be inspected up to *any* depth. Thus, we might want to encode the coinductive type of streams $\mathsf{Stream}\,A$ as $\Pi_{(s:\mathsf{Size})}\mathsf{SStream}\,A\,s$.

Unfortunately, this approach does not exactly capture the right intuition. Firstly, $\Sigma_{(s:\mathsf{Size})}\mathsf{SList}\,A\,s$ is 'larger' than the type $\mathsf{List}\,A$, which we can illustrate as follows. For a size s, we also have the sizes $\uparrow s$ and $\uparrow\uparrow s$. Thus, we may define the term $\mathsf{nil} : \mathsf{SList}\,A\,s$ and so $\mathsf{cons}\,s\,a\,\mathsf{nil} : \mathsf{SList}\,A\,(\uparrow s)$ for some $a : A$, since $s < \uparrow s$. For the same reason, we also get a term $\mathsf{cons}\,s\,a\,\mathsf{nil} : \mathsf{SList}\,A\,(\uparrow s)$. This gives two terms $((\uparrow s), \mathsf{cons}\,s\,a\,\mathsf{nil})$ and $((\uparrow\uparrow s), \mathsf{cons}\,s\,a\,\mathsf{nil})$, both of type $\Sigma_{(s:\mathsf{Size})}\mathsf{SList}\,A\,s$. Semantically, these terms should represent the same list, but they cannot be determined to be definitionally or propositionally equal in Agda. Yet there only exists a unique term $\mathsf{cons}\,a\,\mathsf{nil}$ representing this list of type $\mathsf{List}\,A$, preventing us from defining a bijection between these types.

Dually, the encoding of streams as $\mathsf{Stream}\,A$ as $\Pi_{(s:\mathsf{Size})}\mathsf{SStream}\,A\,s$ is missing a certain coherence condition. The size-indexed streams in the range of such a dependent product should agree on their observations up to the depths on which they are defined, but this encoding does not guarantee this condition. As has been explored in previous work [NVD17], the way to resolve these issues, which we will investigate, is to ensure that sizes are treated parametrically, or uniformly. For inductive types, this will be done by making use of a weak sum type instead of dependent sums. For coinductive types, we will need to ensure that the dependent product type is suitably parametric.

Another challenge arises when working with infinitely-branching inductive types. An example of such a type is that of sized Brouwer ordinals, $\mathsf{Brw}\,s$, with the following constructors, assuming a type of natural numbers:

$$\begin{aligned}\mathsf{Z} &: \mathsf{Brw}\,s\\ \mathsf{S} &: \forall r < s.\ \mathsf{Brw}\,r \to \mathsf{Brw}\,s\\ \mathsf{lim} &: \forall r < s.\ (\mathbb{N} \to \mathsf{Brw}\,r) \to \mathsf{Brw}\,s\end{aligned}$$

With the size algebra in Agda, which has a successor operator and a binary supremum operator, there is no way to construct the first infinite ordinal ω, which would be the limit of the function assigning to each natural number its corresponding finite ordinal. This is because this construction requires a corresponding limit operation on sizes. Thus, the absence of a limit operation on sizes means that we can define terms in the full type of Brouwer ordinals that cannot be defined in the sized variant. In Agda, this would be remedied by using the infinite size ∞, which causes inconsistencies.

1.3 Related Work

In this section we mention an alternative approach to type-based termination which has been developed extensively in recent years and connects to sized types and this work. We also give a brief overview of the notion of parametricity in type theory, as this will be relevant in this work as well.

1.3.1 Guarded Type Theory

Guarded recursion is an alternative approach to recursive equations originally introduced by Nakano [Nak00] in which there exists a guarding modality $\blacktriangleright$, called "later", that can be applied to a type A. The intuition often given in the literature is that a term of type $\blacktriangleright A$ is a value that is only available "one time step" in the future. This modality allows for the construction of *guarded recursive types*, in which the recursive variable only occurs under the guarding modality. For example, the type of guarded streams over A, which is denoted by $\mathsf{Stream}_g\,A$, should satisfy the following:

$$\mathsf{Stream}_g\,A \simeq A \times \blacktriangleright\mathsf{Stream}_g$$

meaning that the head of stream is available "now", whereas the tail is available only after one step of computation. Recursion on the level of terms is done via a guarded fixpoint operator

$$\mathsf{fix} : (\blacktriangleright A \to A) \to A$$

stating that recursive calls should be guarded by the later modality.

However, guarded recursive types do not yet correspond semantically to coinductive types, and the guarding modality by itself is not sufficient to construct coinductive types. Instead, the type system is extended with so-called clock variables [AM13], which represent different time streams and can be used to define coinductive types. Much work has also been on the semantics of guarded type theory, which is often given by contravariant presheaves over the preorder of the natural numbers, also called the topos of trees [Bir+12].

We will not go further into detail on guarded recursion here, but do mention that the connection between sized types and guarded recursion has been studied [VW19], where it was shown that guarded recursion in a simple type theory with only one clock can be simulated using sized types. Moreover, the approach we take to sized types, in which we use quantification over sizes to construct (co)inductive types, is reminiscent of guarded recursion, in which coinductive types are constructed via universal quantification. We highlight some further connections in Chapter 6.

1.3.2 Parametricity in Type Theory

As briefly mentioned, we will introduce several axioms in our type theory expressing how existential and universal quantification over sizes commute with the type formers. One perspective on these axioms is that they internalise certain parametricity properties that are satisfied by the realisability model we construct for the theory. Parametricity has become a somewhat collective term for properties of type theories that express some notion of uniformity or representation-independence.

The typical example of a theory with parametricity is the polymorphic lambda calculus, or System F. In this calculus, we can define only one term of the polymorphic type $\forall X.\ X \to X$, namely the identity. The intuitive idea behind this is that a term of this type must compute a term of type X given a term of type X, for *any* type X. Since it must be able to do so for any type X, it cannot inspect the given term and thus can only return back the input. By a similar intuition, there are exactly two terms of type $\forall X.\ X \to X \to X$.

The notion of parametricity was first formalised by Reynolds [Rey83] for System F using logical relations: parametrically polymorphic functions should preserve all relations. His interpretation of System F validates the consequences of the parametricity property. Since then, his approach has been extended to the more general notion of a reflexive graph model, which have also been used to define models for sized types [Lim19]. There also exists a realisability model for System F, making use of partial equivalence relations, which validates consequences of parametricity [LM91]. Thus, a realisability model will be a good candidate for expressing parametricity properties for quantification over sizes in the theory we define in this work.

However, such parametricity statements are only proved *externally*, not within the theory. To have the consequences of parametricity available in the theory, it needs to be *internalised*. Two main approaches for this have been studied recently, namely the addition of parametric quantifiers to the theory [NVD17], and the addition of axioms expressing the consequences of parametricity. The second approach is similar to the one we take in this work: the added axioms allow us to express the fact that quantification over sizes is parametric, meaning that sizes should be treated uniformly.

1.4 Overview

In this thesis, we define an intensional dependent type theory with a type of sizes and an impredicative universe, and construct a semantics for the type theory and size-indexed types using assemblies and partial equivalence relations. This thesis expands on previous work [NVD17] in which inductive and

coinductive types may be defined by existential quantification and universal quantification respectively over the size-indexed variant of the types. This addresses one of the main issues with the theory of sized types, namely the addition of a closure ordinal in the type of sizes, as we can instead encode unbounded (co)inductive types via quantification over sizes. In order to do so, we require the type of sizes to be treated parametrically by these quantifiers. This parametricity requirement should thus be internalised in the theory, and we do so in the form of additional equivalences, describing how quantification over sizes commutes with other type formers. We work in an intensional setting, but with function extensionality and uniqueness of identity proofs, although it is likely that the last is not strictly required.

In previous work, a parametric model for type theory with sized types was defined using reflexive graphs [Lim19]. Here, we work with a realisability model, as the impredicativity of this model allows us to investigate the consequences of interpreting the types of sizes as a larger set of ordinals. This allows us to show that initial algebras for all polynomial functors over the universe of types can be defined via existential quantification over size-indexed types. Dually, terminal coalgebras for all polynomial functors can be defined via universal quantification over size-indexed types.

The structure for the remainder of this work is as follows. In Chapter 2, we define the theory in which we will work, explaining also the additional axioms. In Chapter 3, we give an overview of the theory of assemblies and partial equivalence relations, and they form a model for the structure of a dependent type theory. In Chapter 4 we interpret the types and terms of our theory in the model. In Chapter 5, we then show how we can construct initial algebras and terminal coalgebras for polynomial functors in the theory. Finally, in Chapter 6 we discuss some connections with guarded type theory and reflexive graphs.

Chapter 2

Dependent Type Theory with Sizes

In this chapter, we describe the type theory that we will be working in, and discuss some of the choices made. The formal syntax with inference rules is described in full at the end of this chapter, in Figures 2.1-2.13.

2.1 Syntax and Rules

We use the symbol $\equiv$ to denote definitional equality in the theory for both types and terms. The theory is formulated using the following forms of judgements:

$$\Gamma\ \mathsf{Ctxt} \quad \Gamma \vdash A\ \mathsf{Type} \quad \Gamma \vdash A \equiv B\ \mathsf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash a \equiv b : A$$

However, we have omitted the standard rules for definitional equality of types, and the rules stating that definitional equality is an equivalence relation and a congruence with respect to all type- and term formers. The rules for context formation, given in Figure 2.1, are standard. We may also assume weakening and substitution rules with respect to all judgements, which are both admissible [Hof97].

2.1.1 Type Formers and Universe

The theory features standard constructions including an empty type, a unit type and booleans, described in Figure 2.3. The main dependent type formers are dependent products and dependent sums. We also include an identity type, which we will use to include axioms expressing internal parametricity of quantification over sizes. We only explicitly write out the induction principles for the dependent sum type, but also assume such principles for the empty-, unit- and boolean types. We make use of both β- and η-rules for Π-types and Σ-types.

For Π-types, described in Figure 2.4, we sometimes omit the typing within a lambda abstraction, writing $\lambda x.b(x)$ instead of $\lambda x{:}A.b(x)$. We will use the notation $A \to B$ for simple function types, i.e. for the dependent product type $\Pi(x : A)B(x)$ where $x : A$ does not occur in the type $B(x)$. We will make use of function composition $g \circ f := \lambda x.\ g(f(x)) : A \to C$, where $f : A \to B$ and $g : B \to C$, as well as an identity map $\mathsf{id}_A := \lambda x.x : A \to A$. The η-rule for Π-types implies that we have definitional equalities for associativity of function composition and that identities are up to definitional equality:

$$f \circ (g \circ h) \equiv (f \circ g) \circ h$$
$$f \circ \mathsf{id} \equiv f \equiv \mathsf{id} \circ f$$

The Σ-type is described in Figure 2.5. Note that the induction principle for the Σ-type allows us to derive the following projections:

$$\frac{\Gamma \vdash c : \Sigma(x : A)B(x)}{\Gamma \vdash \pi_1(c) : A} \qquad \frac{\Gamma \vdash c : \Sigma(x : A)B(x)}{\Gamma \vdash \pi_2(c) : B(\pi_1(c))}$$

such that $\pi_1(\langle x, y\rangle) \equiv x$ and $\pi_2(\langle x, y\rangle) \equiv y$. The rule for η-equality of Σ-types then gives us the definitional equality $c \equiv \langle \pi_1(c), \pi_2(c)\rangle$. Note that there are two presentations of Σ-types: the positive, where the elimination rule is defined by the constructor, and the negative, where the elimination rule is defined by the destructors. We are mixing the positive and negative presentations of Σ-types here, but this is only for convenience, as each can be recovered from the other.

We will also want to make use of binary sums and products. These could be added as primitive types, but to reduce the complexity of the semantics later on we encode the type $A + B$ as $\Sigma(x : \mathsf{Bool})C(x)$ and $A \times B$ as $\Pi(x : \mathsf{Bool})C(x)$, where the type C is defined by $C(\mathsf{tt}) = A$ and $C(\mathsf{ff}) = B$. Note that, unlike most other work, we do not encode the type $A \times B$ as $\Sigma(x : A)B$. The reason for this will become clear later on, and is mostly a technicality to avoid additional work and axioms.

We also include an impredicative Tarski-style universe U, in Figure 2.2, with an explicit decoder El, that is closed under the base types and the dependent type constructors, as described in Figure 2.11. The associated definitional type equalities for the decoder El are also given in Figure 2.12. We will often call the types in the universe U *small types*. A consequence of the η-rule for Π-types is that we may identify a family of small types $x : A \vdash B(x) : U$ indexed by the type A with a function $B : A \to U$. We will make extensive use of this in later sections.

2.1.2 Identity Type and Function Extensionality

The type theory includes a standard identity type, given in Figure 2.6, for expressing propositional equality with elimination and β-equality rules. Given a type A and terms $a_1, a_2 : A$, we will say that a_1 and a_2 are propositionally equal if the type $a_1 =_A a_2$ is inhabited. With only the constructor for reflexivity, terms witnessing the symmetry and transitivity of propositional equality can be defined in the syntax using the elimination rule, see for example [Uni13].

We highlight an important property of the identity type, namely the *action on paths* for a map $f : A \to B$, which we define as the following type:

$$\mathsf{ap}_f : \Pi(x, y : A).\ (x =_A y) \to (f(x) =_B f(y))$$

To define ap_f, by the induction principle for the identity type it suffices to define it for the case of $\mathsf{refl}\, x$:

$$\mathsf{ap}_f(x, x, \mathsf{refl}\, x) := \mathsf{refl}\, f(x)$$

The terminology of paths for identity types comes from homotopy type theory. Another useful application of the identity type is that it gives us a *transport* operation between propositionally equal small types:

$$\mathsf{tr} : \Pi(A, B : U).\ (A =_U B) \to \mathrm{El}\, A \to \mathrm{El}\, B$$

Again, by the induction principle for the identity type it suffices to define it for the case of $\mathsf{refl}\, x$:

$$\mathsf{tr}(A, A, \mathsf{refl}_A) := \mathsf{id}_{\mathrm{El}\, A}$$

We also assume an axiom for function extensionality, stating that for two dependent products $f, g : \Pi(x : A)B(x)$, a proof of their propositional equality is derived from a term witnessing that they are pointwise propositionally equal. This relatively common axiom will be true in the model we construct, and will greatly simplify the encoding of (co)inductive types.

2.1.3 Type of Sizes

The type Size, given in Figure 2.7, has a zero element and a successor operation. It also comes with a preorder relation $\leq$ on the type of sizes, given in Figure 2.8. We will encode this preorder relation as a *proposition*, meaning that, given any two terms $p, q : s_1 \leq s_2$, we have a propositional equality $p = q$.

This will be necessary in order not to introduce any additional structure to the theory when making use of the ordering on sizes, and will hold in the model. We will simply write $\Gamma \vdash s_1 \leq s_2$ when there exists a (unique) term $\Gamma \vdash p : s_1 \leq s_2$.

We let $s_1 < s_2$ Type be notation for $\uparrow s_1 \leq s_2$ Type. We will shortly introduce existential quantification over sizes in the form of a weak sum type. Similarly, we will use the dependent product to quantify over sizes as well. Additional axioms will ensure that dependent products over sizes are suitably parametric, so we will use the notation $\forall i.A(i)$, similar to the polymorphic lambda calculus, to mean either $\pi(i : \mathsf{Size})A(i)$ when $A(i) : U$ and $\Pi(i : \mathsf{Size})A(i)$ when we have A Type.

An additional important notion will be bounded quantification over sizes, which we encode as follows. We will use the notation $i : \mathsf{Size} \vdash \forall j < i.A(j)$ to denote the type $i : \mathsf{Size} \vdash \forall j.(j < i) \to A(j)$. Similarly $i : \mathsf{Size} \vdash \exists j < i.A(i)$ will denote the type $i : \mathsf{Size} \vdash \exists j.((j < i) \times A(j))$. When defining functions in the type theory, we will often omit the inequality proofs, writing the term $\langle j, \langle p, a\rangle\rangle : \exists j < i.A(j)$ as $\langle j, a\rangle$ instead. Similarly, if we have a term $f : \forall j < i.\ A(j)$, we will simply write $f\, j$ to mean $f\, j\, p$ where $p : j < i$. In lambda notation, we may also write $\lambda j < i$ when constructing a term of typing $\forall j < i.\ A(j)$.

The type Size also comes with an elimination principle in the form of a fixpoint operator, corresponding to well-founded induction on sizes. The β-rule for fix determines it to be such that for any $i : \mathsf{Size}$ and $f : \forall i.(\forall j < i.A(j) \to A(i))$ the following propositional equality holds:

$$\mathsf{fix}\, f\, i = f\, i\, (\lambda j < i.\ \mathsf{fix}\, f\, j) : A(i)$$

We may view this as the unfolding rule of the fixpoint operator. The model we construct later will in fact support this as a definitional equality, but this would lead to a loss of decidable type checking, if allowed to unfold indefinitely. Thus, we opt for propositional equality instead. We discuss how a related issue is handled in the context of guarded recursion in Chapter 6

Importantly, we can prove in the theory that, up to propositional equality, $\mathsf{fix}\, f$ is the *unique* term satisfying this unfolding, which we may view as its η-rule:

Proposition 2.1 (Uniqueness of fixpoints)**.** *Suppose we have a term*

$$\Gamma \vdash p : \Pi(i : \mathsf{Size}).\ g\, i = f\, i\, (\lambda j < i.\ g\, j)$$

witnessing for some $g : \forall i.(\forall j < i.A(j) \to A(i))$ that it behave likes the fixpoint of f. Then there exists a term

$$\Gamma \vdash \mathsf{fix}^{\eta}\, p : \Pi(i : \mathsf{Size}).\ g\, i = \mathsf{fix}\, f\, i$$

witnessing pointwise propositional equality of g and $\mathsf{fix}\, f$. In particular, by function extensionality this means that $g = \mathsf{fix}\, f$.

Proof. We can show this by making use of the fixpoint operator itself. We consider the type $\Gamma, i : \mathsf{Size} \vdash g\, i = \mathsf{fix}\, f\, i$ Type. Using the fixpoint operator, to construct a term of type $\Pi(i : \mathsf{Size}).\ g\, i = \mathsf{fix}\, f\, i$, it suffices to give a term

$$q : \forall i.\forall j < i.(g\, j = \mathsf{fix}\, f\, j) \to (g\, i = \mathsf{fix}\, f\, i)$$

So, suppose that we have an $i : \mathsf{Size}$ and $q' : \forall j < i.(g\, j = \mathsf{fix}\, f\, j)$. Then we have the term $p\, i : g\, i = f\, i\, (\lambda j < i.\ g\, j)$, and the term $q'\, j : g\, j = \mathsf{fix}\, f\, j$ allows us to show that $g\, i = f\, i\, (\lambda j < i.\ g\, j) = f\, i\, (\lambda j < i.\ \mathsf{fix}\, f\, j) = \mathsf{fix}\, f\, i$, allowing us to construct q. Thus, using the fixpoint operator, we obtain a term

$$\Gamma \vdash \mathsf{fix}\, q : \Pi(i : \mathsf{Size}).\ g\, i = \mathsf{fix}\, f\, i$$

and by function extensionality we get $g = \mathsf{fix}\, f$. □

Moreover, the term $\mathsf{fix}^{\eta}\, p : \Pi(i : \mathsf{Size}).\ g\, i = \mathsf{fix}\, f\, i$ is itself also unique up to propositional equality, by the same argument as in the previous proposition. Thus, we can say that $\mathsf{fix}\, f$ is the unique term satisfying its unfolding, up to *unique* propositional equality.

2.1.4 Existential Type

We also include a small type for existential quantification over sizes, often called a weak sum type. It is similar to the dependent sum type, but the formation rule, given in Figure 2.10, tells us that we may only quantify over small types $A(i) : U$, and that the resulting existentially quantified type $\exists i.A(i) : U$ is also small. Moreover, the elimination rule for the existential type is 'scoped', as it only allows us to eliminate to small types. A consequence of this restriction is that, unlike the dependent sum type, we cannot define a projection to the first component, since Size does not itself live in U. Thus, we may view the existential type as a kind of dependent sum type where the first component is not allowed to 'escape', thus keeping the size given to the second component abstract.

2.2 Type Equivalences

Besides the syntax and rules given in the previous section, the theory will also include several type equivalences for small types as additional axioms. These equivalences describe how universal- and existential quantification over sizes commute with the type constructors for small types. The equivalences will be justified by the model constructed in Chapter 3, and will be important for encoding (co)inductive types in the calculus.

We thus require a definition of equivalence of types, especially small types, which has been given for intensional type theory and in particular homotopy type theory [Uni13]. Intuitively, we might say that a map $f : A \to B$ is an equivalence if it has an inverse $g : B \to A$. However, encoding this in the theory would result in a type which is not a *mere proposition*, in the terminology of homotopy type theory [Uni13]. The type of equivalences being a mere proposition is desirable, as it means that up to propositional equality, there is a unique $f : A \to B$ that forms an equivalence.

Thus, we add an additional condition to the definition of equivalence in order to satisfy this requirement. This leads us to define type equivalence in the following way.

Definition 2.2 (Equivalence)**.** *For types A and B, a map $f : A \to B$ is a* (half adjoint) equivalence *if there exist $g : B \to A$ and proofs $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$ such that we have a term τ witnessing the following coherence condition:*

$$\tau : \Pi(x : \mathrm{El}\, A).\ \mathsf{ap}_f(\eta\, x) = \epsilon(f x)$$

This gives rise to the type $\mathsf{isequiv}(f)$, which is a mere proposition. We say that small types A and B are equivalent, denoted $A \simeq B$, if there exists a map $f : A \to B$ that is an equivalence. Thus, we define

$$A \simeq B := \Sigma(f : A \to B).\mathsf{isequiv}(f)$$

In practice, we will not use the coherence condition, as it is only there to ensure that $\mathsf{isequiv}(f)$ is a mere proposition. We will mostly be interested in the fact that the type $\mathsf{isequiv}(f)$ being inhabited gives us a unique $g : \mathrm{El}\, B \to \mathrm{El}\, A$ together with proofs that $g \circ f \sim \mathsf{id}_{\mathrm{El}\, A}$ and $f \circ g \sim \mathsf{id}_{\mathrm{El}\, B}$. By function extensionality, this is equivalent to saying that $g \circ f = \mathsf{id}_{\mathrm{El}\, A}$ and $f \circ g = \mathsf{id}_{\mathrm{El}\, B}$. If these two conditions hold, we say that g is a *quasi-inverse* of f. Note that f being a half adjoint equivalence via (g, η, ϵ) implies that g is a quasi-inverse of f. It can be shown that if g is a quasi-inverse of f, then f is a half adjoint equivalence [Uni13], making use of g. Moreover, then g itself is also a half adjoint equivalence via f.

Before introducing the additional axioms for the calculus, we consider three equivalences which can be constructed in the calculus directly.

Remark. For small types $A, B : U$, we will say that they are equivalent if $\mathrm{El}\, A \simeq \mathrm{El}\, B$, and denote this by $A \simeq B$.

Lemma 2.3 (Definable equivalences)**.** *The following equivalences can be constructed in the calculus:*

$$\forall i.\Pi^U(x : A)B(i, x) \simeq \Pi^U(x : A)(\forall i.B(x, i)) \qquad i \notin fv(A)$$

$$\exists i.(\Sigma^U(x : A).B(i, x)) \simeq \Sigma^U(x : A).(\exists i.B(x, i)) \qquad i \notin fv(A)$$

Proof. It suffices to define maps in both directions and show that these are quasi-inverses, as this implies that we have a half-adjoint equivalence as well. For the first equivalence, we define the following maps

$$
\begin{aligned}
&f : \forall i.\Pi(x : A)B(i,x) \to \Pi(x : A)(\forall i.B(x,i))\\
&f\,p := \lambda x.\lambda i.\ p\,i\,x\\
&g : \Pi(x : A)(\forall i.B(x,i)) \to \forall i.\Pi(x : A)B(i,x)\\
&g\,q := \lambda i.\lambda x.\ q\,x\,i
\end{aligned}
$$

It is not difficult to see that $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$, which thus allows us to construct an equivalence. For the second equivalence, we define the maps:

$$
\begin{aligned}
&f : \exists i.(\Sigma(x : A).B(i,x)) \to \Sigma(x : A).(\exists i.B(x,i))\\
&f\,\langle i,(a,b)\rangle := (a,\langle i,b\rangle)\\
&g : \Sigma(x : A).(\exists i.B(x,i)) \to \exists i.(\Sigma(x : A).B(i,x))\\
&g\,(a,\langle i,b\rangle) := \langle i,(a,b)\rangle
\end{aligned}
$$

The required equalities to show that $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$ follow from the η-rule for Σ-types. □

The equivalences above show how the quantifiers commute with some of the type constructors. For the remaining type constructors, the required equivalences are not constructible in the calculus directly. Instead, the equivalences described in Figure 2.13 are added as axioms. More precisely, for each equivalence listed, a canonical map f can be defined from left to right. The additional axioms thus state that for each such f, the type $\mathsf{isequiv(f)}$ is inhabited. Note that this gives us a unique map g going from right to left which is thus also a half adjoint equivalence. We give the definitions of the maps from left to right below.

Proposition 2.4 (Definable maps for the axioms)**.** *For each equivalence listed in Figure 2.13, there exists a canonical map from left to right.*

Proof. We simply list the canonical maps. The maps for universal quantification are as follows:

$$
\begin{aligned}
&\lambda a.\lambda i.\ a : A \to \forall i.A\\
&\lambda g.\lambda i.\lambda j.\ g\,j : \forall i.A(i) \to \forall i.\forall j < i.A(j)\\
&\lambda\langle x,g\rangle.\lambda i.\ \langle x,g\,i\rangle : \Sigma^U(x : A)(\forall i.B(x,i)) \to \forall i.(\Sigma^U(x : A)B(i,x))
\end{aligned}
$$

The maps for existential quantification are as follows:

$$
\begin{aligned}
&\lambda\langle i,a\rangle.\ a : \exists i.A \to A\\
&\lambda\langle i,j,a\rangle.\ \langle j,a\rangle : \exists i.\exists j < i.A(j) \to \exists i.A(i)\\
&\lambda\langle i,g\rangle.\lambda a.\ g\,a : \exists i.(\Pi^U(x : \mathrm{El}\,A).\ \exists j < i.B(x,j)) \to \Pi^U(x : \mathrm{El}\,A)(\exists i.B(i,x))
\end{aligned}
$$

□

Note that the following equivalence

$$\exists i.(\Pi^U(x : \mathrm{El}\,A).\ \exists j < i.B(x,j)) \simeq \Pi^U(x : \mathrm{El}\,A)(\exists i.B(i,x))$$

differs in the fact that the existential quantifier does not exactly commute with the small Π-type; we will call this *weakly commuting* later on. The intuitive reason is that in both cases, to go from right to left, there must be a *limit* size i of all sizes in the range of the (dependent) product. However, we cannot guarantee that the type B is monotone, or covariant, in Size, so we make use of $\exists j < i.B(x,j)$ instead, which is monotone in Size. This axiom thus tells us that we have a way of constructing limits of sequences of sizes in the theory, without explicitly adding a limit operator on sizes to the syntax.

Note also that our chosen encodings for binary sums and binary products tell us that they commute with the universal quantifier, and (weakly) commute with the existential quantifier.

Finally, we note the similarity between our axiom $A \simeq \forall i.A$ when $i \notin \mathrm{fv}(A)$, which we may view as a size-irrelevance principle, and the clock-irrelevance principle used in guarded type theory. The axiom $\exists i.A \simeq A$ when $i \notin \mathrm{fv}(A)$ is its dual for existential quantification, but this does not have an analogue in guarded type theory, which exclusively uses universal quantification over clocks.

$$\frac{}{\vdash \diamond\ \mathsf{Ctxt}} \qquad \frac{\vdash \Gamma\ \mathsf{Ctxt} \qquad \Gamma \vdash A\ \mathsf{Type}}{\vdash \Gamma, x : A\ \mathsf{Ctxt}}$$

$$\frac{\vdash \Gamma, x : A, \Delta\ \mathsf{Ctxt}}{\Gamma, x : A, \Delta \vdash x : A}$$

Figure 2.1: Rules for context formation

$$\frac{}{\Gamma \vdash U\ \mathsf{Type}} \qquad \frac{\Gamma \vdash A : U}{\Gamma \vdash \mathrm{El}\, A\ \mathsf{Type}}$$

Figure 2.2: Rules for universe type

$$\frac{}{\Gamma \vdash \top\ \mathsf{Type}} \qquad \frac{}{\Gamma \vdash \bot\ \mathsf{Type}} \qquad \frac{}{\Gamma \vdash \mathsf{Bool}\ \mathsf{Type}}$$

$$\frac{}{\Gamma \vdash \star : \top} \qquad \frac{}{\Gamma \vdash \mathsf{tt} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{ff} : \mathsf{Bool}}$$

Figure 2.3: Rules for empty-, unit-, and boolean type (Their elimination principles are left out)

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma, x : A \vdash B(x)\ \mathsf{Type}}{\Gamma \vdash \Pi(x : A)B(x)\ \mathsf{Type}}$$

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x{:}A.\ b(x) : \Pi(x : A)B(x)} \qquad \frac{\Gamma \vdash f : \Pi(x : A)B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B(a)}$$

$$\frac{\Gamma, x : A \vdash b : B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x{:}A.\ b(x))\ a \equiv b(a) : B[a/x]} \qquad \frac{\Gamma \vdash f : \Pi(x : A)B(x)}{\Gamma \vdash (\lambda x{:}A.\ f\ x) \equiv f : \Pi(x : A)B(x)}$$

Figure 2.4: Rules for Π-types with β- and η-rules

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma, x : A \vdash B(x)\ \mathsf{Type}}{\Gamma \vdash \Sigma(x : A)B(x)\ \mathsf{Type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \Sigma(x : A)B(x)}$$

$$\frac{\begin{array}{c}\Gamma, z : \Sigma(x : A)B(x) \vdash P(z)\ \mathsf{Type} \\ \Gamma, x : A, y : B \vdash p : P((x, y))\end{array}}{\Gamma, z : \Sigma(x : A)B(x) \vdash \mathsf{Ind}_\Sigma(z, p) : P(z)}$$

$$\frac{\begin{array}{c}\Gamma, z : \Sigma(x : A)B(x) \vdash P(z)\ \mathsf{Type} \\ \Gamma, x : A, y : B \vdash p : P((x, y))\end{array}}{\Gamma, x : A, y : B \vdash \mathsf{Ind}_\Sigma((x, y), p) \equiv p : P((x, y))}$$

$$\frac{\Gamma \vdash c : \Sigma(x : A)B(x)}{\Gamma \vdash c \equiv (\pi_1(c), \pi_2(c))}$$

Figure 2.5: Rules for Σ-types with β- and η-rules (note that π_1 and π_2 are the derived projections)

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma \vdash a_1, a_2 : A}{\Gamma \vdash a_1 =_A a_2\ \mathsf{Type}} \qquad \frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\, a : a =_A a}$$

$$\frac{\begin{array}{c}\Gamma, x : A, y : A, z : x =_A y \vdash P(x, y, z)\ \mathsf{Type} \\ \Gamma, x : A \vdash p(x) : P(x, x, \mathsf{refl}\, x)\end{array}}{\Gamma, x : A, y : A, z : x =_A y \vdash \mathsf{Ind}_=(x, y, z, p) : P(x, y, z)}$$

$$\frac{\begin{array}{c}\Gamma, x : A, y : A, z : x =_A y \vdash P(x, y, z)\ \mathsf{Type} \\ \Gamma, x : A \vdash p(x) : P(x, x, \mathsf{refl}\, x)\end{array}}{\Gamma, x : A \vdash \mathsf{Ind}_=(x, x, \mathsf{refl}\, x, p) \equiv p(x) : P(x, x, \mathsf{refl}\, x)}$$

$$\frac{\Gamma \vdash f, g : \Pi(x : A)B(x) \qquad \Gamma, x : A \vdash e_x : f(x) =_{B(x)} g(x)}{\Gamma \vdash \mathsf{funext}(f, g, e) : f =_{\Pi(x:A)B(x)} g}$$

Figure 2.6: Rules for Propositional Equality and Function Extensionality

$$\frac{}{\Gamma \vdash \mathsf{Size}\ \mathsf{Type}} \qquad \frac{\Gamma \vdash s_1, s_2 : \mathsf{Size}}{\Gamma \vdash\ s_1 \leq s_2\ \mathsf{Type}}$$

$$\frac{}{\Gamma \vdash 0 : \mathsf{Size}} \qquad \frac{\Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash \uparrow s : \mathsf{Size}} \qquad \frac{\Gamma \vdash s_1, s_2 : \mathsf{Size}}{\Gamma \vdash s_1 \sqcup s_2 : \mathsf{Size}}$$

Figure 2.7: Type of Sizes

$$\frac{\Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash 0 \leq s} \qquad \frac{\Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash s \leq \uparrow s} \qquad \frac{\Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash s \leq s}$$

$$\frac{\Gamma \vdash s_1 \leq s_2 \qquad \Gamma \vdash s_2 \leq s_3}{\Gamma \vdash s_1 \leq s_3}$$

Figure 2.8: Ordering of Sizes

$$\frac{\Gamma, i : \mathsf{Size} \vdash A(i)\ \mathsf{Type} \qquad \Gamma \vdash f : \forall i.((\forall j < i.A(j)) \to A(i))}{\Gamma \vdash \mathsf{fix}\, f : \forall i.A(i)}$$

$$\frac{}{\Gamma \vdash \mathsf{fix}^{\beta}\, f : \Pi(i : \mathsf{Size}).\ \mathsf{fix}\, f\, i = f\, i\, (\lambda j < i.\ \mathsf{fix}\, f\, j)}$$

Figure 2.9: Fixpoint operator with a propositional β-rule

$$\frac{\Gamma, i : \mathsf{Size} \vdash A(i) : U}{\Gamma \vdash \exists i.A(i) : U}$$

$$\frac{\Gamma, i : \mathsf{Size} \vdash A(i) : U \qquad \Gamma \vdash s : \mathsf{Size} \qquad \Gamma \vdash a : \mathrm{El}\, A(i)}{\Gamma \vdash \langle s, a \rangle : \mathrm{El}\, (\exists i.A(i))}$$

$$\frac{\begin{array}{c}\Gamma, z : \mathrm{El}\, (\exists i.A(i)) \vdash P(z) : U \\ \Gamma, i : \mathsf{Size}, x : \mathrm{El}\, A \vdash p : \mathrm{El}\, P(\langle i, x \rangle)\end{array}}{\Gamma, x : \exists i.A(i) \vdash \mathsf{Ind}_{\exists}(x, p) : \mathrm{El}\, P}$$

$$\frac{\begin{array}{c}\Gamma, z : \mathrm{El}\, (\exists i.A(i)) \vdash P(z) : U \\ \Gamma, i : \mathsf{Size}, x : \mathrm{El}\, A \vdash p : \mathrm{El}\, P(\langle i, x \rangle)\end{array}}{\Gamma, i : \mathsf{Size}, x : \mathrm{El}\, A \vdash \mathsf{Ind}_{\exists}(\langle i, x \rangle, p) \equiv p : \mathrm{El}\, P(\langle i, x \rangle)}$$

Figure 2.10: Existential Type

$$\frac{}{\Gamma \vdash \top^U : U} \qquad \frac{}{\Gamma \vdash \bot^U : U} \qquad \frac{}{\Gamma \vdash \mathsf{Bool}^U : U}$$

$$\frac{\Gamma \vdash A\ \mathsf{Type} \qquad \Gamma, x : A \vdash B(x) : U}{\Gamma \vdash \Pi^U (x : A) B(x) : U}$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : \mathrm{El}\, A \vdash B(x) : U}{\Gamma \vdash \Sigma^U (x : A) B(x) : U}$$

$$\frac{\Gamma \vdash s_1, s_2 : \mathsf{Size}}{\Gamma \vdash s_1 \leq^U s_2 : U}$$

Figure 2.11: Codes for Small Types

$$
\begin{aligned}
\mathrm{El}\,(\bot^U) &\equiv \bot \\
\mathrm{El}\,(\top^U) &\equiv \top \\
\mathrm{El}\,(\mathsf{Bool}^U) &\equiv \mathsf{Bool} \\
\mathrm{El}\,(\Pi^U(x:A)B(x)) &\equiv \Pi(x:A)(\mathrm{El}\,B(x)) \\
\mathrm{El}\,(\Sigma^U(x:A)B(x)) &\equiv \Sigma(x:\mathrm{El}\,A)(\mathrm{El}\,B(x)) \\
\mathrm{El}\,(s_1 \leq^U s_2) &\equiv s_1 \leq s_2
\end{aligned}
$$

Figure 2.12: Type Equalities for the universe

$$
\begin{aligned}
A &\simeq \forall i.\ A && i \notin \mathrm{fv}(A) \\
\forall i.\ A(i) &\simeq \forall i.\ \forall j < i.\ A(j) && \\
\Sigma^U(x:A)(\forall i.B(x,i)) &\simeq \forall i.(\Sigma^U(x:A)B(i,x)) && i \notin \mathrm{fv}(A) \\
\\
\exists i.\ A &\simeq A && i \notin \mathrm{fv}(A) \\
\exists i.\ \exists j < i.\ A(j) &\simeq \exists i.\ A(i) && \\
\exists i.(\Pi^U(x:\mathrm{El}\,A).\ \exists j < i.B(x,j)) &\simeq \Pi^U(x:\mathrm{El}\,A)(\exists i.B(x,i)) && i \notin \mathrm{fv}(A)
\end{aligned}
$$

Figure 2.13: Type equivalences for existential and universal quantification. Note that for each equivalence, the left-to-right map f is definable. The additional axiom states that $\mathsf{isequiv}(\mathrm{f})$ is inhabited.

Chapter 3

Realisability and Categories with Families

This chapter introduces assemblies and partial equivalence relations, and shows how they can be given the structure of a category with families. In a realisability model for type theory, types are interpreted as sets together with a realisability relation between the set and a set of *realisers*. The intuition given by Reus [Reu99] is that realisers can be seen as "codes" in a low-level language. A code that is related to an object in the set is said to realise it, and can be seen as a representation of that object.

3.1 Partial Combinatory Algebras

The definitions of partial equivalence relations and assemblies are parametrised by the notion of a partial combinatory algebra, which we may view as an abstract model of computation. We will introduce only the necessary characteristics of a partial combinatory algebra, before restricting ourselves to a particular instance.

Definition 3.1 (Partial combinatory algebra)**.** *A* partial combinatory algebra *(pca) is a tuple* $(\mathcal{A}, \cdot)$ *consisting of a set* $\mathcal{A}$ *and a partial operation* $\cdot : \mathcal{A} \times \mathcal{A} \rightharpoonup \mathcal{A}$*, such that there exist elements* $\mathtt{k}, \mathtt{s} \in \mathcal{A}$ *satisfying the following for all* $a, b, c \in \mathcal{A}$*, writing* $a \cdot b \downarrow$ *when* $a \cdot b$ *is defined:*

1. $\mathtt{k} \cdot a \cdot b \downarrow$ *and* $\mathtt{k} \cdot a \cdot b = a$*;*
2. $\mathtt{s} \cdot a \cdot b \downarrow$
3. $\mathtt{s} \cdot a \cdot b \cdot c \simeq a \cdot c \cdot (b \cdot c)$

Here, $\simeq$ *denotes Kleene equality, meaning that either both sides are undefined, or both sides are defined and they are equal. Moreover, when omitting parentheses, we assume that the partial operation associates to the left, so* $\mathtt{k} \cdot a \cdot b$ *means* $(\mathtt{k} \cdot a) \cdot b$*.*

This is a rather abstract definition, and it will not be very insightful for our use case. Instead, we give a different characterisation that will prove more helpful. For this, we need the notion of a term over a pca.

Definition 3.2 (Terms)**.** *The set of* terms *over a pca* $(\mathcal{A}, \cdot)$ *is built up from countably many variables and elements of* $\mathcal{A}$ *via the partial binary operation. A term* t *is* closed *if it contains no variables.*

We can extend the notion of being defined to closed terms, by saying that a closed term t is defined when all applications $a \cdot b$ which are subterms of t are defined. This gives us a notion of Kleene equality $t_1 \simeq t_2$ between terms. The important point for this work is that a pca admits a notion of abstraction, similar to λ-abstraction.

Proposition 3.3. *$(\mathcal{A}, \cdot)$ is a partial combinatory algebra iff for every polynomial t over $\mathcal{A}$ containing only the variables $x, x_1, \ldots, x_n$ and for every variable x, there exists a polynomial $\Lambda x.t$ over $\mathcal{A}$ containing only the variables $x_1, \ldots, x_n$ such that*

$$([\Lambda x.t][a_1/x_1, \ldots, a_n/x_n]) \downarrow$$

and

$$([\Lambda x.t][a_1/x_1, \ldots, a_n/x_n]) \cdot a \simeq t[a/x, a_1/x_1, \ldots, a_n/x_n]$$

for any $a, a_1, \ldots, a_n \in \mathcal{A}$.

The proof of this characterisation and a general overview of partial combinatory algebras can be found in [Oos08]. In particular, this means that terms may be represented as elements of a pca by using abstraction, which we will use extensively when constructing the model.

In this work we will restrict ourselves to a specific pca, namely the first Kleene algebra $\mathcal{K}_1$, consisting of the set of natural numbers $\mathbb{N}$ together with an enumeration of the partial recursive functions $\{\varphi_n\}_{n\in\mathbb{N}}$. We can define a partial operation as $m \cdot n := \varphi_m(n)$ and thus write $m \cdot n \downarrow$ when $\varphi_m(n)$ halts. Note that this operation is not associative. Often, we will call the elements of $\mathbb{N}$ in this context *codes* or *realisers*, viewing a natural number n as a code for the partial recursive function φ_n. Moreover, we assume that a computable bijective function $\langle \_, \_ \rangle : \mathbb{N}^2 \to \mathbb{N}$ is given, together with projections $\mathsf{pr}_1, \mathsf{pr}_2 : \mathbb{N} \to \mathbb{N}$. Such an enumeration of partial recursive functions and a pairing function can be found in many treatments of recursion theory [Odi89]. Proper definitions of the combinators `k` and `s` can be given using the s-m-n theorem.

3.2 Partial Equivalence Relations and Assemblies

With the required definitions of partial combinatory algebra, we are ready to define partial equivalence relations and assemblies over the pca $\mathcal{K}_1$.

Definition 3.4 (Partial equivalence relations)**.** *A* partial equivalence relation *(PER) $R \subseteq \mathbb{N} \times \mathbb{N}$ is a symmetric and transitive relation on the natural numbers. We define the following notations:*

$$\begin{aligned} dom(R) &:= \{n \in \mathbb{N} \mid nRn\} \\ [n]_R &:= \{m \in \mathbb{N} \mid nRm\} \\ \mathbb{N}/R &:= \{[n]_R \mid n \in dom(R)\} \end{aligned}$$

Note that R is an equivalence relation when restricted to $dom(R)$. Given PERs R and S, a morphism of PERs from R to S is a function $f : \mathbb{N}/R \to \mathbb{N}/S$ such that there exists some $e \in \mathbb{N}$ with the property that for every $[n]_R \in \mathbb{N}/R$, $f([n]_R) = [e \cdot n]_S$. We say that f is tracked *by e. We denote the category of PERs by* PER.

Note that in general, the computation $e \cdot n = \phi_e(n)$ need not be defined, since e is a *partial* recursive function. Thus, when writing $[e \cdot n]_S$, we implicitly assume $(e \cdot n) \downarrow$.

Example 3.5. The initial object in PER is the empty relation $\mathbf{0} := \varnothing$, since there exists a unique empty function $f : \mathbb{N}/\mathbf{0} \to \mathbb{N}/R$ to each PER R, vacuously tracked by every realiser.

The terminal object in PER is the total relation $\mathbf{1} := \mathbb{N} \times \mathbb{N}$, since there exists a unique function $f : \mathbb{N}/R \to \mathbb{N}/\mathbf{1}$ from each PER R sending everything to the unique element in $\mathbb{N}/\mathbf{1}$. To see that this is a morphism, note that it is tracked by every realiser n such that φ_n is total.

An intuition for PERs is that the natural numbers are codes for certain objects, and two natural numbers are related if they code the same object. Not every natural number must code an object, so the relation need not be reflexive.

To interpret the large types in the theory, we make use of assemblies, also called ω-sets when working with the natural numbers as the set of realisers.

Definition 3.6 (Assemblies)**.** *An* assembly *is a pair $(A, \Vdash_A)$ consisting of a set A and a realisability relation $\Vdash_A \subseteq \mathbb{N} \times A$ such that for every $a \in A$, there exists an $n \in \mathbb{N}$ such that $n \Vdash_A a$. We say that n* realises *a, or that n is a* code *for a. We will often use A to denote both the assembly and the underlying set.*

A morphism of assemblies $(A, \Vdash_A) \to (B, \Vdash_B)$ is a function $f : A \to B$ between the underlying sets such that there exists some $e \in \mathbb{N}$ satisfying the following:

$$\forall n \in \mathbb{N}, \forall a \in A.\ n \Vdash_A a \implies e \cdot n \Vdash_B f(a)$$

In this case, we say that f is tracked *by e. Note that the realiser tracking f is not part of the data of the morphism. We denote the category of assemblies by* Asm*. Here again, the notation $e \cdot n \Vdash_A f(a)$ implicitly assumes $(e \cdot n)\downarrow$.*

Example 3.7. As with PERs, the initial object in Asm is the empty set $\varnothing$ with the empty realisability relation, whereas the terminal object is the singleton set $\{*\}$ with the total realisability relation $\mathbb{N} \times \{*\}$. We will also denote these by $\mathbf{0}$ and $\mathbf{1}$ respectively, confusing notation with the initial and terminal objects in PER. The reason for this will become clear momentarily.

Note that any set A can be lifted to an assembly $(A, \mathbb{N} \times A)$ with the full or complete realisability relation, which we will denote as ∇A. Moreover, the operation $\nabla : \mathsf{Set} \to \mathsf{Asm}$ is functorial, since every set-theoretic function $f : A \to B$ is also an assembly morphism $f : \nabla A \to \nabla B$, as is shown by the following proposition.

Proposition 3.8. *For any assembly $(A, \Vdash_A)$ and set X, every set-theoretic function $f : A \to X$ is also an assembly morphism $f : A \to \nabla X$.*

Proof. It suffices to show that f is tracked by some realiser, and in fact f is tracked by every realiser e where φ_e is total. Given such an $e \in \mathbb{N}$ and $a \in A$ where $n \Vdash_A a$, we have $(e \cdot n) \Vdash_{\nabla X} f(a)$ since ∇X carries full realisability structure. □

Note that there is also an obvious forgetful functor from Asm to $\mathbf{Set}$, which is left adjoint to ∇. There is a close connection between PERs and assemblies, as PERs can be given the structure of an assembly: given a PER R, the pair $(\mathbb{N}/R, \in)$ is an assembly. In fact, the category PER is equivalent to an important subcategory of assemblies, namely that of modest sets.

Definition 3.9 (Modest sets)**.** *An assembly $(A, \Vdash_A)$ is* a modest set *if every $n \in \mathbb{N}$ codes at most one $a \in A$: $\forall n \in \mathbb{N}.\ n \Vdash_A a_1, a_2 \implies a_1 = a_2$. We denote the category of modest sets by* Mod*, which is a full subcategory of* Asm*.*

Note that for a PER R, $(\mathbb{N}/R, \in)$ is modest, since if $n \in [m]_R$ and $n \in [m']_R$, then $[m]_R = [m']_R$ since R is an equivalence relation on $\mathrm{dom}(R)$.

Lemma 3.10. *There is an equivalence between the categories* PER *and* Mod*.*

Proof. We can define the inclusion functor $\iota : \mathsf{PER} \to \mathsf{Mod}$ as $\iota(R) = (\mathbb{N}/R, \in)$ and for $f : R \to S$, $\iota(f) = f$. This gives a morphism of assemblies since f is tracked by some e and $m \in [n]_R$ implies $[n]_R = [m]_R$ and thus implies that $e \cdot m \in f([n]_R) = [e \cdot m]_S$.

Conversely, we can define a functor $\mathsf{E} : \mathsf{Mod} \to \mathsf{PER}$ where for a modest set A we have $\mathsf{E}(A) = \{(m, n) \mid \exists a \in A.\ m, n \Vdash_A a\}$ and for $f : A \to B$ we define $\mathsf{E}(f) : \mathsf{E}(A) \to \mathsf{E}(B)$ as $\mathsf{E}(f)([n]_{\mathsf{E}(A)}) = f(a)$, where a is the unique element such that $n \Vdash_A a$, since A is modest.

It is now easy to check that equality holds in one direction, so $\mathsf{E}(\iota(R)) = R$, whereas we get a natural isomorphism in the other direction: $\iota(\mathsf{E}(A)) \cong A$. □

The category PER is thus a subcategory of Mod. In fact, the previous proof shows that every modest set is isomorphic to a PER, when viewing the PERs as assemblies. We can thus view Mod as the isomorphism-closure, or repletion, of PER, and Mod is an isomorphism-closed, or replete, subcategory

of Asm. These relations are what will allow us to interpret our type theory using modest sets and assemblies.

For any modest set A, we may write $\sim_A$ for the PER obtained by $\mathsf{E}(A)$. Thus, for any PER R, we have that $\sim_{\iota(R)} = R$. We will often blur the distinction between a PER R and its corresponding assembly $(\mathbb{N}/R, \in)$, treating $\mathbb{N}/R$ as both a PER and an assembly. Thus, we may write $n \Vdash_{\mathbb{N}/R} [m]_R$, or equivalently nRm.

3.3 Assemblies as a Category with Families

To interpret the type theory using assemblies, we make use of the categories with families (CwF) framework, which gives a general structure for interpreting type theories that follows the syntax quite closely. There are several treatments giving the details of this framework for interpreting type theory [Hof97; CCD21], so we will only cover what is necessary.

We make use of the definitions from Hofmann [Hof97], in which the construction of a CwF begins with the following structure:

- A category $\mathcal{C}$ of semantic contexts and context morphisms;
- For each $\Gamma \in \mathcal{C}$, a collection $\mathsf{Ty}(\Gamma)$ of semantic types in context Γ.
- For each $\Gamma \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Gamma)$, a collection $\mathsf{Tm}(\Gamma, A)$ of semantic terms of type A in context Γ.

We will often leave out "semantic" where this does not cause confusion. For the model in assemblies, the category Asm will be the category of contexts. Given an assembly Γ, a type $A \in \mathsf{Ty}(\Gamma)$ will be a family of assemblies over Γ, so $A : \Gamma \to \mathsf{Asm}$. We will often write such a family as $A = (A_\gamma)_{\gamma \in \Gamma}$. Finally, a term $a \in \mathsf{Tm}(\Gamma, A)$ is a dependent morphism $a : (\gamma \in \Gamma) \to A_\gamma$, often written $a = (a_\gamma)_{\gamma \in \Gamma}$, meaning that a is tracked by some $e \in \mathbb{N}$, i.e.

$$\forall n \in \mathbb{N}.\ n \Vdash_\Gamma \gamma \implies (e \cdot n) \Vdash_{A_\gamma} a_\gamma$$

A CwF requires a notion of *semantic substitution*, given by an operation on types and one on terms. Given a context morphism $f : \Delta \to \Gamma$, there should be an operation $-[f] : \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Delta)$, and for each $A \in \mathsf{Ty}(\Gamma)$ an operation $-[f] : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Delta, A[f])$. Moreover, these operations should be compatible with identities and composition [Hof97]. For assemblies, both operations are defined by precomposition with f. Given an assembly morphism $f : \Delta \to \Gamma$ and a type $A \in \mathsf{Ty}(\Delta)$, we define $A[f]_\delta = A_{f(\delta)}$. Similarly, if $a \in \mathsf{Tm}(\Gamma, A)$, we define $(a[f])_\delta = a_{f(\delta)}$.

A CwF should be able to interpret the *context formation* rules of the theory. The empty context $\diamond$ will be modelled by the terminal object in $\mathcal{C}$. In the category of assemblies, the terminal object is given by $\mathbf{1} = (\{*\}, \mathbb{N} \times \{*\})$. To interpret context extension, we require for each $\Gamma \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Gamma)$ an extended context $\Gamma.A \in \mathcal{C}$, together with context morphisms $\mathbf{p}_A : \Gamma.A \to \Gamma$ and $\mathbf{v}_A \in \mathsf{Tm}(\Gamma.A, A[\mathbf{p}_A])$, corresponding to first and second projections respectively. Context extension is defined as follows for assemblies:

Definition 3.11 (Context extension)**.** *Given an assembly Γ and $A \in \mathsf{Ty}(\Gamma)$, we define the context extension $\Gamma.A$ as:*

$$\Gamma.A = \{(\gamma, a) \mid \gamma \in \Gamma \wedge a \in A_\gamma\}$$
$$\langle n_1, n_2 \rangle \Vdash_{\Gamma.A} (\gamma, a) \text{ iff } n_1 \Vdash_\Gamma \gamma \text{ and } n_2 \Vdash_{A_\gamma} a$$

The morphisms $\mathbf{p}_A : \sigma(\Gamma, A) \to \Gamma$ and $\mathbf{v}_A \in \mathsf{Tm}(\Gamma.A, A[\mathbf{p}_A])$ are given by $(\gamma, a) \mapsto \gamma$ and $(\gamma, a) \mapsto a$ respectively, which are assembly morphisms tracked by the first and second projections on the realisers. It is moreover clear that in this model, $A[\mathbf{p}_A]_{(\gamma,a)} = A_\gamma$.

Finally, a CwF requires an operation to extend a context morphism by a term. That is, given a context morphism $f : \Delta \to \Gamma$, a type $A \in \mathsf{Ty}(\Gamma)$, and a term $a \in \mathsf{Tm}(\Delta, A[f])$, there should be a morphism $\langle f, a\rangle : \Delta \to \Gamma.A$ satisfying several equations, as found in [Hof97]. In the realisability model, this is defined as $\langle f, a\rangle(\delta) = (f_\delta, a_\delta)$. Note that if d tracks f and e tracks M, then $\langle d, e\rangle$ tracks $\langle f, a\rangle$, so this is indeed an assembly morphism. If $a \in \mathsf{Tm}(\Gamma, A)$, then we trivially have that $a \in \mathsf{Tm}(\Gamma, A[\mathsf{id}_\Gamma])$, so we get a context morphism $\langle \mathsf{id}_\Gamma, a\rangle : \Gamma \to \Gamma.A$, where $\langle \mathsf{id}_\Gamma, a\rangle(\gamma) = (\gamma, a_\gamma)$. We will denote this morphism by $\overline{a} : \Gamma \to \Gamma.A$, as it will be useful later.

This completes the construction of a category with families from the category of assemblies, allowing us to interpret the structural rules of our dependent type theory in assemblies. To give an idea of the formal interpretation that we derive from the categories with families structure, we define the formal interpretation of the context rules given in Figure 2.1 inductively as:

$$\begin{aligned}
[\![\vdash \diamond\, \mathsf{Ctxt}]\!] &= \mathbf{1} \\
[\![\vdash \Gamma, x : A\, \mathsf{Ctxt}]\!] &= [\![\vdash \Gamma\, \mathsf{Ctxt}]\!].[\![\Gamma \vdash A\ \mathsf{Type}]\!] \\
[\![\Gamma, x : A \vdash x : A]\!] &= \mathbf{v}_{[\![\Gamma \vdash A\ \mathsf{Type}]\!]} \\
[\![\Gamma, x : A, \Delta, y : B \vdash x : A]\!] &= [\![\Gamma, x : A, \Delta \vdash x : A]\!][\mathbf{p}_{[\![\Gamma, x:A, \Delta \vdash B\ \mathsf{Type}]\!]}]
\end{aligned}$$

Chapter 4

Interpretation of the Type Theory

Now that we have given the category of assemblies the structure of a categories with families, we know that it serves as a model for the structural rules of the dependent type theory. However, we still need to specify how the type and term formers in the theory should be interpreted. We will only give the interpretations of the type and term formers; the specific conditions that these interpretations must satisfy can be found in [Hof97].

4.1 Note on the Model

Our model will ensure that the theory we defined in Chapter 2 is consistent. However, the fact that we construct a realisability model gives us more than just consistency. Although, as will become clear, the interpretation from the theory to the model is not computable, every function in the model itself is tracked by a realiser, in our case a partial recursive function which is defined on all codes realising elements in the domain of the function. Thus, the model gives us an interpretation of the theory in terms of computable functions.

4.2 Type- and Term Formers

The formal interpretation of the theory will largely follow the structure of the categories with families framework. Thus, the interpretation of a type judgment $[\![\Gamma \vdash A\ \mathsf{Type}]\!]$ will be given by the semantic type $A \in \mathsf{Ty}(\Gamma)$. Similarly, the interpretation of a term judgement $[\![\Gamma \vdash a : A]\!]$ is given by the semantic term $a \in \mathsf{Tm}(\Gamma, A)$ where $A \in \mathsf{Ty}(\Gamma)$. In a few cases, the formal interpretation will deviate from this, in which case we will explicitly mention this. When interpreting terms, we will show that they are tracked by some realiser in order to show that the interpretation constitutes a morphism.

4.2.1 Universe Type

We start by giving an interpretation of the universe type U containing codes for small types. This will allow us to give simultaneous definitions for the small and large constructors for the other type constructors afterwards. As mentioned, a code in the universe U should be interpreted by a PER, and the corresponding type will then be the assembly that arises from the inclusion of the PER into the category of assemblies. This suggests interpreting the universe U as the set of all PERs using the construction $\nabla\mathsf{PER}$.

Definition 4.1 (Universe type)**.** *For any context Γ, there is a type $U \in \mathsf{Ty}(\Gamma)$ defined as $U_\gamma = \nabla\mathsf{PER}$, viewing PER as the set of all PERs. Moreover, there exists a type $\mathrm{El} \in \mathsf{Ty}(\Gamma.U)$ defined as $\mathrm{El}_{(\gamma,R)} = \iota(R)$, where $\iota(R)$ is the PER R viewed as an assembly. Given a type $A \in \mathsf{Tm}(\Gamma, U)$ in the universe, we obtain a type $\mathrm{El}\,[\overline{A}] \in \mathsf{Ty}(\Gamma)$, which we will denote by $\mathrm{El}\,A$, following the definitions in [AGJ14].*

Here, the formal interpretation would be $[\![\Gamma \vdash U\ \mathsf{Type}]\!]\gamma = U_\gamma = \nabla\mathsf{PER}$. Then given an interpretation $[\![\Gamma \vdash A : U]\!] = A \in \mathsf{Tm}(\Gamma, U)$, we define the interpretation $[\![\Gamma \vdash \mathrm{El}\, A\ \mathsf{Type}]\!]\gamma = \mathrm{El}\,[\overline{A}]_\gamma = \iota(A_\gamma)$.

Note that a code for a small type $A \in \mathsf{Tm}(\Gamma, U)$ is thus defined as an assembly morphism $A : \Gamma \to \nabla\mathsf{PER}$. By Proposition 3.8, any set-theoretic function $A : \Gamma \to \mathsf{PER}$ is also an assembly morphism $A : \Gamma \to \nabla\mathsf{PER}$.

4.2.2 Dependent Product

When interpreting the dependent product type in the assembly model, we will run into an issue in trying to satisfy the type equality $\mathrm{El}\,(\Pi^U(x : A)B(x)) \equiv \Pi(x : A)(\mathrm{El}\, B(x))$. This stems from the fact that the categories Mod and PER are only equivalent, not isomorphic. In particular, we will need to make a case distinction within the interpretation. We will see the reason more clearly after we have given the definitions, but for now it will require us to define both type- and term constructors in PER for the dependent product. First we define the large product and its associated terms.

Definition 4.2 (Dependent product)**.** *Given types $A \in \mathsf{Ty}(\Gamma)$ and $B \in \mathsf{Ty}(\Gamma.A)$, the type $\Pi(A, B) \in \mathsf{Ty}(\Gamma)$ consists of all dependent morphisms from A to B, defined as follows:*

$$\Pi(A, B)_\gamma = \{f \in \Pi_{a \in A_\gamma} B_{(\gamma,a)} \mid \exists e.\ e \Vdash_{\Pi(A,B)_\gamma} f\}$$
$$e \Vdash_{\Pi(A,B)_\gamma} f \ \textit{ iff }\ \forall n \in \mathbb{N}, \forall a \in A_\gamma.\ n \Vdash_{A_\gamma} a \implies e \cdot n \Vdash_{B_{(\gamma,x)}} f(a)$$

Note that the type $A \to B \in \mathsf{Ty}(\Gamma)$ then consists of all non-dependent morphisms from A to B. Given a term $b \in \mathsf{Tm}(\Gamma.A, B)$, we define a term $\mathsf{abs}^\Pi_{A,B}(b) \in \mathsf{Tm}(\Gamma, \Pi(A, B))$ as

$$\mathsf{abs}^\Pi_{A,B}(b)(\gamma) = \lambda a \in A_\gamma.\ b_{(\gamma,a)}$$

Note that $\mathsf{abs}^\Pi_{A,B}$ is tracked by $\Lambda m.\Lambda z.\Lambda n.\ m \cdot \langle z, n\rangle$. Conversely, given a term $f \in \mathsf{Tm}(\Gamma, \Pi(A, B))$ and $a \in \mathsf{Tm}(\Gamma, A)$, we have a term $\mathsf{app}^\Pi_{A,B}(f, a) \in \mathsf{Tm}(\Gamma, B[\overline{a}])$ defined as follows

$$\mathsf{app}^\Pi_{A,B}(f, a)(\gamma) = f_\gamma(a_\gamma)$$

Note that $\mathsf{app}^\Pi_{A,B}(f, a)$ is tracked by $\Lambda e.\Lambda n.\Lambda z.\ (e \cdot z) \cdot (n \cdot z)$.

We note a useful property of the definition given here, which will allow for impredicativity in the model. To do so, we will call a type $A \in \mathsf{Ty}(\Gamma)$ modest if A_γ is modest for every $\gamma \in \Gamma$.

Proposition 4.3. *For any context Γ, if $B \in \mathsf{Ty}(\Gamma.A)$ is modest, then $\Pi(A, B)$ is modest for any $A \in \mathsf{Ty}(\Gamma)$.*

Proof. It suffices to show that $\Pi(A, B)_\gamma$ is modest. Suppose that we have $e \in \mathbb{N}$ such that $e \Vdash_{\Pi(A,B)_\gamma} f_1, f_2 \in \Pi_{a \in A_\gamma} B_{(\gamma,a)}$. Then for any $a \in A_\gamma$, there is some $n \in \mathbb{N}$ such that $n \Vdash_{A_\gamma} a$ and thus $e \cdot n \Vdash_{B_{(\gamma,x)}} f_1(a), f_2(a)$, which shows that $f_1(a) = f_2(a)$ since B is modest, and thus $f_1 = f_2$, which shows that $\Pi(A, B)$ is modest. □

Now we give analogous definitions for the small impredicative product type.

Definition 4.4 (Small Product)**.** *Given a type $A \in \mathsf{Ty}(\Gamma)$ and a small type $B \in \mathsf{Tm}(\Gamma.A, U)$, we define the small (impredicative) product $\pi(A, B) \in \mathsf{Tm}(\Gamma, U)$ as a Γ-indexed set of PERs, where $\pi(A, B)_\gamma$ is the PER defined as follows:*

$$d \sim_{\pi(A,B)_\gamma} e \ \textit{ iff}$$
$$\forall a \in A_\gamma, \forall m, n \in \mathbb{N}.\ m, n \Vdash_{A_\gamma} a \implies (d \cdot m) \sim_{B_{(\gamma,a)}} (e \cdot n)$$

Note that $\pi(A,B)_\gamma$ relates two codes if they are realisers for the same dependent morphism $f : (a \in A_\gamma) \to B_{(\gamma,a)}$. In fact, it is worth noting that the definition of $\pi(A,B)$ is equivalent to $\mathsf{E}(\Pi(A, \mathrm{El}\, B))$, noting that $\Pi(A, \mathrm{El}\, B)$ is modest since $\mathrm{El}\, B$ is modest. Since $\Pi(A, \mathrm{El}\, B)_\gamma \cong \iota(\mathsf{E}(\Pi(A, \mathrm{El}\, B)_\gamma))$, we can treat the equivalence classes of $\pi(A,B)_\gamma$ as such dependent morphisms, and we may define an equivalence class of $\pi(A,B)_\gamma$ by defining a dependent morphism $f : (a \in A_\gamma) \to B_{(\gamma,a)}$ and giving a realiser for it.

The definitions of the term formers also follow from the previous observation, although we give explicit definitions here.

Definition 4.5 (Small term formers)**.** *Given a term $p : (\gamma \in \Gamma) \to \iota(\pi(A,B)_\gamma)$ and $a \in \mathsf{Tm}(\Gamma, A)$ we define a term $\mathsf{app}^\pi(p,a) : (\gamma \in \Gamma) \to \iota(B(\gamma, a))$ as:*

$$\mathsf{app}^\pi_{A,B}(p,a)(\gamma) = \{d \cdot n \mid d \in p_\gamma \wedge n \Vdash_{A_\gamma} a_\gamma\}$$

It is clear from the definition of $\pi(A,B)$ that $\mathsf{app}^\pi_{A,B}(p,a)(\gamma)$ is an equivalence class of the PER $B(\gamma,a)$. Conversely, if we are given a term $t : ((\gamma,a) \in \Gamma.A) \to \iota(B(\gamma,a))$, we define a term $\mathsf{abs}^\pi_{A,B}(t) : (\gamma \in \Gamma) \to \pi(A,B)(\gamma)$ as:

$$\mathsf{abs}^\pi_{A,B}(t)(\gamma) = \\ \{e \mid \forall a \in A_\gamma.\, \forall n \in \mathbb{N}.\, n \Vdash_{A_\gamma} a \implies (e \cdot n) \in t_{(\gamma,a)}\}$$

Equivalently, $\mathsf{abs}^\pi_{A,B}(t)(\gamma)$ is the set of realisers for the dependent morphism defined by

$$a \mapsto t_{(\gamma,a)} : (a : A) \to B(\gamma, a)$$

which is realised by $\Lambda d.\Lambda m.\Lambda n.\, d \cdot \langle m, n\rangle$.

The definition for the impredicative product is somewhat complicated to work with. However, we will mostly be interested in working with the type $\pi(A,B)$ where A is the type Size, for which we give the interpretation in the following section. Importantly, Size_γ will carry the full realizability structure, meaning that it will be interpreted as ∇S for some set S. With this is mind, the following lemma will be useful.

Lemma 4.6. *Whenever A_γ carries full realisability structure, meaning that $A_\gamma = \nabla S$ for some set S, then there exists an isomorphism of PERs as follows*

$$\pi(A,B)_\gamma \cong \bigcap_{a \in A_\gamma} B_{(\gamma,a)}$$

Proof. Since $A_\gamma = \nabla S$, we have $\forall a \in A_\gamma, \forall m \in \mathbb{N}.\, m \Vdash_{A_\gamma} a$. Thus, the definition of $\pi(A,B)_\gamma$ reduces to:

$$d \sim_{\pi(A,B)_\gamma} e \;\text{ iff }\; \forall a \in A_\gamma, \forall m,n \in \mathbb{N}.\, (d \cdot m) \sim_{B(\gamma,a)} (e \cdot n)$$

We define a PER morphism $h : \pi(A,B)_\gamma \to \bigcap_{a \in A_\gamma} B_{(\gamma,a)}$ as:

$$h([e]_{\pi(A,B)_\gamma}) = [e \cdot n]_{B(\gamma,a)} \quad \text{for an arbitrary } n \in \mathbb{N} \text{ and } a \in A_\gamma$$

Note that we have $e \in \mathrm{dom}(\pi(A,B)_\gamma)$ and so

$$\forall a \in A_\gamma, \forall m,n \in \mathbb{N}.\, (e \cdot m) \sim_{B(\gamma,a)} (e \cdot n)$$

so for each $a \in A_\gamma$ we have $[e \cdot m]_{B(\gamma,a)} = [e \cdot n]_{B(\gamma,a)}$ and thus

$$h([e]_{\pi(A,B)_\gamma}) \in \bigcap_{a \in A_\gamma} B_{(\gamma,a)}$$

Conversely, let $\Lambda k.m$ be a code for the constant function returning an arbitrary $m \in \mathbb{N}$. We define $h^{-1} : \bigcap_{a \in A_\gamma} B_{(\gamma,a)} \to \pi(A,B)_\gamma$ as:

$$h^{-1}([m]_{\bigcap_{a \in A_\gamma} B_{(\gamma,a)}}) = [\Lambda k.m]_{\pi(A,B)_\gamma}$$

Now to see these are mutually inverse, if we have $e \in \mathrm{dom}(\pi(A,B)_\gamma)$, then we get

$$h^{-1}(h([e]_{\pi(A,B)_\gamma})) = [\Lambda k.(e \cdot n)]_{\pi(A,B)_\gamma} = [e]_{\pi(A,B)_\gamma}$$

Note that the last equality holds since $e \in \mathrm{dom}(\pi(A,B)_\gamma)$. If we have $m \in \mathrm{dom}(\bigcap_{a \in A_\gamma} B_{(\gamma,a)}$, then we get the following by definition:

$$h(h^{-1}([m]_{\bigcap_{a \in A_\gamma} B_{(\gamma,a)}})) = [(\Lambda k.m)_{\bigcap_{a \in A_\gamma} B_{(\gamma,a)}} \cdot n] = [m]_{\bigcap_{a \in A_\gamma} B_{(\gamma,a)}}$$

For h to be an isomorphism of PERs, both h and its inverse must be realisable. It is a simple check that h^{-1} is realised by $\Lambda d.\Lambda e.d$ and h is realised by $\Lambda e.e \cdot n$, where $n \in \mathbb{N}$ may be chosen arbitrarily. □

Now that we have given the required definitions for both the large and small product, we give the formal interpretation for the types by making a case distinction on whether the type B in $\Pi(x : A)B$ is interpreted to be modest. To see why this is necessary, consider again the type equality for small product types:

$$\mathrm{El}\,(\Pi^U(x : A)B(x)) \equiv \Pi(x : A)(\mathrm{El}\,B(x))$$

If we were to simply interpret $\Pi(x : A)(\mathrm{El}\,B(x))$ as $\Pi(A, \mathrm{El}\,B)$ and $\Pi^U(x : A)B(x)$ as $\pi(A,B)$, then the structure $\mathrm{El}\,(\pi(x : A)B(x))$ will only be isomorphic to $\Pi(x : A)(\mathrm{El}\,B(x))$, not necessarily equal. This is because the categories Mod and PER are only equivalent, and for a modest set X we only have an isomorphism $\iota(\mathsf{E}(X)) \cong X$. Thus, instead we are required to make a case distinction on whether the dependent type B in $\Pi(x : A)B(x)$ is small. To the best of our knowledge, their is no way around making such a case distinction when working with Tarski universes. One alternative is instead to work with weak Tarski universes, in which the requirement of the type equality for lifting small types is weakened to an equivalence.

$$\begin{aligned}
&[\![\Gamma \vdash \Pi^U(x : A)B(x) : U]\!]\gamma = \pi([\![\Gamma \vdash A\ \mathsf{Type}]\!], [\![\Gamma, x : A \vdash B : U]\!])_\gamma \\
&[\![\Gamma \vdash \Pi(x : A)B(x)\ \mathsf{Type}]\!]\gamma = f([\![\Gamma \vdash A\ \mathsf{Type}]\!], [\![\Gamma, x : A \vdash B\ \mathsf{Type}]\!])_\gamma \\
&\qquad \text{where } f = \begin{cases} \iota \circ \pi \text{ if } \mathrm{rng}([\![\Gamma, x : A \vdash B :\ \mathsf{Type}]\!]) \subseteq \mathsf{Mod} \\ \Pi \text{ otherwise} \end{cases}
\end{aligned}$$

The term formers require similar case distinctions:

$$\begin{aligned}
&[\![\Gamma \vdash \lambda x{:}A.b(x) : \Pi(x : A).B(x)]\!]\gamma = \mathsf{abs}^k_{A,B}([\![\Gamma, x : A \vdash b : B(x)]\!](\gamma) \\
&[\![\Gamma \vdash f\ a : B(a)]\!]\gamma = \mathsf{app}^k_{A,B}([\![\Gamma \vdash f : \Pi(x : A)B(x)]\!], [\![\Gamma \vdash a : A]\!])(\gamma) \\
&\text{where } k = \begin{cases} \pi \text{ if } \mathrm{rng}([\![\Gamma, x : A \vdash B(x)\ \mathsf{Type}]\!]) \subseteq \mathsf{Mod} \\ \Pi \text{ otherwise} \end{cases}
\end{aligned}$$

4.2.3 Dependent Sum

The interpretation of the dependent sum type will require a similar case distinction as the dependent product type. We first give the definition of the large dependent sum type, which is simply a generalisation of context extension, defined as follows.

Definition 4.7 (Dependent sum)**.** *Given a type $A \in \mathsf{Ty}(\Gamma)$ and $B \in \mathsf{Ty}(\Gamma.A)$, the type $\Sigma(A,B) \in \mathsf{Ty}(\Gamma)$ is defined as follows:*

$$\Sigma(A,B)_\gamma = \{(a,b) \mid a \in A_\gamma \wedge b \in B_{(\gamma,a)}\}$$
$$\langle n_1, n_2\rangle \Vdash_{\Sigma(A,B)_\gamma} (a,b) \;\; \textit{iff} \;\; n_1 \Vdash_{A_\gamma} a \;\; \textit{and} \;\; n_2 \Vdash_{B_{(\gamma,a)}} b$$

To interpret the pairing operation, we define a morphism

$$\mathsf{pair}^\Sigma_{A,B} : \Gamma.A.B \to \Gamma.\Sigma(A,B)$$
$$\mathsf{pair}^\Sigma_{A,B}(\gamma, a, b) = (\gamma, (a,b))$$

This morphism is tracked by the pairing operation on the realisers. To interpret the elimination rule, we are given a type $P \in \mathsf{Ty}(\Gamma.\Sigma(A,B))$ and a term $p \in \mathsf{Tm}(\Gamma.A.B, P[\mathsf{pair}^\Sigma_{A,B}])$ and define a term $\mathsf{Ind}^p_\Sigma \in \mathsf{Tm}(\Gamma.\Sigma(A,B), P)$ as follows:

$$\mathsf{Ind}^p_\Sigma(\gamma,(a,b)) = p_{(\gamma,a,b)}$$

If p is tracked by d, then this term is tracked by $\Lambda k.\Lambda n.\; d \cdot \langle k, \mathsf{pr}_1 n, \mathsf{pr}_2 n\rangle$.

Note that here we are being somewhat informal in defining the realiser for the elimination rule, since we only assumed a computable bijection $\langle\cdot,\cdot\rangle$ for pairing. However, such a computable bijection may be defined for n-tuples as well.

For the dependent sum type, it is easy to see that if A and B are modest, then $\Sigma(A,B)$ is modest too. We give an analogous definition for the type constructor in PER. Note that, unlike for the Π-type, we do not require separate term constructors in PER.

Definition 4.8 (Small dependent sum)**.** *Given a small type $A \in \mathsf{Tm}(\Gamma, U)$ and $B \in \mathsf{Tm}(\Gamma.\mathrm{El}\, A, U)$, we define the dependent sum $\sigma(A,B) \in \mathsf{Tm}(\Gamma, U)$ pointwise as the PER $\sim_{\sigma(A,B)(\gamma)}$ defined as follows:*

$$\langle m_1, n_1\rangle \sim_{\sigma(A,B)_\gamma} \langle m_2, n_2\rangle \;\; \textit{iff} \;\; m_1 \sim_{A(\gamma)} m_2 \;\; \textit{and} \;\; n_1 \sim_{B(\gamma,[m_1]_{A(\gamma)})} n_2$$

Note that this definition is equivalent to $\sigma(A,B) = \mathsf{E}(\Sigma(\mathrm{El}\, A, \mathrm{El}\, B))$ and thus we have $\sigma(A,B) \cong \Sigma(\mathrm{El}\, A, \mathrm{El}\, B)$.

This allows us to formally interpret the Σ-type constructors as follows

$$[\![\Gamma \vdash \sigma(x:A)B(x) : U]\!]\gamma = \sigma([\![\Gamma \vdash A : U]\!], [\![\Gamma, x : \mathrm{El}\, A \vdash B(x) : U]\!])_\gamma$$
$$[\![\Gamma \vdash \Sigma(x:A)B(x)\ \mathsf{Type}]\!]\gamma = f([\![\Gamma \vdash A\ \mathsf{Type}]\!], [\![\Gamma, x : A \vdash B(x)\ \mathsf{Type}]\!])_\gamma$$
$$\text{where } f = \begin{cases} \iota \circ \sigma \text{ if } \mathrm{rng}([\![\Gamma \vdash A\ \mathsf{Type}]\!]) \cup \mathrm{rng}([\![\Gamma, x : A \vdash B(x)\ \mathsf{Type}]\!]) \subseteq \mathsf{Mod} \\ \Sigma \text{ otherwise} \end{cases}$$

We omit the formal interpretation for the term constructors, since they follow directly from the definitions given above without any case distinction.

4.2.4 Empty, Unit and Boolean Types

The empty-, unit-, and boolean types will be simpler to interpret in the model, since they do not require any case distinctions. Note that for these types, we also implicitly assume an induction principle, which is also not explicitly specified in the type theory, but is easily defined in the interpretation.

Definition 4.9 (Empty Type)**.** *The code for the empty type $\bot^U \in \mathsf{Tm}(\Gamma, U)$ will be interpreted as the initial PER for each $\gamma \in \Gamma$:*

$$\bot^U(\gamma) = \mathbf{0}$$

This definition allows us to define the empty type as $\mathrm{El}\,(\bot^U)$, which makes the required type equality true by definition.

Definition 4.10 (Unit type)**.** *The small unit type* $\top^U \in \mathsf{Tm}(\Gamma, U)$ *will be interpreted as the terminal PER for each* $\gamma \in \Gamma$*:*

$$\top^U(\gamma) = \mathbf{1}$$

As with the empty type, the actual unit type is interpreted as $\mathrm{El}\,(\top^U)$*, i.e. the terminal assembly for each* γ*, making the required type equality true in the interpretation. The unique term* $\star \in \mathsf{Tm}(\Gamma, \top)$ *is then the unique morphism to the terminal assembly.*

Definition 4.11 (Boolean type)**.** *We define* Bool^U *as the following PER:*

$$\forall m, n \in \mathbb{N}.\ \langle 0, m\rangle \sim_{\mathsf{Bool}^U_\gamma} \langle 0, n\rangle$$
$$\forall m, n \in \mathbb{N}.\ \langle 1, m\rangle \sim_{\mathsf{Bool}^U_\gamma} \langle 1, n\rangle$$

Again, we define the interpretation of the type Bool *as* $\mathrm{El}\,(\mathsf{Bool}^U)$*. The terms* $\mathsf{ff}, \mathsf{tt} \in \mathsf{Tm}(\Gamma, \mathsf{Bool})$ *are defined to be the constant morphisms* $\gamma \mapsto \{\langle 0, m\rangle \mid m \in \mathbb{N}\}$ *and* $\gamma \mapsto \{\langle 1, m\rangle \mid m \in \mathbb{N}\}$*, realised by* $\Lambda k.\langle 0, k\rangle$ *and* $\Lambda k.\langle 1, k\rangle$ *respectively.*

As there are no case distinctions required, the explicit formal interpretations for the theory follow directly from the definitions given here.

4.2.5 Identity Type and Function Extensionality

Definition 4.12 (Identity type)**.** *We interpret the identity type using the following definitions: for any type* $A \in \mathsf{Ty}(\Gamma)$ *and terms* $a, a' \in \mathsf{Tm}(\Gamma, A)$*, we have a type* $\mathsf{Id}(A, a, a') \in \mathsf{Ty}(\Gamma)$ *defined as*

$$\mathsf{Id}(A, a, a')_\gamma = \begin{cases} \mathbf{1} & \text{if } a = a' \\ \mathbf{0} & \text{if } a \neq a' \end{cases}$$

For each term $a \in \mathsf{Tm}(\Gamma, A)$*, we have a term* $\mathsf{refl}_a \in \mathsf{Tm}(\Gamma, \mathsf{Id}(A, a, a))$ *defined by* $\mathsf{refl}_a(\gamma) = *$*.*

Interpreting the induction principle explicitly in the categories with families framework becomes somewhat syntactic. Since this is not the focus of this work, we note instead that it can be shown that it suffices to give a natural isomorphism $\mathsf{Tm}(\Gamma, \mathsf{Id}(A, a, a')) \cong \{\star \mid a = a'\}$ *for each* $a, a' \in \mathsf{Tm}(\Gamma, A)$ *[AG25]. Clearly, both sets are singletons if* $a = a'$*, and empty otherwise.*

Informally, function extensionality holds in the realisability model since it is set-theoretic, so (dependent) functions are equal when they are pointwise equal. Explicitly, suppose we are given $f, g \in \mathsf{Tm}(\Gamma, \Pi(A, B))$, together with a proof of pointwise equality

$$p \in \mathsf{Tm}(\Gamma.A, \mathsf{Id}(B[\overline{a}], \mathsf{app}_{A,B}(f, a), \mathsf{app}_{A,B}(g, a)))$$

. Then for any $\gamma \in \Gamma$ and $a_\gamma \in A_\gamma$ we have that $f_\gamma(a_\gamma) = g_\gamma(a_\gamma)$, thus implying that $f = g$, giving us a term $\mathsf{refl}_f \in \mathsf{Tm}(\Gamma, \mathsf{Id}(\Pi(A, B), f, g))$.

Note that the realisability model thus validates uniqueness of identity proofs, since the identity types are either empty or a singleton, but we do not add this axiom to the type theory.

4.3 Type of Sizes

In other work [NVD17], the type of sizes is interpreted as the set of natural numbers. The impredicativity of our realisability model will allow us to interpret the type Size as a large type, while still having products over Size with a small type as codomain live in U. Thus, given a context Γ, we define the semantic type $\mathsf{Size} \in \mathsf{Ty}(\Gamma)$ as $\mathsf{Size}_\gamma = \nabla\omega_1$, where ω_1 is the first uncountable ordinal. Note that

the set ω_1 is exactly the set of all countable ordinals. Giving the type a full realisability structure encapsulates the idea that sizes should be treated parametrically.

Informally, the reason for interpreting the type of sizes as ω_1 is that it will allow us to justify the equivalence

$$\exists i.(\pi(x : \mathrm{El}\, A).\ \exists j < i.B(x, j)) \simeq \pi(x : \mathrm{El}\, A)(\exists i.B(x, i)) \qquad i \notin \mathrm{fv}(A)$$

Intuitively, the right-to-left direction of this equivalence states that if we are given a dependent function that gives a pair $\langle s, b\rangle : \exists i.B(i)$ for each $a : \mathrm{El}\, A$, there should be a corresponding 'limit' size α such that $s < \alpha$ for each $\langle s, b\rangle$ in the range of the dependent function. As we will see, this will require limits of sequences of sizes to be a size as well. Note that, since $\mathrm{El}\, A$ is interpreted as a partial equivalence relation over $\mathbb{N}$, it has countably many equivalence classes. Since countable limits of countable ordinals are themselves countable, ω_1 will suffice for this purpose.

The operators $\circ \in \mathsf{Tm}(\Gamma, \mathsf{Size})$ and $\uparrow \in \mathsf{Tm}(\Gamma.\mathsf{Size}, \mathsf{Size})$ can be interpreted as the constant-zero and successor functions respectively:

$$\begin{aligned} \circ(\gamma) &= 0 \\ \uparrow(\gamma, \alpha) &= \alpha + 1 \end{aligned}$$

Note that since Size_γ carries a full realisability structure, these are tracked by any total realiser.

We will sometimes treat a term $s \in \mathsf{Tm}(\Gamma, \mathsf{Size})$ of type Size as a constant value in Size_γ, since we build up such terms from the zero, successor and supremum functions, which behave uniformly over Γ.

The ordering on sizes $\leq^U$ is interpreted as a small type in a context with two size variables, i.e. $\leq^U \in \mathsf{Tm}(\Gamma.\mathsf{Size}.\mathsf{Size}[\mathbf{p}_{\mathsf{Size}}], U)$. The small type $s_1 \leq^U s_2$ is either the terminal singleton PER or the initial empty PER, depending on whether $s_1 \leq s_2$ holds with the regular relation on natural numbers.

$$\leq^U (\gamma, s_1, s_2) = \begin{cases} \mathbf{1} & \text{if } s_1 \leq s_2 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

The actual type $\leq \in \mathsf{Ty}(\Gamma.\mathsf{Size}.\mathsf{Size}[\mathbf{p}_{\mathsf{Size}}])$ is then simply the lifting to assemblies: $\leq = \mathrm{El}\,(\leq^U)$. We will often write $(s_1 \leq s_2)_\gamma$ instead of $\leq (\gamma, s_1, s_2)$ and we will blur the distinction between $\leq$ and $\leq^U$, since the important property is that $s_1 \leq s_2$ is pointwise either the initial or terminal object in the category of PERs or assemblies. Note that this interpretation validates the notion that the ordering on sizes in the theory is a proposition.

4.4 Bounded Quantification and Fixpoint Operator

Recall that the notation $\Gamma, i : \mathsf{Size} \vdash \forall j < i.\ A(j)$ is shorthand for $\Gamma, i : \mathsf{Size} \vdash \forall j.\ (j < i \to A(j))$. Thus, a term g of type $\forall j < i.\ A(j)$ is interpreted as a $\Gamma.\mathsf{Size}$-indexed collection of dependent morphisms, where

$$g_{(\gamma,\alpha)} : (\beta \in \mathsf{Size}_\gamma) \to ((\beta < \alpha)_\gamma \to A_{(\gamma,\beta)})$$

If $(\beta < \alpha)_\gamma$ holds in the interpretation, then $(\beta < \alpha)_\gamma$ is the terminal singleton assembly. and thus the set of morphisms of type $(\beta < \alpha)_\gamma \to A_{(\gamma,\beta)}$ is isomorphic to $A_{(\gamma,\beta)}$, allowing us to treat $g_{(\gamma,\alpha)}(\beta)$ as an element of $A_{(\gamma,\beta)}$. If $(\beta < \alpha)_\gamma$ does not hold in the interpretation, then $(\beta < \alpha)_\gamma$ is the initial (empty) assembly and there thus exists a unique morphism of type $(\beta < \alpha)_\gamma \to A_{(\gamma,\beta)}$, namely the empty morphism, again given by $g_{(\gamma,\alpha)}(\beta)$. This confirms the syntactic intuition that we may treat $g_{(\gamma,\alpha)}$ as a dependent morphism where $\mathrm{dom}(g_{(\gamma,\alpha)}) = \mathsf{Size}_\gamma \upharpoonright \alpha$. We will thus write the type of $g_{(\gamma,\alpha)}$ as:

$$g_{(\gamma,\alpha)} : (\beta \in \mathsf{Size}_{<\alpha,\gamma}) \to A_{(\gamma,\beta)}$$

Given this intuition, we consider the interpretation of the fixpoint operator fix. Given a type $A \in \mathsf{Ty}(\Gamma.\mathsf{Size})$, and a dependent morphism of the following type

$$f_\gamma : (\alpha : \mathsf{Size}_\gamma) \to ((\beta \in \mathsf{Size}_{<\alpha,\gamma}) \to A_{(\gamma,\beta)}) \to A_{(\gamma,\alpha)}$$

corresponding to a semantic term of the syntactic type $\forall i.(\forall j < i.A(j) \to A(i))$, we need to define a term $\mathsf{fix}\, f : (\gamma \in \Gamma) \to \Pi(\mathsf{Size}, A)_\gamma$. Thus, we require that $(\mathsf{fix}\, f)_\gamma$ is a dependent morphism from Size to A, i.e.

$$(\mathsf{fix}\, f)_\gamma : (\alpha \in \mathsf{Size}_\gamma) \to A_{(\gamma,\alpha)}$$

We define this morphism by well-founded induction on Size_γ, i.e. the ordinal ω_1 and thus a well-founded ordering:

$$(\mathsf{fix}\, f)_\gamma(\alpha) = f_\gamma(\alpha, \lambda\beta < \alpha.\ (\mathsf{fix}\, f)_\gamma(\beta))$$

Note that, in the case where $\alpha = 0$, there is only the unique empty morphism with type $(\beta \in \mathsf{Size}_{<0,\gamma}) \to A_{(\gamma,\beta)}$, since the set $\mathsf{Size}_{<0,\gamma}$ is empty. Note furthermore that this definition mirrors exactly the β-rule for fix. The uniqueness property for $\mathsf{fix}\, f$ also follows by well-founded induction over Size_γ.

To be precise, the type $\forall i.A(i)$ may be interpreted as $\iota(\forall(\mathsf{Size}, A))$ when A is a small type. In this case, we can make use of the previous morphism as well, only applying the isomorphism $\iota(\forall(\mathsf{Size}, A)) \cong \Pi(\mathsf{Size}, \mathrm{El}\, A)$.

For fix to be a term, it must be tracked by some realiser. Note that, by our definition of fix, if we are given some f realised by $f_r \in \mathbb{N}$ and γ realised by $\gamma_r \in \mathbb{N}$, we require that a realiser F for fix satisfying the following for every $n \in \mathbb{N}$, where we use as convention that the partial operation associates to the left so that we may omit parentheses:

$$F \cdot f_r \cdot \gamma_r \cdot n = f_r \cdot \gamma_r \cdot n \cdot (\Lambda m.\ F \cdot f_r \cdot \gamma_r \cdot m)$$

In order to define such a realiser, we make use of the fact that a fixpoint operator exists in every partial combinatory algebra:

Proposition 4.13 (Fixpoint operator in pca)**.** *For every pca $(\mathcal{A}, \cdot)$, there exists a $\mathsf{fix} \in \mathcal{A}$ such that*

$$\mathsf{fix}\, f \downarrow \ \textit{and}\ \mathsf{fix}\, f\, a \simeq f\, (\mathsf{fix}\, f)\, a$$

A proof of this can be found in [Oos08]. Importantly, it allows us to define a realiser for the fixpoint operator. First, we define a code $\varphi \in \mathbb{N}$ by

$$\varphi := \Lambda f.\Lambda e.\Lambda k.\Lambda n.\ e \cdot k \cdot n \cdot (\Lambda m.\ f \cdot e \cdot k \cdot m)$$

This gives us the following:

$$\begin{aligned}(\mathsf{fix}\, \varphi) \cdot f_r \cdot \gamma_r \cdot n &\simeq \varphi \cdot (\mathsf{fix}\, \varphi) \cdot f_r \cdot \gamma_r \cdot n \\ &\simeq f_r \cdot \gamma_r \cdot n \cdot (\Lambda m.(\mathsf{fix}\, \varphi) \cdot f_r \cdot \gamma_r \cdot m)\end{aligned}$$

This shows that $\mathsf{fix}\, \varphi$ is a realiser for our fixpoint operator, which thus constitutes a term.

4.5 Existential Type

To interpret the existential type $\exists i.A(i)$, we will need to quotient the corresponding Σ-type in order for it to live inside the universe U, i.e. canonically construct a partial equivalence relation from the assembly $\Sigma(\mathsf{Size}, A)$. Instead of creating a construction for just this type, we can do this more generally for any type. To do so, we make use of the following lemma.

Lemma 4.14. *The inclusion functor* $I : \mathsf{Mod} \to \mathsf{Asm}$ *has a left adjoint* $\mathsf{M} : \mathsf{Asm} \to \mathsf{Mod}$.

Proof. Let $A \in \mathsf{Asm}$ be an assembly, we define $\mathsf{M}(A) = (A/\sim_{\mathsf{M}}, \Vdash_{\mathsf{M}(A)})$ where $\sim_{\mathsf{M}}$ is the least equivalence relation such that the following holds:

$$\exists n \in \mathbb{N}.\ n \Vdash_A a_1, a_2 \implies a_1 \sim_{\mathsf{M}} a_2$$

The realisability relation is defined as

$$n \Vdash_{\mathsf{M}(A)} [a]_{\sim_{\mathsf{M}}} \text{ iff } \exists a' \in [a]_{\sim_{\mathsf{M}}} (n \Vdash_A a')$$

It is clear that $\mathsf{M}(A)$ is indeed a modest set. We define a map $\eta_A : A \to \mathsf{M}(A)$ as $a \mapsto [a]_{\sim_{\mathsf{M}}}$, which is realised by $\Lambda x.x$ and is thus a morphism of assemblies. To show that M extends to a left adjoint of I, let $X \in \mathsf{Mod}$ be a modest set and suppose there exists a morphism $g : A \to X$. Then g uniquely factors as $f \circ \eta_A$, where $f : \mathsf{M}(A) \to X$ is defined as $f([a]_{\sim_{\mathsf{M}}}) = g(a)$.

It remains to show that f is well-defined. Suppose that $a_1 \sim_{\mathsf{M}} a_2$; if there exists $n \in \mathbb{N}$ such that $n \Vdash_A a_1, a_2$, then since g is tracked by some $e \in \mathbb{N}$, this means that $e \cdot n \Vdash_X g(a_1), g(a_2)$. Thus, since X is a modest set, $g(a_1) = g(a_2)$ and so f is well-defined.

Else, there exist $n_1, ..., n_k \in \mathbb{N}$ and $b_1, ..., b_{k-1} \in A$ such that $n_1 \Vdash_A a_1, b_1$, $n_2 \Vdash_A b_1, b_2$, ..., and $n_k \Vdash_A b_{k-1}, a_2$. Then by similar reasoning via transitivity, we will get that $g(a_1) = g(b_1) = \cdots = g(b_{k-1}) = g(a_2)$ □

Given this construction, we can truncate any assembly $A \in \mathsf{Asm}$ to $\mathsf{M}(A)$ in order (almost) fit in the universe U. However, the universe U consists of PERs rather than modest sets. Fortunately, the categories PER and Mod are equivalent via the functor $\mathsf{E} : \mathsf{Mod} \to \mathsf{PER}$ defined in Section 3.2. Thus, we will define the truncation $\|A\| \in \mathsf{Tm}(\Gamma, U)$ of a type $A \in \mathsf{Ty}(\Gamma)$ as

$$\|A\|_\gamma = \mathsf{EM}(A_\gamma)$$

However, in practice it will be more intuitive to work with $\mathsf{M}(A_\gamma)$ than $\mathsf{EM}(A_\gamma)$ in the following sections. Since $\mathsf{M}(A_\gamma) \cong \iota(\mathsf{EM}(A_\gamma))$, this does not present any theoretical issues. For a type $A \in \mathsf{Ty}(\Gamma)$, we might denote an element of $\|A\|_\gamma$ as $[a]_{\sim_{\mathsf{M}}}$ where $a \in A_\gamma$. In reality, however, there exists a unique $[n]_{\mathsf{EM}(A_\gamma)} \in \mathsf{EM}(A_\gamma)$ such that $n \Vdash_{\mathsf{M}(A_\gamma)} [a]_{\sim_{\mathsf{M}}}$, where $[a]_{\sim_{\mathsf{M}}} \in \mathsf{M}(A_\gamma)$. Thus, instead of working with the equivalence class of the corresponding PER, we will work with the equivalence class of the modest set.

The truncation operation should also operate on the level of terms, that is, given a term $a \in \mathsf{Tm}(\Gamma, A)$, there should be a term $|a|$ in the truncated type. Thus, we define a context morphism

$$|\cdot| : \Gamma.A \to \Gamma.\mathrm{El}\,(\|A\|)$$
$$|(\gamma, a)| = (\gamma, [a]_{\sim_{\mathsf{M}}})$$

which is realised by the identity $\Lambda x.x$. Moreover, the existential type comes with an elimination rule for small types. Thus, a truncated type $\|A\|$ in the model should come with such a rule as well. That is, given a small type $P \in \mathsf{Tm}(\Gamma.\mathrm{El}\,\|A\|, U)$ and a term $t \in \mathsf{Tm}(\Gamma.A, (\mathrm{El}\,P)[|\cdot|])$, we define the following term

$$\mathsf{elim}_t \in \mathsf{Tm}(\Gamma.\mathrm{El}\,(\|A\|), \mathrm{El}\,P)$$
$$\mathsf{elim}_t(\gamma, [a]_{\sim_{\mathsf{M}}}) = t(\gamma, a)$$

To show that elim_t is well defined, we need to show that $a_1 \sim_M a_2$ implies $t(\gamma, a_1) = t(\gamma, a_2)$. Since $a_1 \sim_M a_2$, we can assume that there exists some $k \in \mathbb{N}$ such that $k \Vdash_A a_1, a_2$. Moreover, t is tracked by some $e \in \mathbb{N}$, meaning that the following holds:

$$\langle m, n \rangle \Vdash_{\Gamma.A} (\gamma, a) \implies e \cdot \langle m, n \rangle \Vdash_{P(\gamma, [a]_{\sim_{\mathsf{M}}})} t(\gamma, a)$$

Then for any $\gamma \in \Gamma$, which must realised by some $m \Vdash_\Gamma \gamma$, we have that $\langle m, k\rangle \Vdash_{\Gamma.A} (\gamma, a_1), (\gamma, a_2)$ and thus:

$$e \cdot \langle m, k\rangle \Vdash_{P(\gamma, [a_1]_{\sim_M})} t(\gamma, a_1), t(\gamma, a_2)$$

since $[a_1]_{\sim_M} = [a_2]_{\sim_M}$. Since P is a small type, $P(\gamma, [a_1]_{\sim_M})$ is a modest set when viewed as an assembly. This implies that $t(\gamma, a_1) = t(\gamma, a_2)$ since any realiser codes at most one object.

Now we are ready to interpret the existential type by truncating the Σ-type.

Definition 4.15 (Existential Type)**.** *Given a small type $A \in \mathsf{Tm}(\Gamma.\mathsf{Size}, U)$, the small existential type $\exists i.A(i) \in \mathsf{Tm}(\Gamma, U)$ is defined as*

$$\exists i.A(i) := \|\Sigma(\mathsf{Size}, \mathrm{El}\, A)\|$$

As with dependent sums, the pairing operation is interpreted with a morphism $\mathsf{pair}_\exists : \Gamma.\mathsf{Size}.A \to \Gamma.\mathrm{El}\,(\exists i.A(i))$ defined as

$$\mathsf{pair}_\exists(\gamma, s, a) = |\mathsf{pair}_\Sigma(\gamma, s, a)|$$

where $\mathsf{pair}_\Sigma(\gamma, s, a) \in \Gamma.\Sigma(\mathsf{Size}, A)$.

Finally, the elimination rule for existential types is interpreted as follows. Given the following terms:

$$P \in \mathsf{Tm}(\Gamma.\mathrm{El}\,(\exists i.A(i)), U)$$
$$p \in \mathsf{Tm}(\Gamma.\mathsf{Size}.A, (\mathrm{El}\, P)\{\mathsf{pair}_\exists\})$$

we define the term $\mathsf{Ind}^p_\exists \in \mathsf{Tm}(\Gamma.(\exists i.A(i)), \mathrm{El}\, P)$ as follows:

$$\mathsf{Ind}^p_\exists = \mathsf{elim}_{\mathsf{Ind}^P_\Sigma} \in \mathsf{Tm}(\Gamma.(\exists i.A(i)), \mathrm{El}\, P)$$

where $\mathsf{elim}_{\mathsf{Ind}^P_\Sigma}(\gamma, [(s, a)]_{\sim_\mathsf{M}}) = \mathsf{Ind}^P_\Sigma(\gamma, (s, a))$ is well-defined since elim is well-defined.

Now that the formal definition is given, we can make an observation about the structure of the existential type. Given a small type $A \in \mathsf{Tm}(\Gamma, U)$, note that for the modest set $\mathsf{M}(\Sigma(\mathsf{Size}, \mathrm{El}\, A)_\gamma)$, the equivalence relation $\sim_\mathsf{M}$ is defined exactly by

$$\begin{aligned}(s_1, a_1) \sim_\mathsf{M} (s_2, a_2) \;\text{ iff }\; & \exists \langle m, n\rangle \in \mathbb{N}.\ \langle m, n\rangle \Vdash_{\Sigma(\mathsf{Size}, \mathrm{El}\, A)_\gamma} (s_1, a_1), (s_2, a_2) \\ \text{iff }\; & \exists n \in \mathbb{N}.\ n \Vdash_{\mathrm{El}\, A_{(\gamma, s_1)}} a_1 \;\text{ and }\; n \Vdash_{\mathrm{El}\, A_{(\gamma, s_2)}} a_2\end{aligned}$$

Thus, the intuition is that two elements $[(s_1, a_1)]_{\sim_\mathsf{M}}$ and $[(s_2, a_2)]_{\sim_\mathsf{M}}$ of $(\exists i.A(i))_\gamma$ are equal if a_1 and a_2 share a realiser.

4.6 Type Equivalences in the Model

To justify the equivalences included in the type theory in Figure 2.13, we show that they hold in the model. To show that the equivalence $\Gamma \vdash A \simeq B$ holds, we give an isomorphism $A \cong B$ where $A, B \in \mathsf{Tm}(\Gamma, U)$ by giving an isomorphism of PERs $A_\gamma \cong B_\gamma$ for each $\gamma \in \Gamma$. Note that giving such an isomorphism between PERs is stronger than the axiom added to the theory. In fact, mirroring these results from the model, we could have stated the stronger axioms as defining inverses up to definitional equality, as has been done in guarded recursion [Møg14]. However, this would be a external axiom, whereas our approach is more in line with other work on internal parametricity.

4.6.1 Universal Quantification

Recalling Lemma 4.6, we know that for any small type $A : U$, the type $\Gamma \vdash \forall i.\ A(i) : U$ is interpreted as $\pi(\mathsf{Size}, A) : \Gamma \to \mathsf{PER}$ and satisfies the isomorphism

$$\pi(\mathsf{Size}, A)_\gamma \cong \bigcap_{s \in \mathsf{Size}_\gamma} A_{\gamma,s}$$

since Size_γ carries full realisability structure. Moreover, recall that the small type $\Gamma, i : \mathsf{Size} \vdash \forall j < i.\ A(j) : U$ is interpreted as the set of dependent morphisms with the following type:

$$((\gamma, s) \in \Gamma.\mathsf{Size}) \to ((r \in \mathsf{Size}_{<s,\gamma}) \to A(\gamma, r))$$

It follows that the interpretation of $\Gamma, i : \mathsf{Size} \vdash \forall j < i.\ A(j) : U$ can be characterized by the following isomorphism:

$$(\forall j < i.\ A)_{(\gamma,s)} \cong \bigcap_{r \in \mathsf{Size}_{<s,\gamma}} A_{(\gamma,r)}$$

We will write the interpretation of $\Gamma, i : \mathsf{Size} \vdash \forall j < i.\ A(j) : U$ as $\pi(\mathsf{Size}_<, A)$.

- $A(i) \simeq \forall i.A$: We recall Lemma 4.6:

  $$\pi(\mathsf{Size}, A)_\gamma \cong \bigcap_{s \in \mathsf{Size}_\gamma} A_{\gamma,s}$$

  Since $i \notin \mathrm{fv}(A)$, A is not dependent on Size, so we have that $A_{\gamma,s_1} = A_{\gamma,s_2}$ for any $s_1, s_2 \in \mathsf{Size}_\gamma$. Thus we have $\bigcap_{s \in \mathsf{Size}_\gamma} A_{\gamma,s} = A_\gamma$ and so $\pi(\mathsf{Size}, A)_\gamma \cong A_\gamma$.

  Note that if $i \notin \mathrm{fv}(A)$, then an element $[e] \in \pi(\mathsf{Size}, A)_\gamma$ corresponds to a unique morphism $f : \mathsf{Size}_\gamma \to A_\gamma$ tracked by $e \in \mathbb{N}$. Then for any $n \in \mathbb{N}$ and $s_1, s_2 \in \mathsf{Size}_\gamma$, we have $n \Vdash_{\mathsf{Size}_\gamma} s_1, s_2$ and so $e \cdot n \Vdash_{A_\gamma} f(s_1), f(s_2) \implies f(s_1) = f(s_2)$ since A_γ is modest. Thus, f is simply a constant function with codomain A_γ, which also justifies the isomorphism $\pi(\mathsf{Size}, A)_\gamma \cong A_\gamma$ when $i \notin \mathrm{fv}(A)$.

- $\forall i.\ A(i) \simeq \forall i.\ \forall j < i.\ A(j)$: Here we have the isomorphism

  $$\pi(\mathsf{Size}, A)_\gamma \cong \bigcap_{s \in \mathsf{Size}_\gamma} A_{\gamma,s}$$

  by Lemma 4.6. Similarly, by our earlier reasoning in this section, we get the following isomorphism:

  $$\pi(\mathsf{Size}, \pi(\mathsf{Size}_<, A))_\gamma \cong \bigcap_{s \in \mathsf{Size}_\gamma} \bigcap_{r \in \mathsf{Size}_{<s,\gamma}} A_{\gamma,r}$$

  It is clear that these are isomorphic PERs, justifying the equivalence.

- $\forall i.(\Sigma^U(x : A)B(i, x)) \simeq \Sigma^U(x : A)(\forall i.B(x, i))$: Recall that here we assume that $i \notin \mathrm{fv}(A)$. Since $\sigma(A, B) \cong \Sigma(\mathrm{El}\, A, \mathrm{El}\, B)$ and $\pi(A, B) \cong \Pi(A, \mathrm{El}\, B)$, it suffices to show that the following isomorphism holds:

  $$\Pi(\mathsf{Size}, \Sigma(\mathrm{El}\, A, \mathrm{El}\, B)) \cong \Sigma(\mathrm{El}\, A, \Pi(\mathsf{Size}, \mathrm{El}\, B))$$

  So suppose that $f \in \Pi(\mathsf{Size}, \Sigma(\mathrm{El}\, A, \mathrm{El}\, B))_\gamma$ is a dependent morphism tracked by $e \in \mathbb{N}$. Then we note the following property of f:

  $$f(s_1) = (a_1, b_1) \wedge f(s_2)(a_2, b_2) \implies a_1 = a_2$$

This holds because s_1, s_2 are tracked by any $n \in \mathbb{N}$ and so, using the fact that A does not depend on Size, we have:

$$e \cdot n \Vdash_{\Sigma(A,B)_{(\gamma,s_1)}} (a_1, b_1) \implies \mathsf{pr}_1(e \cdot n) \Vdash_{A_\gamma} a_1$$
$$e \cdot n \Vdash_{\Sigma(A,B)_{(\gamma,s_2)}} (a_2, b_2) \implies \mathsf{pr}_1(e \cdot n) \Vdash_{A_\gamma} a_2$$

Since A is modest, we thus have that $a_1 = a_2$. So, given f, we construct the element $(\pi_1(f(r)), \lambda s.\ \pi_2(f(s))) \in \Sigma(\mathrm{El}\, A, \Pi(\mathsf{Size}, \mathrm{El}\, B))_\gamma$ for an arbitrary $r \in \mathsf{Size}$. This map is tracked by $\Lambda e.\ \langle \mathsf{pr}_1\,(e \cdot n), \Lambda k.\ \mathsf{pr}_2\,(e \cdot k)\rangle$ for an arbitrary $n \in \mathbb{N}$.

Conversely, given $(a, f) \in \Sigma(\mathrm{El}\, A, \Pi(\mathsf{Size}, \mathrm{El}\, B))_\gamma$, we can construct an element $\lambda s.\ (a, f\, s) \in \Pi(\mathsf{Size}, \Sigma(\mathrm{El}\, A, \mathrm{El}\, B))_\gamma$. This map is tracked by $\Lambda k.\Lambda n.\langle \mathbf{p}_1\, k, (\mathsf{pr}_2\, k) \cdot n\rangle$. Moreover, it is easily seen that these maps form an isomorphism.

4.6.2 Existential Quantification

- $A \simeq \exists i.A$: In this case we have $i \notin \mathrm{fv}(A)$, so $A_{\gamma,s_1} = A_{\gamma,s_2}$ for all $s_1, s_2 \in \mathsf{Size}_\gamma$, so we simply use the notation A_γ. The equivalence relation $\sim_{\mathsf{M}}$ on $\Sigma(\mathsf{Size}, \mathrm{El}\, A)_\gamma$ can then be characterized as follows, since A_γ is a modest set:

$$\begin{aligned}(s_1, a_1) \sim_{\mathsf{M}} (s_2, a_2) \text{ iff } & \exists \langle m, n\rangle \Vdash_{\Sigma(\mathsf{Size}, \mathrm{El}\, A)_\gamma} (s_1, a_1), (s_2, a_2)\\ \text{iff } & \exists n.\ n \Vdash_{A_{\gamma,s_1}} a_1 \wedge n \Vdash_{A_{\gamma,s_1}} a_2\\ \text{iff } & \exists n.\ n \Vdash_{A_\gamma} a_1 \wedge n \Vdash_{A_\gamma} a_2\\ \text{iff } & a_1 = a_2\end{aligned}$$

  Thus, we get an isomorphism $h : \mathsf{M}(\Sigma(\mathsf{Size}, A))_\gamma \cong A_\gamma$ where $h([(s, a)]_{\sim_{\mathsf{M}}}) = a$ and $h^{-1}(a) = [(s, a)]_{\sim_{\mathsf{M}}}$ for any $s \in \mathsf{Size}_\gamma$. Here, h is realised by the second projection, and h^{-1} by $\Lambda k.\langle m, k\rangle$ for any fixed $m \in \mathbb{N}$.

- $\exists i.\ A(i) \simeq \exists i.\ \exists j < i.\ A(j)$: We define an isomorphism

$$h : \mathsf{M}(\Sigma(\mathsf{Size}, A))_\gamma \cong \mathsf{M}(\Sigma(\mathsf{Size}, \Sigma(\mathsf{Size}, A)))_\gamma$$

  where $h([(s, a)]_{\sim_{\mathsf{M}}}) = [(\uparrow s, (s, a))]_{\sim_{\mathsf{M}}}$ and $h^{-1}([s, (r, a)]_{\sim_{\mathsf{M}}}) = [(r, a)]_{\sim_{\mathsf{M}}}$. It is easily checked that these mappings are well-defined since the equivalence relation $\sim_{\mathsf{M}}$ depends only on a. To show that h is an isomorphism, it suffices to show that for any $s_1, s_2 \in \mathsf{Size}_\gamma$ we have $(s_1, (s, a)) \sim_{\mathsf{M}} (s_2, (s, a))$. This follows from the fact that in the type $\exists i.\ \exists j < i.\ A(j)$, A depends only on j. Thus, the same reasoning from the previous case gives the required result.

- $\exists i.(\Pi^U(x : \mathrm{El}\, A).\ \exists j < i.B(x, j)) \simeq \Pi^U(x : \mathrm{El}\, A)(\exists i.B(x, i))$: Here we recall that we assume that $i \notin \mathrm{fv}(A)$. To define this equivalence, we make use of the fact that Size_γ is interpreted as the ordinal ω_1, allowing us to take countable limits of countable sequences over Size_γ.

  Since $\sigma(A, B) \cong \Sigma(\mathrm{El}\, A, \mathrm{El}\, B)$ and $\pi(A, B) \cong \Pi(A, \mathrm{El}\, B)$, it suffices to show that the following isomorphism holds:

$$\mathsf{M}(\Sigma(\mathsf{Size}, \Pi(\mathrm{El}\, A, \mathsf{M}(\Sigma(\mathsf{Size}_<, B)))))_\gamma \cong \Pi(\mathrm{El}\, A, \mathsf{M}(\Sigma(\mathsf{Size}, B)))_\gamma$$

  The nested occurrences of the functor M makes this a difficult representation to work with. Instead, it is more instructive to define a map

$$\Sigma(\mathsf{Size}, \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}_<, B)))_\gamma \to \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}, B))_\gamma$$

  which sends a pair (s, f) to f. Since $f \in \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}_<, B))_\gamma$, we have that f sends an element $a \in \mathrm{El}\, A_\gamma$ to a pair (r, b) where $r < s$. In particular then, we also have that $(r, b) \in \Sigma(\mathsf{Size}, B)_{(\gamma,a)}$.

Conversely, to define a map

$$\Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}, B))_\gamma \to \Sigma(\mathsf{Size}, \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}_<, B)))_\gamma$$

we make use of the fact that we can take limits of countable sequences. Thus, we send $g \in \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}, B))_\gamma$ to the pair $(\mathsf{sup}_{(s,b)\in \mathrm{rng}(g)}\langle s\rangle, g)$. Note that the first element of this pair is the supremum of all sizes in the range of g, and that the range of g is countable because $(\mathrm{El}\, A)_\gamma$ is interpreted as a PER, and is thus countable. Thus, for any $(s, b) \in \mathrm{rng}(g)$ we have that $s < \mathsf{sup}_{(s,b)\in \mathrm{rng}(g)}\langle s\rangle$, making this map also well-defined.

Note that these maps do not form an isomorphism

$$\Sigma(\mathsf{Size}, \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}_<, B))) \cong \Pi(\mathrm{El}\, A, \Sigma(\mathsf{Size}, B))$$

However, lifting these maps to work with their quotients via M does give the required isomorphism.

Chapter 5

(Co)inductive Types

Instead of defining inductive and coinductive types and their terms as primitives, we will construct them as the fixpoints of suitable functors in the type theory. This will allow us to make the intuition of size-indexed types as approximations of full types explicit.

5.1 (Co)inductive Types as (Co)algebras

Dependently-typed programming languages such as Agda or Rocq offer explicit methods to define inductive datatypes, such as lists or finite trees, by specifying the constructors of the type. More recently, Agda also introduced functionality to define coinductive types via copatterns, which essentially allow the user to define a coinductive type by specifying the destructors of the type. We take an alternative approach to explicitly including certain (co)inductive types in the syntax. Namely, we will encode inductive types and coinductive types as initial algebras and terminal coalgebras respectively, for suitable functors in the category of small types. There are several good introductions to this categorical perspective on (co)inductive datatypes, of which we refer to just one [JR97].

As mentioned before, in the type theory we would like to work with both (co)inductive types and their size-indexed variants, which should themselves also be (co)inductive types. We can illustrate the categorical perspective via typical examples of inductive and coinductive types: lists and streams over a type A. In Agda, lists are intuitively defined as the 'least' type generated by the following two constructors:

$$\begin{aligned} \mathsf{nil} &: \mathsf{List}\, A \\ \mathsf{cons} &: A \to \mathsf{List}\, A \to \mathsf{List}\, A \end{aligned}$$

We might encode lists over a small type $A : U$ as the initial algebra of the functor $F\, X = \top + A \times X$, where $F : U \to U$. The type $\mathsf{List}\, A$ being the initial algebra of F would imply that $\mathsf{List}\, A \simeq \top + A \times \mathsf{List}\, A$, matching the intuition that lists are either empty ($\top$) or constructed by adding an element onto an existing list ($A \times \mathsf{List}\, A$). The corresponding size-indexed type, $\mathsf{SList}\, A : \mathsf{Size} \to U$, might come with the following constructors:

$$\begin{aligned} \mathsf{nil}_s &: \mathsf{SList}\, A\, s \\ \mathsf{cons}_s &: \forall r < s.\ A \to \mathsf{SList}\, A\, r \to \mathsf{SList}\, A\, s \end{aligned}$$

Here, the constructor cons_s could also have been defined as

$$\mathsf{cons}_s : A \to \exists r < s. \mathsf{SList}\, A\, r \to \mathsf{SList}\, a\, s$$

This is because we have an equivalence of types $\forall s.\ (A(s) \to B) \simeq \exists s. A(s) \to B$ when s does not occur in B. This allows us to encode the size-indexed list type as the initial algebra for the size-indexed functor $G\, X\, s = \top + \exists r < s.\ A \times X\, r$ where $G : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U)$. We will use this

construction of size-indexed inductive types to derive regular inductive types. A similar construction works for coinductive types. The following sections make the notion of (size-indexed) functors and (co)-algebras precise.

5.2 Uniqueness of Identity Proofs

In this chapter, we will need to assume an additional axiom: *uniqueness of identity proofs* (UIP) for small types. This axiom states that for any $A : U$ and $a, b : \mathrm{El}\, A$, any two terms of the type $a =_A b$ are propositionally equal:

$$\Pi(A : U).\ \Pi(a, b : \mathrm{El}\, A).\ \Pi(p, q : a =_A b)\ (p =_{a=_A b} q)$$

This axiom will simplify the following development, since we don't need to worry about the manipulation of identity types. It is also true in the realisability model. However, assuming UIP means that this development cannot be formulated in homotopy type theory, as it rejects the axiom. We have also worked through most of the following work without UIP and strongly believe that the same results hold. In particular, we require contractibility of the Σ-type pairing the fixpoint of a function with its propositional unfolding to ensure that the size-indexed algebra we define in the following sections satisfies initiality. Moreover, the adjunctions we construct require care to ensure that they work with identity types. However, we do not have all the required details for identity proofs spelled out, so we opt to use UIP here for simplicity.

5.3 Size-indexed Functors

In this section and the following, we will fix a context Γ, and all definitions will be parameterised by the context. Moreover, we will make extensive implicit use of the functional extensionality axiom in what follows. Given a fixed context Γ, we can define the category of small types, i.e. types in the universe, as follows.

Definition 5.1 (Category of small types)**.** *The* category of small types *has as objects small types $A : U$ and as morphisms functions $f : \mathrm{El}\, A \to \mathrm{El}\, B$. Since we will only be working with small types in these sections, we will denote this by $f : A \to B$. The notion of equality is propositional equality. For an object $A : U$, the identity morphism is given by $\mathsf{id}_A := \lambda x.x : A \to A$. Given $f : A \to B$ and $g : B \to C$, we define their composition as $g \circ f := \lambda x.\ g\,(f\,x)$. Note that, making use of the η-rule for Π-types, we have associativity of composition up to definitional equality, and identities are up to definitional equality.*

We define an endofunctor on this category as a pair of terms (F_0, F_1) where

$$F_0 : U \to U$$
$$F_1 : \Pi_{A,B:U}\ (A \to B) \to (FA \to FB)$$

such that the functor laws are satisfied propositionally:

$$F_{A,A}(\mathsf{id}_A) = \mathsf{id}_{FA}$$
$$F_{A,C}(g \circ f) = F_{B,C}\, g \circ F_{A,B} f$$

Note that we will write F for both F_0 and F_1 when there is no confusion, as is commonly done in category theory, and will omit the subscripts for F_1 when they are clear from the context. We also define the category of size-indexed types, which we will use to develop encodings for size-indexed (co)inductive types.

Definition 5.2 (Category of size-indexed types)**.** *The category of size-indexed types has as objects terms* $A : \mathsf{Size} \to U$*. Morphisms in this category are then terms of the form* $f : \forall s.\ (A\,s) \to (B\,s)$*, which we denote by* $f : A \xrightarrow{s} B$*. Given* $f : A \xrightarrow{s} B$ *and* $g : B \xrightarrow{s} C$*, we define the composition as* $g \circ f : A \xrightarrow{s} C$ *as*

$$g \circ f := \lambda s.(g\,s) \circ (f\,s) := \lambda s.\lambda x.\ g\,s\,(f\,s\,x)$$

where the second composition refers to the notion defined in the category of small types. Again, we have associativity of composition up to definitional equality. Note that the identity map $\mathsf{id}_A : A \xrightarrow{s} A$ *here is* $\lambda s.\lambda x.\ x$*.*

An endofunctor (F_0, F_1) *on this category, which we will often called* size-indexed functors*, similarly consists of terms*

$$F_0 : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U)$$
$$F_1 : \Pi_{(A,B:\mathsf{Size}\to U)}\,(A \xrightarrow{s} B) \to (FA \xrightarrow{s} FB)$$

satisfying the same requirements of identity and composition as in the category of small types:

$$F_{A,A}(\mathsf{id}_A) = \mathsf{id}_{FA}$$
$$F_{A,C}(g \circ f) = F_{B,C}g \circ F_{A,B}f$$

Following the notation in [Vez15; NVD17], we now give two important size-indexed functors, i.e. endofunctors on the category of size-indexed types, for bounded existential and universal quantification. In many of the following constructions, function extensionality is key to them being well-defined.

Proposition 5.3. *The following terms constitute a size-indexed functor* $(\Diamond_0, \Diamond_1)$*.*

$$\Diamond_0 : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U)$$
$$\Diamond_0\,A\,s := \exists r < s.\ A\,r$$
$$\Diamond_1 : \Pi_{(A,B:\mathsf{Size}\to U)}(A \xrightarrow{s} B) \to (\Diamond_0 A \xrightarrow{s} \Diamond_0 B)$$
$$\Diamond_{A,B}\,f\,s\,\langle r, a\rangle := \langle r, f\,r\,a\rangle$$

Proof. The functor laws for identity and composition follow from unfolding the definitions. □

Proposition 5.4. *The following terms constitute a size-indexed functor* $(\Box_0, \Box_1)$*.*

$$\Box_0 : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U)$$
$$\Box_0\,A\,s := \forall r < s.\ A\,r$$
$$\Box_1 : \Pi_{(A,B:\mathsf{Size}\to U)}(A \xrightarrow{s} B) \to (\Box_0 A \xrightarrow{s} \Box_0 B)$$
$$\Box_{A,B}\,f\,s\,g\,r := f\,r\,(g\,r)$$

Proof. Here as well, the laws for identity and composition follow from unfolding the definitions. □

We will write $\Diamond_s A$ for $\Diamond\,A\,s := \exists r < s.\ A\,r$ and similarly $\Box_s A$ for $\Box\,A\,s := \forall r < s.\ A\,r$. Given a functor F on the category of small types, we can make use of $\Diamond$ and $\Box$ to "lift" F to the category of size-indexed types as follows.

Proposition 5.5. *Let* F *be a functor on the category of small types. Then the following terms constitute a size-indexed functor.*

$$F[\Diamond -]_0 : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U)$$
$$F[\Diamond\,A]_0\,s := F(\Diamond_s A) = F(\exists r < s.\ A\,r)$$
$$F[\Diamond -]_1 : \Pi_{(A,B:\mathsf{Size}\to U)}(A \xrightarrow{s} B) \to (F[\Diamond\,A] \xrightarrow{s} F[\Diamond\,B])$$
$$F[\Diamond\,f]_{A,B}\,s := F_{\Diamond_s A, \Diamond_s B}(\Diamond_{A,B}\,f\,s)$$

Proof. The laws for identity and composition follow from unfolding the definitions, using the fact that F and $\Diamond$ are well-defined functors. □

Given a functor F on the category of small types describing the shape of an inductive type, the size-indexed functor $F[\Diamond -]$ should describe the shape of the size-indexed variant of the type. In our previous example, we considered the functor $F : U \to U$ where $F\,X = \top + A \times X$ for some fixed $A : U$, whose initial algebra corresponds to the inductive definition for lists over A. The functor $F[\Diamond -]$ is then defined on objects as

$$\begin{aligned} F[\Diamond X]\,s &:= F(\exists r < s.\ X\,r) \\ &:= \top + A \times \exists r < s. X\,r \end{aligned}$$

which corresponds to the informal intuition for size-indexed lists described in the previous section. We have an analogous construction for bounded universal quantification.

Proposition 5.6. *Let F be a functor on the category of small types. Then the following terms constitute a size-indexed functor.*

$$\begin{aligned} &F[\Box -]_0 : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U) \\ &F[\Box\,A]_0\,s := F(\Box_s\,A) = F(\forall r < s.\ A\,r) \\ &F[\Box -]_1 : \Pi_{A,B:\mathsf{Size}\to U}.\ (A \xrightarrow{s} B) \to (F[\Box\,A] \xrightarrow{s} F[\Box\,B]) \\ &F[\Box\,f]_{A,B}\,s := F_{\Box_s A, \Box_s B}(\Box_{A,B}\,f\,s) \end{aligned}$$

Proof. The required properties follow from the fact that F and $\Diamond$ are well-defined functors. □

Dually, the functor $F[\Box -]$ should lift a functor F describing the shape of a coinductive type to that of the size-indexed variant. For example, the functor for streams over A is defined by $F\,X = A \times X$, and thus we have the functor $F[\Box X] = F(\forall r < s.\ X\,r) = A \times \forall r < s.\ X\,r$, describing the shape of size-indexed streams.

5.4 Size-indexed Algebras

We now require definitions for algebras over functors, in order the define a notion of initial algebra.

Definition 5.7 (F-Algebra)**.** *Given a functor F on the category of small types, an* F-algebra *is a pair (A, k_A) consisting of a carrier $A : U$ together with an algebra map $k_A : FA \to A$. Thus, explicitly an F-algebra is a term of the type*

$$F\text{-}\mathsf{Alg} := \Sigma(A : U)(FC \to C)$$

Given F-algebras (A, k_A) and (B, k_B), a morphism $h : (A, k_A) \to (B, k_B)$ is a term $h : A \to B$ such that h satisfies the required equality propositionally:

$$p_h : k_B \circ Fh = h \circ k_A$$

Due to the inclusion of UIP, the proof term for the equality does not need to be part of the data.

For an algebra (A, k_A), there is an obvious identity map using id_A. Given morphisms $f : (A, k_A) \to (B, k_B)$ and $g : (B, k_B) \to (C, k_C)$, their composition is given by $g \circ f$, where we have the following propositional equalities:

$$\begin{aligned} k_C \circ F(g \circ f) &= k_C \circ Fg \circ Ff \\ &= g \circ k_B \circ Ff \\ &= (g \circ f) \circ k_A \end{aligned}$$

We will denote the category of F-algebras by F-Alg.

An initial algebra is then the initial object in the category of F-algebras, which we will denote by $(\mu F, \mathsf{in})$. Concretely, this means that for any algebra (A, k_A) there exists a term

$$\mathsf{fold}\, A\, k_A : \mu F \to A$$

such that $\mathsf{fold}\, A\, k_A$ is an algebra morphism and is moreover unique up to propositional equality.

Definition 5.8 (Size-indexed algebra)**.** *Given a functor F on the category of small types, a* size-indexed F-algebra *is defined to be an algebra for the size-indexed functor $F[\Diamond -]$. Explicitly, it consists of a pair (A, k_A) where $A : \mathsf{Size} \to U$ and $k_A : F[\Diamond A] \xrightarrow{s} A$. A morphism $h : (A, k_A) \to (B, k_B)$ is a term $h : A \xrightarrow{s} B$ such that h satisfies the required equality propositionally:*

$$p_h : k_B \circ F_{A,B}\, h = h \circ k_A$$

Note that here composition is defined as in the category of size-indexed types. This gives rise to a category of size-indexed F-algebras, which we denote by $F^{\Diamond}$-Alg.

For any F-algebra on the category of small types, we can create a size-indexed F-algebra which is 'constant' in the sense that it ignores the size index.

Proposition 5.9. *Every F-algebra (A, k_A) gives rise to a $F^{\Diamond}$-algebra (TA, k_{TA}) where the first component is defined as $TA := \lambda i.A : \mathsf{Size} \to U$. Moreover, the operation T is functorial.*

Proof. Given the F-algebra (A, k_A), we define $TA : \mathsf{Size} \to U$ by $TA := \lambda i.A$. To define the morphism $k_{TA} : F[\Diamond A] \to A$, we first define the following auxiliary term, noting the fact that $i \notin \mathrm{fv}(A)$:

$$\begin{aligned}&\mathsf{extract} : \Pi_{(A:U)} \Diamond TA \xrightarrow{s} TA\\ &\mathsf{extract}_A\, s\, \langle r, a\rangle := a\end{aligned}$$

This allows us to define $k_{TA}\, i := k_A \circ F(\mathsf{extract}_{\mathsf{A}}\, i)$, which is well-typed as $k_A : FA \to A$ and $F(\mathsf{extract}_{\mathsf{A}}\, i) : F(\exists j < i.\ TA\, i) \to F(TA\, i) \equiv F(\exists j < i.\ A) \to FA$. We can also write this as: $k_{TA} := (\lambda i.k_A) \circ F[\mathsf{extract}_A]$ using the definition of composition for size-indexed types

Given two F-algebras (A, k_A) and (B, k_B) and an algebra morphism $f : (A, k_A) \to (B, k_B)$, we define $Tf : TA \xrightarrow{s} TB$ by

$$\begin{aligned}&Tf : TA \xrightarrow{s} TB\\ &Tf := \lambda i.f\end{aligned}$$

By unfolding the definitions and making use of function extensionality, we can now see that $\mathsf{extract}$ is natural in A, meaning that the following square commutes for any $f : A \to B$:

$$\begin{array}{ccc} \Diamond TA & \xrightarrow{\mathsf{extract}_A} & TA \\ {\scriptstyle \Diamond Tf}\downarrow & & \downarrow{\scriptstyle Tf} \\ \Diamond TB & \xrightarrow[\mathsf{extract}_B]{} & TB \end{array}$$

It remains to show that Tf is an algebra morphism, i.e we require the following identity

$$Tf \circ k_{TA} = k_{TB} \circ F[\Diamond Tf]$$

This follows from unfolding the definitions, together with the observation that $\mathsf{extract}$ is natural in A. We make use of function extensionality. Explicitly, we have the following, making us of the

propositional equality $f \circ k_A = k_B \circ Ff$:

$$\begin{aligned}(Tf \circ k_{TA})\, s &\equiv Tf\, s\, (k_A \circ F(\mathsf{extract}_A\, s)) \\ &\equiv (f \circ k_A) \circ F(\mathsf{extract}_A\, s) \\ &= (k_B \circ Ff) \circ F(\mathsf{extract}_A\, s) \\ &= (k_B \circ F(f \circ \mathsf{extract}_A\, s)) \\ &= k_B \circ F(\mathsf{extract}_B\, s) \circ (F[\Diamond Tf]s) \\ &\equiv (k_{TB} \circ F[\Diamond Tf])\, s\end{aligned}$$

Thus, $T : F\text{-}\mathsf{Alg} \to F^{\Diamond}\text{-}\mathsf{Alg}$ is a functor.

□

We will now show that we can define initial size-indexed F-algebras in the calculus using the fixpoint operator. This will then allow us to derive initial algebras for a class of functors on the category of small types.

Lemma 5.10. *Let F be a functor on the category of small types and $F[\Diamond -]$ be the corresponding size-indexed functor. Then $F[\Diamond -]$ has an initial algebra with first component $\mu^{\Diamond}F : \mathsf{Size} \to U$ where:*

$$\mu^{\Diamond}F\, s := \mathsf{fix}\, (\lambda i.\ \lambda X.\ F(\Diamond_i\, X))\, s$$

Proof. We first note that the propositional β-rule for the fixpoint operator gives us the following chain of equalities:

$$\begin{aligned}\mu^{\Diamond}F\, s &:= \mathsf{fix}\, (\lambda i.\ \lambda X.\ F(\Diamond_i\, X))\, s \\ &= (\lambda i.\ \lambda X.\ F(\Diamond_i\, X))\, s\, (\lambda r < s.\ \mathsf{fix}\, (\dots)\, r) \\ &= F(\Diamond_s\, \mathsf{fix}\, (\lambda i.\ \lambda X.\ F(\Diamond_i\, X))) \\ &=: F[\Diamond(\mu^{\Diamond}F)]\, s\end{aligned}$$

Thus, we can construct a term $p : \forall i.\mu^{\Diamond}F\, i = F[\Diamond(\mu^{\Diamond}F)]\, i$ of pointwise propositional equality, which allows us to define the (invertible) algebra map $\mathsf{in} : F[\Diamond(\mu^{\Diamond}F)] \xrightarrow{s} \mu^{\Diamond}F$ using the transport operation pointwise, which we defined in Section 2.1.2:

$$\mathsf{in}\, s := \mathsf{tr}((F[\Diamond(\mu^{\Diamond}F)]\, s), (\mu^{\Diamond}F\, s), p\, s)$$

We denote its pointwise inverse by $\mathsf{out} : \mu^{\Diamond}F \xrightarrow{s} F[\Diamond(\mu^{\Diamond}F)]$. For any $m : \mu^{\Diamond}F$ and $s : \mathsf{Size}$ we have $(\mathsf{out}\, s\, (\mathsf{in}\, s\, m))$. Now we require a definition for the term

$$\mathsf{fold}^{\Diamond} : \Pi_{(A:\mathsf{Size}\to U)}(F[\Diamond A] \xrightarrow{s} A) \to (\mu^{\Diamond}F \xrightarrow{s} A)$$

We define this term using fix:

$$\begin{aligned}&\mathsf{fold}^{\Diamond}\, A\, k_A := \mathsf{fix}\, f \\ &f : \forall i.\ (\forall j < i.\ \mu^{\Diamond}F\, j \to A\, j) \to (\mu^{\Diamond}F\, i \to A\, i) \\ &f\, s\, \mathsf{fold}'\, m := k_A\, s\, (F_{\Diamond_s(\mu^{\Diamond}F),\Diamond_s A}\, (\lambda\, \langle r, m'\rangle.\ \langle r, \mathsf{fold}'\, r\, m'\rangle)\, (\mathsf{out}\, s\, m))\end{aligned}$$

Note here that we have:

$$F_{\Diamond_s(\mu^{\Diamond}F),\Diamond_s A} : (\Diamond_s(\mu^{\Diamond}F) \to \Diamond_s A) \to (F[\Diamond(\mu^{\Diamond}F)]\, s \to F[\Diamond A]\, s)$$

Thus, the definition is well-typed, noting that $m : \mu^{\Diamond}F\, s$ and that we have the propositional equality $\mu^{\Diamond}F\, s = F[\Diamond(\mu^{\Diamond}F)]\, s$, allowing us to transport m, such that $\mathsf{out}\, s\, m : F[\Diamond(\mu^{\Diamond}F)]\, s$.

It remains to show that $\mathsf{fold}^\Diamond$ is indeed a morphism of algebras, so we need to show the following equality holds for $A : U$ and $k_A : F[\Diamond A] \xrightarrow{s} A$:

$$k_A \circ F[\Diamond\,(\mathsf{fold}^\Diamond\, A\, k_A)] = \mathsf{fold}^\Diamond\, A\, k_A \circ \mathsf{in}$$

This is verified by unfolding the definitions and the unfolding of fix. Given $s : \mathsf{Size}$ and $m : F[\Diamond(\mu^\Diamond F)]\, s$, and letting f denote the term given above in the definition of $\mathsf{fold}^\Diamond$, we have:

$$\begin{aligned}
((\mathsf{fold}^\Diamond\, A\, k_A) \circ \mathsf{in})\, s\, m &:= (\mathsf{fold}^\Diamond\, A\, k_A)\, s\, (\mathsf{in}\, s\, m) \\
&\equiv \mathsf{fix}\, f\, s\, (\mathsf{in}\, s\, m) \\
&= f\, s\, (\lambda r < s.\ \mathsf{fix}\, f\, r)\, (\mathsf{in}\, s\, m) && (\beta\text{-rule for } \mathsf{fix}) \\
&\equiv k_A\, s\, (F(\lambda\langle r, m'\rangle.\ \langle r, \mathsf{fix}\, f\, r\, m'\rangle)\, (\mathsf{out}\, s\, (\mathsf{in}\, s\, m))) && (\text{def. of } f) \\
&= k_A\, s\, (F(\lambda\langle r, m'\rangle.\ \langle r, \mathsf{fix}\, f\, r\, m'\rangle)\, m) && (\mathsf{out} \text{ inverse } \mathsf{in}) \\
&\equiv k_A\, s\, (F(\lambda\langle r, m'\rangle.\ \langle r, (\mathsf{fold}^\Diamond\, A\, k_A)\, r\, m'\rangle)\, m) \\
&\equiv k_A\, s\, (F[\Diamond_s\, (\mathsf{fold}^\Diamond\, A\, k_A)]\, m) && (\text{def. of } F[\Diamond -]) \\
&\equiv (k_A \circ F[\Diamond\, (\mathsf{fold}^\Diamond\, A\, k_A)])\, s\, m
\end{aligned}$$

Thus, by function extensionality we get the desired equality. Finally, we require the uniqueness condition for $\mathsf{fold}\, A\, k_A$ up to propositional equality. This follows from the uniqueness condition for fix. Suppose we have an algebra morphism $h : (\mu F, \mathsf{in}) \to (A, k_A)$, where $h : \mu^\Diamond F \xrightarrow{s} A$ is such that $k_A \circ F[\Diamond h] = h \circ \mathsf{in}$. Then we recall Proposition 2.1, which tells us that $\mathsf{fold}\, A\, k_A := \mathsf{fix}\, f$ is the unique fixpoint of f up to propositional equality. By this proposition it suffices for us to show that the following equality holds for every $s : \mathsf{Size}$:

$$h\, s = f\, s\, (\lambda r < s.\ h\, r)$$

Note that since $k_A \circ F[\Diamond h] = h \circ \mathsf{in}$ and $\mathsf{in} \circ \mathsf{out} = \mathsf{id}$, we get that $h = k_A \circ F[\Diamond h] \circ \mathsf{out}$. Also, for $m : \mu^\Diamond F\, s$, we have:

$$\begin{aligned}
f\, s\, (\lambda r < s.\ h\, r)\, m &\equiv k_A\, s\, (F(\lambda\langle r, m'\rangle.\ \langle r, h\, r\, m\rangle)\, (\mathsf{out}\, s\, m)) \\
&= k_A\, s\, (F[\Diamond_s h]\, (\mathsf{out}\, s\, m)) \\
&= (k_A \circ F[\Diamond h] \circ \mathsf{out})\, s\, m \\
&= h\, s\, m
\end{aligned}$$

Thus, by function extensionality we get $h\, s = f\, s\, (\lambda r < s.\ h\, r)$ and so $h = \mathsf{fix}\, f = \mathsf{fold}\, A\, k_A$ by Proposition 2.1. □

Thus, all size-indexed functors have initial algebras, which are defined using the fixpoint operator. This allows us to encode size-indexed inductive types. Now we would like to work towards defining initial algebras of functors by existential quantification over the corresponding size-indexed type. We first note that the existential type constructor gives rise to a functor.

Proposition 5.11. *The existential type constructor gives rise to a functor from the category of size-indexed types to the category of small types.*

Proof. This definition is similar to that of bounded existential quantification. We define the operations on morphisms as follows:

$$\begin{aligned}
&\exists_{A,B:\mathsf{Size}\to U} : (A \xrightarrow{s} B) \to (\exists i.A\, i \to \exists i.B\, i) \\
&\exists_{A,B}\, f\, \langle i, a\rangle := \langle i, f\, i\, a\rangle
\end{aligned}$$

Using function extensionality, it is easy to check that this satisfies the required propositional equalities. □

To construct the initial algebra of a functor by existential quantification, we require the functor to satisfy the following additional property.

Definition 5.12. *Let F be a functor on the category of small types. We say that F* weakly commutes *with the existential type if the following canonical map:*

$$\begin{aligned}&\mathsf{can}_A : \exists i.\ F[\Diamond A]\, i \to F(\exists i.A\, i)\\ &\mathsf{can}_A\langle i, a'\rangle := F\,(\varphi_A\, i)\, a'\end{aligned}$$

is an equivalence for any $A : \mathsf{Size} \to U$, where φ is defined as

$$\begin{aligned}&\varphi_A : \forall i.\ (\exists j < i.\ A\, j \to \exists j.\ A\, j)\\ &\varphi_A\, i\, \langle j, a\rangle := \langle j, a\rangle\end{aligned}$$

Note that for any $i : \mathsf{Size}$, the term $\varphi_A\, i : \Diamond_i A \to \exists j.\ A\, j$ is natural in A, meaning that the following diagram commutes for any $f : A \xrightarrow{s} B$:

$$\begin{array}{ccc} \Diamond_i\, A & \xrightarrow{\Diamond f\, i} & \Diamond_i\, B \\ \downarrow{\scriptstyle \varphi_A\, i} & & \downarrow{\scriptstyle \varphi_B\, i} \\ \exists\, A & \xrightarrow{\exists f} & \exists\, B \end{array}$$

For functors that satisfy this property, we can define a functor $\exists : F^{\Diamond}\text{-}\mathsf{Alg} \to F\text{-}\mathsf{Alg}$ by lifting the functor we previously defined to algebras.

Proposition 5.13. *Given a functor F on the category of small types that weakly commutes with the existential quantifier, every size-indexed F-algebra (A, k_A) gives rise to a type $\exists i.A\, i$ which is the first component of an F-algebra. Moreover, this operation is functorial.*

Proof. Let $\mathsf{can}_A^{-1} : F(\exists i.A\, i) \to (\exists i.\ F[\Diamond A]\, i)$ be the equivalence obtained from F weakly commuting with the existential type at A. Note that here we have $k_A : F[\Diamond A] \xrightarrow{s} A$. We can define the F-algebra $(\exists i.A\, i, k_{\exists A})$, where $k_{\exists A} : F(\exists i.A\, i) \to \exists i.A\, i$ is defined by

$$k_{\exists A} := \exists\, k_A \circ \mathsf{can}_A^{-1}$$

Given an algebra morphism $f : (A, k_A) \to (B, k_B)$, we define an algebra morphism $(\exists i.A\, i, k_{\exists A}) \to (\exists i.B\, i, k_{\exists B})$ using the existential type on maps as $\exists f : \exists i.A\, i \to \exists i.B\, i$. To show that this constitutes an algebra morphism, we need to show that the following holds:

$$\exists f \circ k_{\exists A} = k_{\exists B} \circ F(\exists f)$$

First, we show that following diagram commutes, where we have bidirectional arrows to denote that can is an equivalence:

$$\begin{array}{ccc} \exists i.\ F[\Diamond A]\, i & \xleftrightarrow{\mathsf{can}_A} & F(\exists i.A\, i) \\ \downarrow{\scriptstyle \exists(F[\Diamond f])} & & \downarrow{\scriptstyle F(\exists f)} \\ \exists i.\ F[\Diamond B]\, i & \xleftrightarrow[\mathsf{can}_B]{} & F(\exists i.B\, i) \end{array}$$

To show this, it suffices to show that it commutes in one direction, since can is a family of equivalences. Thus, we have the following derivation for $\langle i, a'\rangle : \exists i.\ F[\Diamond A]\, i$:

$$\begin{aligned}(\mathsf{can}_B \circ \exists(F[\Diamond f]))\, \langle i, a'\rangle &= \mathsf{can}_B\langle i, F(\Diamond f\, i)\, a'\rangle \\ &= F(\varphi_B\, i)\, (F(\Diamond f\, i)\, a') \\ &= F(\varphi_B\, i \circ \Diamond f\, i)\, a' \\ &= F(\exists f \circ \varphi_A\, i)\, a' && \text{(naturality of } \varphi_B\, i) \\ &= (F(\exists f) \circ \mathsf{can}_A)\, \langle i, a'\rangle\end{aligned}$$

This gives us a form of naturality for can. Now the fact that $\exists f$ is an algebra morphism follows from the identity $f \circ k_A = k_B \circ F[\Diamond f]$ together with the naturality of can:

$$\begin{aligned}
\exists f \circ k_{\exists A} &:= \exists f \circ \exists k_A \circ \mathsf{can}_A^{-1} \\
&= \exists (f \circ k_A) \circ \mathsf{can}_A^{-1} \\
&= \exists (k_B \circ F[\Diamond f]) \circ \mathsf{can}_A^{-1} && (f \text{ is a morphism}) \\
&= \exists k_B \circ (\exists F[\Diamond f] \circ \mathsf{can}_A^{-1}) \\
&= \exists k_B \circ \mathsf{can}_B^{-1} \circ F(\exists f) && (\text{naturality of } \mathsf{can}) \\
&=: k_{\exists B} \circ F(\exists F)
\end{aligned}$$

□

Finally, we are able to prove our desired result, namely that for functors that weakly commute with the existential type, their initial algebra is obtained by existential quantification over the initial size-indexed algebra.

Theorem 5.14. *Let F be a functor on the category of small types that weakly commutes with the existential type. Then F has an initial algebra with first component:*

$$\mu F := \exists i.(\mu^{\Diamond} F)\, i$$

Proof. We show that $\exists$ is left adjoint to T:

$$F^{\Diamond}\text{-}\mathsf{Alg} \underset{T}{\overset{\exists}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} F\text{-}\mathsf{Alg}$$

In fact, this corresponds to the currying isomorphism. Suppose we have an $F^{\Diamond}$-algebra (A, k_A) and an F-algebra (B, k_B). Let $f : (\exists i.A\, i, k_{\exists A}) \to (B, k_B)$ be a morphism of F-algebras, then we define

$$\begin{aligned}
&f^\star : A \xrightarrow{s} TB \\
&f^\star\, s\, a := f\, \langle s, a \rangle
\end{aligned}$$

We need to show that $f^\star$ is an $F^{\Diamond}$-algebra morphism $f^\star : (A, k_A) \to (TB, k_{TB})$. Since f is an F-algebra morphism, we have the following commuting diagram, where we expand $k_{\exists A} := \exists k_A \circ \mathsf{can}_A^{-1}$:

$$\begin{array}{ccc}
F(\exists i.A\, i) & \xrightarrow{Ff} & FB \\
{\scriptstyle \mathsf{can}_A}\uparrow\downarrow & & \\
\exists i.\, F[\Diamond A]\, i & & \Big\downarrow {\scriptstyle k_B} \\
{\scriptstyle \exists k_A}\Big\downarrow & & \\
\exists i.\, A\, i & \xrightarrow[f]{} & B
\end{array}$$

We want to show that the following diagram commutes:

$$\begin{array}{ccc}
F[\Diamond A] & \xrightarrow{F[\Diamond f^\star]} & F[\Diamond TB] \\
 & & \Big\downarrow {\scriptstyle F[\mathsf{extract}_B]} \\
\Big\downarrow {\scriptstyle k_A} & & F[TB] \\
 & & \Big\downarrow {\scriptstyle \lambda i.k_B} \\
A & \xrightarrow[f^\star]{} & TB
\end{array}$$

where we have expanded $k_{TB} := (\lambda i.\ k_B) \circ F[\mathsf{extract}_B]$. First, we note that given $i : \mathsf{Size}$ and $a' : F[\Diamond A]\, i$, we have

$$(f^\star \circ k_A)\, i\, a' = (f \circ \exists k_A)\, \langle i, a' \rangle$$

which follows by simple unfolding of their definitions. This is the (bottom) left-hand part of both diagrams. Since the first diagram commutes, it now suffices to show that the same holds for the (top) right-hand part of the diagrams, so we need to show:

$$(k_B \circ Ff \circ \mathsf{can}_A)\, \langle i, a' \rangle = ((\lambda i.k_B) \circ F[\mathsf{extract}_B] \circ F[\Diamond f^\star])\, i\, a'$$

For this, we need to make use the following equality for any $i : \mathsf{Size}$, which also follows by unfolding the definitions and making use function extensionality:

$$f \circ (\varphi_A\, i) = \mathsf{extract}_B\, i \circ \Diamond f^\star\, i$$

Now we can unfold the definitions and use the previous equality to show that the diagram commutes

$$\begin{aligned} k_B\, (Ff\, (\mathsf{can}_A\, \langle i, a' \rangle)) &= k_B\, (Ff\, (F(\varphi_A\, i)\, a')) \\ &= k_B\, (F(f \circ \varphi_A\, i)\, a') \\ &= k_B\, (F((\mathsf{extract}_B\, i) \circ \Diamond(f^\star\, i))\, a') \\ &= ((\lambda i.k_B) \circ F[\mathsf{extract}_B] \circ F[\Diamond f^\star])\, i\, a \end{aligned}$$

This shows that $f^\star : (A, k_A) \to (TB, k_{TB})$ is an algebra morphism.

Conversely, if $g : (A, k_A) \to (TB, k_{TB})$ is a morphism of $F^\Diamond$-algebras, then we define

$$\begin{aligned} &g_\star : \exists i.A\, i \to B \\ &g_\star\, \langle i, a \rangle := g\, i\, a \end{aligned}$$

Note first that it is clear that these operations $(-)^\star$ and $(-)_\star$ are mutual inverses. We need to show that $g_\star$ is an algebra morphism from $(\exists i.A\, i, k_{\exists A})$ to (B, k_B). Now we may suppose that the second diagram above commutes (replacing $f^\star$ by g) and we need to show that the first diagram commutes (replacing f by $g_\star$). This follows from analogous reasoning to the first case: we can see by unfolding that the following holds for any $i : \mathsf{Size}$ and $a' : F[\Diamond A]\, i$:

$$(g_\star \circ \exists k_A)\, \langle i, a' \rangle = (g \circ k_A)\, i\, a'$$

Then we get the following equality, also by function extensionality:

$$g_\star \circ (\varphi_A\, i) = \mathsf{extract}_B\, i \circ \Diamond g\, i$$

which allows us to show that the first diagram commutes. Thus, we have an algebra morphism $g_\star : (\exists i.A\, i, k_{\exists A}) \to (B, k_B)$.

This gives the adjunction pictured above. Finally, since left adjoints preserve initial objects and $\mu^\Diamond F$ is the first component of the initial $F^\Diamond$-algebra, we may conclude that $\exists i.(\mu^\Diamond F)\, i$ is the first component to the initial F-algebra. □

5.5 Size-indexed Coalgebras

We can dualise the results for initial algebras of functors weakly commuting with the existential quantifier to terminal coalgebras of functors weakly commuting with the universal quantifier. The structure of the results in this section follows that of the previous section closely, so we will be somewhat more concise.

Definition 5.15 (Functor coalgebra)**.** *Given a functor F on the category of small types, an* F-coalgebra *is a pair of terms (C, k_C) where $C : U$ and $k_C : C \to FC$. Given F-coalgebras (C, k_C) and (C, k_C), a morphism $h : (C, k_C) \to (D, k_D)$ is a term $h : C \to D$ such that h satisfies the required propositional equality:*

$$p_h : k_D \circ h = F_{C,D}\, h \circ k_C$$

As with F-algebras, identities and composition can be defined, so these definitions give rise to a category F-Coalg.

A terminal colagebra is the terminal object in the category F-Coalg, which we will denote as $(\nu F, \mathsf{out})$, if it exists. Thus, we require a term

$$\mathsf{unfold} : \Pi_{(C:U)}(C \to FC) \to (C \to \nu F)$$

such that $\mathsf{unfold}\, C\, k_C$ is a unique morphism from (C, k_C) to $(\nu F, \mathsf{out})$ up to propositional equality.

Definition 5.16 (Size-indexed coalgebra)**.** *Given a functor F on the category of small types, a* size-indexed F-coalgebra *is defined to be an $F[\Box -]$-coalgebra. Explicitly, it consists of a pair (C, k_C) where $C : \mathsf{Size} \to U$ and $k_C : C \xrightarrow{s} F[\Box C]$. This gives rise to a category of size-indexed F-coalgebras, which we denote by $F^{\Box}$-Coalg.*

For any F-coalgebra on the category of small types, we can create a size-indexed F-coalgebra which is 'constant' in the sense that it ignores the size index.

Proposition 5.17. *Every F-coalgebra (C, k_C) gives rise to a $F^{\Box}$-coalgebra (TC, k_{TC}) with first component $TC := \lambda i.C$. Moreover, the operation T is functorial.*

Proof. This follows the same structure as for the case of F-algebras. Given $C : U$, define $TC : \mathsf{Size} \to U$ by $TC = \lambda s.C$. To define the morphism $k_{TC} : TC \xrightarrow{s} F(\Box_s C)$, we first define an auxiliary term:

$$\mathsf{insert} : \Pi_{(C:U)} TC \xrightarrow{s} \Box_s TC$$
$$\mathsf{insert}_C\, i\, c := \lambda j < i.\ c$$

This allows us to define $k_{TC}\, i := F(\mathsf{insert}_C\, i) \circ k_C$, which is well-typed. We can also write this as $k_{TC} := F[\mathsf{insert}_C] \circ (\lambda i.\ k_C)$ using composition for size-indexed types.

Given a coalgebra morphism $f : (C, k_C) \to (D, k_D)$, we define $Tf : (TC, k_{TC}) \to (TD, k_{TD})$ by

$$Tf : TC \xrightarrow{s} TD$$
$$Tf := \lambda i.f$$

By unfolding the definitions, we can now see that insert is natural in C, meaning that the following square commutes for any $f : C \to D$, where $C, D : U$:

$$\begin{array}{ccc} TC & \xrightarrow{\mathsf{insert}_C} & \Box TC \\ {\scriptstyle Tf}\downarrow & & \downarrow{\scriptstyle \Box Tf} \\ TD & \xrightarrow[\mathsf{insert}_D]{} & \Box TD \end{array}$$

As in the algebra case, this is sufficient to show that Tf satisfies the required equality for a coalgebra morphism. Thus, T as defined here is a functor from F-Coalg to $F^{\Box}$-Coalg. □

As with algebras, we can obtain the terminal size-indexed coalgebra for a functor F using the fixpoint operator, as follows.

Lemma 5.18. *Let F be a functor on the category of small types and $F[\Box -]$ be the corresponding size-indexed functor. Then $F[\Box -]$ has a terminal coalgebra $\nu^{\Box} F : \mathsf{Size} \to U$ where:*

$$\nu^{\Box} F := \mathsf{fix}\,(\lambda i.\ \lambda X.\ F(\Box_i\, X))$$

Proof. As in the case of the initial algebra, the propositional β-rule for the fixpoint operator shows that $\nu^{\Box} F = F[\Box\, \nu^{\Box} F]$, so the coalgebra map $\mathsf{out} : \nu^{\Box} F \xrightarrow{s} F[\Box \nu^{\Box} F]$ may also be defined using the transport operation, and we denote its pointwise inverse by $\mathsf{in} : F[\Box \nu^{\Box} F] \xrightarrow{s} \nu^{\Box} F$. The term

$$\mathsf{unfold} : \Pi_{(C:\mathsf{Size}\to U)}(C \xRightarrow{s} F[\Box C]) \to (C \xRightarrow{s} \nu^{\Box} F)$$

is then defined using fix as:

$$\begin{aligned}
&\mathsf{unfold}^{\Box}\, C\, k_C := \mathsf{fix}\, f \\
&\quad f : \forall i.\ (\forall j < i.\ C\, j \Rightarrow \nu^{\Box} F\, j) \to (C\, i \Rightarrow \nu^{\Box} F\, i) \\
&\quad f\, i\, \mathsf{unfold}'\, c := \mathsf{in}\, i\, (F_{\Box_i C, \Box_i (\nu^{\Box} F)} (\lambda c'. \lambda j < i.\ \mathsf{unfold}'\, j\, (c\, j))(k_C\, i\, c))
\end{aligned}$$

Note that we have the following type for F:

$$F_{\Box_i C, \Box_i (\nu^{\Box} F)} : (\Box_i C \Rightarrow \Box_i (\nu^{\Box} F)) \to (F(\Box C)\, i \Rightarrow F[\Box (\nu^{\Box} F)]\, i)$$

Thus, the fact that $\mathsf{in}\, i : F[\Box \nu^{\Box} F]\, i \to \nu^{\Box} F\, i$ ensures the correct typing of the term $\mathsf{unfold}^{\Box}$. As in the case of size-indexed algebras, the β-equality and uniqueness of fix suffice to show that $\mathsf{unfold}^{\Box}$ is a morphism of coalgebras and is unique. □

We have an analogous property of commuting with the $\forall$ quantifier, which we first show gives rise to a functor as well.

Proposition 5.19. *The $\forall$ type constructor gives rise to a functor from the category of size-indexed types to the category of small types.*

Proof. We define the operations on morphisms as follows:

$$\begin{aligned}
&\forall_{C,D:\mathsf{Size}\to U} : (C \xrightarrow{s} D) \to (\forall i. C\, i \to \forall i. D\, i) \\
&\forall_{C,D}\, f\, g := \lambda i. f\, i\, (g\, i)
\end{aligned}$$

Using function extensionality, it is easy to check that this satisfies the required propositional equalities. □

Definition 5.20. *Let F be a functor on the category of small types. We say that F* weakly commutes *with $\forall$ if the following canonical map*

$$\begin{aligned}
&\mathsf{can}_C : F(\forall i. C\, i) \to (\forall i.\ F[\Box C]\, i) \\
&\mathsf{can}_C\, c'\, i := F\, (\psi_C\, i)\, c'
\end{aligned}$$

is an equivalence for each $C : \mathsf{Size} \to U$, where we define ψ_C as:

$$\begin{aligned}
&\psi_C : \forall i.\ (\forall j.\ C\, j \to \forall j < i.\ C\, j) \\
&\psi_C\, i\, c'\, j := c'\, j
\end{aligned}$$

Note that for any $i : \mathsf{Size}$, the term $\psi_A\, i : \forall j. C\, j \to \forall j < i. C\, j$ is natural in C, meaning that the following diagram commutes for any $f : C \xrightarrow{s} D$:

$$\begin{array}{ccc}
\forall C & \xrightarrow{\forall f} & \forall D \\
\downarrow{\scriptstyle \psi_C\, i} & & \downarrow{\scriptstyle \psi_D\, i} \\
\Box_i C & \xrightarrow{\Box f\, i} & \Box_i D
\end{array}$$

For functors that satisfy this property, we can define a functor $\forall : F^{\Box}\text{-}\mathsf{Coalg} \to F\text{-}\mathsf{Coalg}$ extending the action of the universal quantifier on size-indexed types as follows.

Proposition 5.21. *Given a functor F on the category of small types that weakly commutes with $\forall$, every $F[\Box -]$-coalgebra (C, k_C) gives rise to a type $\forall i.A\, i$ which is the first component of an F-coalgebra. Moreover, this operation is functorial.*

Proof. Let $\mathsf{can}_C^{-1} : \forall i.\ F[\Box C]\, i \to F(\forall i.C\, i)$ be the isomorphism obtained from F weakly commuting with $\forall$. For an $F[\Box -]$-coalgebra (C, k_C), where $k_C : C \xrightarrow{s} F[\Box C]$, we obtain the F-coalgebra $(\forall i.\, C\, i\,, k_{\forall C})$ where we define

$$k_{\forall C} : \forall i.\, C\, i \to F(\forall i.\, C\, i)$$
$$k_{\forall C} := \mathsf{can}_C^{-1} \circ \forall k_C$$

Given an coalgebra morphism $f : (C, k_C) \to (D, k_D)$, we obtain the map $\forall f : \forall i.C\, i \to \forall i.D\, i$. To show that this constitutes a coalgebra morphism

$$\forall f : (\forall i.\, C\, i\,, k_{\forall C}) \to (\forall i.\, D\, i\,, k_{\forall D})$$

we need to show the following equality holds:

$$k_{\forall D} \circ \forall f = F(\forall f) \circ k_{\forall C}$$

As in the algebra case, we first show that the following diagram commutes, where the bidirectional arrows denote the fact that can is a family of equivalences:

$$\begin{array}{ccc} F(\forall i.C\, i) & \xleftrightarrow{\mathsf{can}_C} & \forall i.\ F[\Box C]\, i \\ \Big\downarrow{\scriptstyle F(\forall f)} & & \Big\downarrow{\scriptstyle \forall(F[\Box f])} \\ F(\forall i.D\, i) & \xleftrightarrow[\mathsf{can}_D]{} & \forall i.\ F[\Diamond D]\, i \end{array}$$

To show this, it suffices to show that it commutes in one direction, since can is a family of equivalences. Thus, we have the following derivation for $c' : F(\forall i.C\, i)$ and $i : \mathsf{Size}$:

$$\begin{aligned} (\forall(F[\Box f]) \circ \mathsf{can}_C)\, c' &= \forall(F[\Box f])(\lambda i.\ F(\psi_C\, i)\, c') \\ &= \lambda i.\ F(\Box f\, i)(F(\psi_C\, i)\, c') \\ &= \lambda i.\ F(\Box f\, i \circ \psi_C\, i)\, c' \\ &= \lambda i.\ F(\psi_D\, i \circ \forall f)\, c' && \text{(naturality of } \psi) \\ &= \lambda i.\ F(\psi_D\, i)(F(\forall f)\, c') \\ &= \mathsf{can}_D(F(\forall f)\, c') \end{aligned}$$

This gives us a form of naturality for can. Now the fact that $\forall f$ is an algebra morphism follows because we have $k_D \circ f = F[\Box f] \circ k_C$ together with the naturality of can:

$$\begin{aligned} k_{\forall D} \circ \forall f &:= \mathsf{can}_D^{-1} \circ \forall k_D \circ \forall f \\ &= \mathsf{can}_D^{-1} \circ \forall(k_D \circ f) \\ &= \mathsf{can}_D^{-1} \circ \forall F[\Box f] \circ \forall k_C \\ &= F(\forall f) \circ \mathsf{can}_C^{-1} \circ \forall k_C \\ &= F(\forall f) \circ k_{\forall C} \end{aligned}$$

□

Dually to the case for initial algebras, we are able to prove that for functors that weakly commute with $\forall$, their terminal coalgebra is obtained by universal quantification over the terminal size-indexed coalgebra.

Theorem 5.22. *Let F be a functor on the category of small types that weakly commutes with the universal quantifier. Then F has an terminal coalgebra with first component:*

$$\nu F := \forall i.(\nu^{\Box} F)\, i$$

Proof. We show that $\forall$ is right adjoint to T, as defined in this section:

$$F^{\Box}\text{-}\mathsf{Coalg} \underset{T}{\overset{\forall}{\underset{\longleftarrow}{\overset{\longrightarrow}{\top}}}} F\text{-}\mathsf{Coalg}$$

This is shown by swapping arguments, and follows a similar structure to the case for algebras. Suppose that we have an F-coalgebra (C, k_C) and an $F[\Box -]$-coalgebra (D, k_D), so $C : U$ and $D : \mathsf{Size} \to U$. Let $f : (C, k_C) \to (\forall i.D\, i, k_{\forall D})$ be a morphism of F-coalgebras. Then we define

$$f^\star : TC \xrightarrow{s} D$$
$$f^\star\, i\, c = f\, c\, i$$

The fact that f is a coalgebra morphism implies that the following diagram commutes, where we unfold the definition $k_{\forall D} = \mathsf{can}_D^{-1} \circ \forall k_D$:

$$\begin{array}{ccc} C & \xrightarrow{\quad f \quad} & \forall i.D\, i \\ & & \downarrow{\scriptstyle \forall k_D} \\ {\scriptstyle k_C}\downarrow & & \forall i.F[\Box D]\, i \\ & & \updownarrow{\scriptstyle \mathsf{can}_D} \\ FC & \xrightarrow[Ff]{} & F(\forall i.D\, i) \end{array}$$

To show that $f^\star$ is a coalgebra morphism, the following diagram should commute, where we unfold the definition $k_{TC} := F[\mathsf{insert}_C] \circ (\lambda i.\, k_C)$:

$$\begin{array}{ccc} TC & \xrightarrow{\quad f^\star \quad} & D \\ {\scriptstyle \lambda i.\, k_C}\downarrow & & \\ F[TC] & & \downarrow{\scriptstyle k_D} \\ {\scriptstyle F[\mathsf{insert}_C]}\downarrow & & \\ F[\Box TC] & \xrightarrow[F[\Box f^\star]]{} & F[\Box D] \end{array}$$

First, we note that for $i : \mathsf{Size}$ and $c : C$, we have the following, corresponding to the top-right part of both diagrams:

$$(k_D \circ f^\star)\, i\, c = (\forall k_D \circ f)\, c\, i$$

This follows from unfolding the definitions:

$$(k_D \circ f^\star)\, i\, c = k_D\, i\, (f^\star\, i\, c) = k_D\, i\, (f\, c\, i) = \forall k_D\, (f\, c)\, i = (\forall k_D \circ f)\, c\, i$$

For the left-hand side of the diagram, we need to show the following equality:

$$(F[\Box f^*] \circ F[\mathsf{insert}_C] \circ (\lambda i.\, k_C))\, i\, c = (\mathsf{can}_D \circ F f \circ k_C)\, c\, i$$

As in the case of algebras, we first make use of the following characterisation:

$$\psi_C\, i \circ f = \Box f^\star\, i \circ \mathsf{insert}_C\, i$$

which we show by function extensionality:

$$(\psi_C\, i \circ f)\, c\, j = (\psi_C\, i)\, (f\, c)\, j = f\, c\, j$$
$$(\Box f^\star\, i \circ \mathsf{insert}_C\, i)\, c\, j = (\Box f^\star\, i)\, (\mathsf{insert}_C\, i)\, c)\, j = f^\star\, j\, (\mathsf{insert}_C\, i\, c\, j) = f^\star\, j\, c = f\, c\, j$$

With this, we can prove the previous equality:

$$\begin{aligned}(F[\Box f^*] \circ F[\mathsf{insert}_C] \circ (\lambda i.\ k_C))\, i\, c &= (F[\Box f^* \circ \mathsf{insert}_C] \circ (\lambda i.\ k_C))\, i\, c \\ &= F(\Box f^*\, i \circ \mathsf{insert}_C\, i)(k_C\, c) \\ &= F(\psi_C\, i \circ f)\, (k_C\, c) \\ &= F(\psi_C\, i)(F f\, (k_C\, c)) \\ &= \mathsf{can}_D\, (F f\, (k_C\, c))\, i \\ &= (\mathsf{can}_D \circ F f \circ k_C)\, c\, i\end{aligned}$$

This shows that the diagram commutes, meaning that $f^\star : (TC, k_{TC}) \to (D, k_D)$ is a coalgebra morphism.

Conversely, if $g : (TC, k_{TC}) \xrightarrow{s} (D, k_D)$ is a morphism of $F^\Box$-coalgebras, then we can define

$$g_\star : C \to \forall i.\ D\, i$$
$$g_\star\, c\, i := g\, i\, c$$

Note first that it is clear that these operations $(-)^\star$ and $(-)_\star$ are mutual inverses. We need to show that $g_\star$ is an algebra morphism from (C, k_C) to $(\forall i.\ D\, i, k_{\forall D})$. Now we may suppose that the second diagram above commutes (replacing $f^\star$ by g) and we need to show that the first diagram commutes (replacing f by $g_\star$). This follows from analogous reasoning to the first case: we can see by unfolding that the following holds for any $i : \mathsf{Size}$ and $c : C$:

$$(\forall k_D \circ g_\star)\, c\, i = (k_D \circ g)\, i\, c$$

Then we get also have the following equality, by function extensionality:

$$\psi_C\, i \circ g_\star = \Box g\, i \circ \mathsf{insert}_C\, i$$

which allows us to show that the first diagram commutes. Thus, we have an algebra morphism

$$g_\star : (C, k_C) \to (\forall i.\ D\, i, k_{\forall D})$$

This gives the adjunction pictured above. Finally, since right adjoints preserve terminal objects and $\nu^\Box F$ is the first component of the initial $F^\Box$-algebra, we may conclude that $\forall i.(\nu^\Box F)\, i$ is the first component to the initial F-algebra. □

5.6 Polynomial Functors

In the previous sections, we have shown that we may obtain initial algebras via existential quantification for functors which weakly commute with the existential quantifier. Dually, we may obtain final coalgebras via universal quantification for functors which weakly commute with the universal quantifier. Thus, the question arises for which class of functors these properties holds. The key result that we obtain from the equivalences in our theory, is that all polynomial functors, also called containers [AAG05], over small types satisfy these properties. First, we define what such polynomial functors are in our theory.

Definition 5.23. *Let $A : U$ be a small type and $B : \mathrm{El}\, A \to U$ be a family of small types depending on A. The* polynomial functor *on the category of small types associated with the pair (A, B) is $P_{A,B} : U \to U$, defined as:*

$$P_{A,B}(X) := \Sigma^U(a : A)(B\, a \to X)$$
$$P_{A,B}(f : X \to Y) := \lambda(a, g).\ (a, g \circ f)$$

Note that W-types, i.e. well-founded trees, are encoded as initial algebras for polynomial functors. Dually, M-types, i.e. non-well-founded trees, are encoded as terminal coalgebras for polynomial functors. These are sufficient to represent strictly-positive (co)inductive types. In order to show that polynomial functors weakly commute with the existential type, we make use of the additional axioms added to the theory listed in Figure 2.13, which are justified by the realisability model we constructed. Indeed, the fact follows directly from the type equivalences given in Lemma 2.3 and Figure 2.13.

Lemma 5.24. *Let $A : U$ be a small type and $B : \mathrm{El}\, A \to U$ be a family of small types depending on A. Then the polynomial functor $P_{A,B}$ associated with (A, B) weakly commutes with the universal and the existential quantifier.*

Proof. Recall that $P_{A,B}$ is defined as:

$$P_{A,B}(X) := \Sigma^U(a : A)(B\, a \to X)$$
$$P_{A,B}(f : X \to Y) := \lambda(a, g).\ (a, g \circ f)$$

Suppose that we have a size-indexed type $\Gamma \vdash C : \mathsf{Size} \to U$. We may equivalently view C as a small type in the extended context $\Gamma, i : \mathsf{Size} \vdash C(i) : U$. Note that $i \notin \mathrm{fv}(A)$ and $i \notin \mathrm{fv}(B\, a)$ for all $a : A$. Then we have the following equivalences for the existential quantifier:

$$\begin{aligned} P_{A,B}(\exists i.A(i)) &:= \Sigma^U(a : A)(B\, a \to \exists i.A(i)) \\ &\simeq \Sigma^U(a : A)(\exists i.(B\, a \to \exists j < i.A(j))) \\ &\simeq \exists i.\Sigma(a : A)(B\, a \to \exists j < i.A(j)) \\ &=: \exists i.P_{A,B}[\Diamond A]\, i \end{aligned}$$

Similarly, for the universal quantifier, we have:

$$\begin{aligned} P_{A,B}(\forall i.A(i)) &:= \Sigma^U(a : A)(B\, a \to \forall i.A(i)) \\ &\simeq \Sigma^U(a : A)(B\, a \to \forall i.\forall j < i.A(j)) \\ &\simeq \Sigma^U(a : A)(\forall i.(B\, a \to \forall j < i.A(j))) \\ &\simeq \forall i.\Sigma(a : A)(B\, a \to \forall j < i.A(j)) \\ &=: \forall i.P_{A,B}[\Box A]\, i \end{aligned}$$

□

Another point of view is that our equivalences allow us to show the property of weakly commuting for all strictly positive types which are built from the constructors $\top, \bot, +, \times, \to$ with the restriction that the variables may only occur on the right side of the arrow [AAG05]. This follows from the fact that binary sums and products are special cases of dependent sums and products, and the fact that $\top$ and $\bot$ are not dependent on Size. Thus, any strictly positive type resulting from these constructors will satisfy the property of weakly commuting.

Finally, we have the following theorem, which follows immediately from our previous results and the fact that polynomial functors weakly commute with both quantifiers:

Theorem 5.25. *For any polynomial functor $P_{A,B}$, we can construct the initial algebra and terminal coalgebra for the size-indexed functors $P_{A,B}[\Diamond -]$ and $P_{A,B}[\Box -]$ respectively, which we can denote by $W^{\Diamond}_{A,B}$ and $M^{\Box}_{A,B}$. The carriers of the initial algebra and terminal coalgebra for $P_{A,B}$ are then defined by $W_{A,B} := \exists i.\ W^{\Diamond}_{A,B}\, i$ and $M_{A,B} := \forall i.\ M^{\Box}_{A,B}\, i$, respectively.*

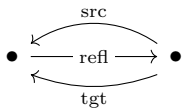
Chapter 6

Discussion

6.1 Assemblies and Reflexive Graphs

As mentioned in Chapter 1, reflexive graphs are another common approach to achieving parametricity or invariance results in models of type theory [AGJ14]. Recently, (cubical) reflexive graph models have been used for modelling parametric quantifiers as well as a simple type theory with sized types [Lim19].

We recall that reflexive graphs may be defined as presheaves over the following category:

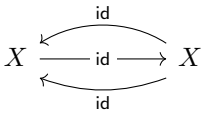


where $\mathrm{src} \circ \mathrm{refl} = \mathrm{tgt} \circ \mathrm{refl} = \mathrm{id}$. Spelling out this definition, a reflexive graph is a 5-tuple defined as follows:

$$\Gamma = (\Gamma_O, \Gamma_R, \Gamma_{\mathsf{refl}}, \Gamma_{\mathrm{src}}, \Gamma_{\mathrm{tgt}})$$
$$\Gamma_O, \Gamma_R \in \mathrm{Set}$$
$$\Gamma_{\mathsf{refl}} : \Gamma_O \to \Gamma_R$$
$$\Gamma_{\mathrm{src}}, \Gamma_{\mathrm{tgt}} : \Gamma_R \to \Gamma_O$$

such that $\Gamma_{\mathrm{src}} \circ \Gamma_{\mathsf{refl}} = \mathrm{id}_{\Gamma_O} = \Gamma_{\mathrm{tgt}} \circ \Gamma_{\mathsf{refl}}$. Intuitively, Γ_O is a set of objects and Γ_R a set of generalised relations on Γ_O; the function Γ_{refl} is then a proof of reflexivity for each object in Γ_O. Morphisms between reflexive graphs are defined via their characterisation as presheaves. We will denote the category of reflexive graphs by RG. Similarly to assemblies, reflexive graphs may be given a categories with families structure and serve as a model for dependent type theory.

Similarly to how we interpret small types by modest sets, or more specifically partial equivalence relations, in the model of dependent type theory with a universe given by Atkey et al. [AGJ14], types in the universe are interpreted by discrete, proof-irrelevant reflexive graphs. A reflexive graph is said to be *discrete* if the distinguished reflexive relation is isomorphic to the equality relation. In [AGJ14], this is defined as being isomorphic to a reflexive graph generated by a set X, i.e. a reflexive graph with the following structure:



A reflexive graph Γ is *proof-irrelevant* if the pairing

$$\langle \Gamma_{\mathrm{src}}(-), \Gamma_{\mathrm{tgt}}(-) \rangle : \Gamma_R \to \Gamma_O$$

is injective. Intuitively, this is the case if $\Gamma_R \subseteq \Gamma_O \times \Gamma_O$ up to isomorphism, meaning that any two objects may only be uniquely related.

To relate assemblies to reflexive graphs, we might define a functor $F : \mathsf{Asm} \to \mathsf{RG}$ as follows:

$$F(X, \Vdash_X) = (X, R_X \subseteq X \times X, x \mapsto (x, x), \pi_1, \pi_2)$$

defining the relation $R_X = \{(x_1, x_2) \mid \exists n \in \mathbb{N}.\ n \Vdash_X x_1, x_2\}$. On morphisms, F is defined as follows:

$$F((X, \Vdash_X) \xrightarrow{f} (Y, \Vdash_Y)) = (f, f_r)$$

where $f_r(x_1, x_2) = (f(x_1), f(x_2))$, which clearly constitutes a reflexive graph morphism. This functor forgets the realisers themselves, retaining only information on which elements share a realiser. For any modest set, this induces the equality relation, and thus a discrete, proof-irrelevant reflexive graph. In fact, for any non-modest assembly, we obtain a proof-irrelevant reflexive graph, though not necessarily discrete. Moreover, it is readily verified that F preserves products and sums, which are defined pointwise in both categories. However, F does not preserve exponentials in general.

Since F is forgetful in the sense that it forgets the realiser, we might attempt to define a different functor $G : \mathsf{Asm} \to \mathsf{RG}$ that retains the realisers:

$$G(X, \Vdash_X) = (X, E_X, r_X, \pi_1, \pi_2)$$

where we define $E_X = \{(x_1, x_2, n) \mid n \Vdash_X x_1, x_2\}$ and $r(x) = (x, x, n)$ for the *least* $n \in \mathbb{N}$ such that $n \Vdash_X x$. This results in a reflexive graph which need not be discrete or proof-irrelevant, even if $(X, \Vdash_X)$ is modest. However, it does not seem that G is canonically definable on morphisms between non-modest assemblies. We might think to define it as

$$G((X, \Vdash_X) \xrightarrow{f} (Y, \Vdash_Y)) = (f, f_r)$$

where $f_r(x_1, x_2, n) = (f(x_1), f(x_2), e \cdot n)$ where $e \in \mathbb{N}$ is such that $e \Vdash f$. However, to satisfy the requirement $r_Y(f(x)) = f_r(r_X(x))$, the code $e \in \mathbb{N}$ to be such that, for each $n \in \mathbb{N}$, $e \cdot n$ is defined and is the least code realising both $f(x_1)$ and $f(x_2)$. Such a realiser need not be partial computable in general. On the other hand, defining the morphism explicitly as $f_r(x_1, x_2, n) = (f(x_1), f(x_2), m)$, where m is the least realiser for $f(x_1)$ and $f(x_2)$ would violate functoriality of G.

The realisability aspect of assemblies make it challenging to form a clear picture of the connection between assemblies and reflexive graphs. Yet realisability models and reflexive graph models yield many of the same properties related to parametricity for type theories. One possible development that may relate realisability models to reflexive graphs is a generalisation of partial equivalence relations to allow for proof-relevance.

6.2 Sized Types and Guarded Type Theory

As briefly mentioned earlier, guarded type theory is an alternative type-based approach to ensuring productivity specifically, since it focuses on encoding coinductive types. This work brings the two approaches closer together, as we borrow several ideas from guarded type theory in our theory.

Guarded type theory also makes use of a fixpoint operator for defining recursive functions on guarded recursive types, similar to the one we define via well-founded induction on sizes. The unfolding rule for their fixpoint operator is given as a definitional equality in earlier work [Møg14; Biz+16]. Since this leads to definitional equality being undecidable, we have opted to add the β-rule for the fixpoint operator as a propositional equality instead. In more recent work on guarded type theory, its integration with cubical type theory [Coh+18], which gives a constructive interpretation of the univalence axiom, has been explored [Bir+16]. In this work, the β-rule for the fixpoint operator is provable within the calculus up to path equality, instead of definitional equality. The approach of sized types may similarly

benefit from integration with cubical type theory, to similarly define its fixpoint operator, and to define the ordering on sizes via propositional truncation.

In order to encode coinductive types using guarded recursive types, guarded type theory makes use of quantification over clocks, similarly to how we use universal quantification over sizes. In earlier work on guarded types [Møg14], similar type isomorphisms are added to the theory, describing how universal quantification over clocks commutes with type formers, in order to prove the property of polynomial functors commuting with universal quantification. The work on guarded type theory has presented their required type isomorphisms as a form of parametricity as well. However, guarded type theory has focused solely on encoding coinductive types via universal quantification over clocks, whereas our approach allows us to encode inductive types as well.

The relation between sized types and guarded types has been studied in a simply-typed setting, where it was shown by Veltri and Van der Weide [VW19] that sized types may be used to simulate guarded recursion in Agda when restricting to a single clock variable. The later modality $\blacktriangleright A$ is interpreted as bounded universal quantification over the type of sizes, matching the way we encode size-indexed coinductive types. They also raise the question of whether sized types can be modelled in the topos of trees, i.e. presheaves over the preorder on the natural numbers. In earlier work on modelling sized types via reflexive graphs, it was suggested that this may not be possible when using sizes for both inductive and coinductive types [Lim19]. One issue arises from the fact that size-indexed inductive types are naturally covariant, or monotone, in their size argument, whereas size-indexed coinductive types are naturally contravariant, or antitone, in their size argument. For example, a list of size at most r is of course a list of size at most s when $r < s$. Conversely, a stream which is defined up to at least s observations is also a stream defined up to at least r observations when $r < s$. Thus, only working with covariant or contravariant presheaves will necessarily restrict how we may use sizes. Perhaps only allowing sizes to work with coinductive types may be a suitable restriction for defining a model of sized types in the topos of trees.

6.3 Implementation of Sized Types

The type equivalences that we have added as axioms to the theory are key to encoding (co)inductive types in the theory. Thus, they should ideally be given a computational interpretation, i.e. be definable within the calculus. For all the equivalences to be definable in a calculus directly, it must come with some internal notion of parametricity. The current implementation of Agda does not have a notion of parametric quantification and thus does not validate the equivalences presented in this work.

The approach most closely mirroring the one here is that given by Nuyts et al. [NVD17; ND18]. In their first work, they develop a dependent type theory with parametric quantifiers. The relevant type equivalences are provable in their calculus, and they show how (co)inductive types might be encoded in the theory using sizes, although only for finitely-branching polynomials. They have developed an experimental extension of Agda with internal parametricity, and it might thus be interesting to see whether that combines well with the current implementation of sizes in Agda. However, their theory has not been shown to be strongly normalising.

Even with an internal notion of parametric quantification, the type equivalence stating that existential quantification (weakly) commutes with dependent products, which is required to encode infinitely-branching inductive types, will likely not be definable in the calculus. This is because it would require us to take limits of sizes indexed by small types, which would expose the ordinal nature of sizes and lead to the ordering on sizes being undecidable. Recent work on type-theoretic approaches to ordinals may also be of interest for modelling sizes, or to be used for termination directly [KNX23]. It would be interesting to study whether the inclusion of this equivalence as an axiom could allow for some form of reasoning using sized types without explicitly including a limit operator.

Chapter 7

Conclusion

In this work, we have studied a dependent type theory with sized types, which is a type-based approach to termination and productivity checking. In order to express that sizes should be treated parametrically, the theory axiomatises several equivalence, describing how quantification over sizes commutes with type constructors. We introduced the notions of partial equivalence relations and assemblies, in order to construct a model of the theory via the categories with families framework. The impredicativity of the realisability model allowed us to interpret the type of sizes as a larger set of ordinals, allowing us to justify the axioms. Finally, we showed how we may encode initial algebras and terminal coalgebras of a class of functors over small types via quantification over sizes. In particular, this class of functors includes all polynomial functors, allowing us to encode strictly positive inductive and coinductive types.

Previous work on sized types has mostly been syntactic, and some recent work has been done on models for sized types in reflexive graphs. By defining a realisability model, we offer a new perspective on the semantics of sized types, and improve on previous results on encoding (co)inductive types via quantification over sizes. It also exemplifies how realisability models allow for enforcing parametric behaviour in an impredicative theory, We have highlighted also some connections between sized types and guarded type theory, which could be explored further in future work in order to compare their expressiveness and to see whether sized types restricted to coinductive types admits a model in the topos of trees.

Although we work in an intensional setting, we assumed uniqueness of identity proofs for small types in the development of (co)algebras. Homotopy type theory rejects UIP, so important future work is to complete the same development without UIP. As mentioned, we are reasonably certain of the sketched development without UIP. In this work, we have limited the theory to work with a single universe of types. On the other hand, Agda has a hierarchy of universes and the type of sizes in a separate universe, and it allows for impredicative quantification over sizes for types in any of its universes. The realisability model in this work would not immediately support such a hierarchy, because for quantification over sizes for a type to be suitably parametric, the type should be interpreted as a small partial equivalence relation, which would only correspond to the lowest universe. Restricting sized types to work with only coinductive types might help admit a model for such a hierarchy.

Finally, the main practical avenue for future work is a consistent implementation of sized types in a language such as Agda, as the current implementation is inconsistent. In order for the axioms expressing parametricity of sizes to be definable, it is likely that some form of internal parametricity will need to be available in Agda as well. This would allow the closure ordinal, which causes inconsistencies in Agda, to be removed, as (co)inductive types may be encoded via quantification over their size-indexed variants. This would result in a robust method for termination- and productivity checking, which would greatly improve the expressivity and usability of total dependently-typed languages.

Bibliography

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theoretical Computer Science* 342.1 (2005). Applied Semantics: Selected Topics, pp. 3–27. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0304397505003373.

[Abe+13] Andreas Abel et al. "Copatterns: programming infinite structures by observations". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 27–38. ISBN: 9781450318327.

[Abe06] Andreas Abel. "Type-based termination: a polymorphic lambda-calculus with sized higher-order types". PhD thesis. Ludwig-Maximilians-Universität München, 2006.

[Abe12] Andreas Abel. "MiniAgda: Integrating Sized and Dependent Types". In: *PAR-10. Partiality and Recursion in Interactive Theorem Provers*. Ed. by Ekaterina Komendantskaya, Ana Bove, and Milad Niqui. Vol. 5. EPiC Series in Computing. EasyChair, 2012, pp. 18–33. DOI: 10.29007/322q.

[AG25] Carlo Angiuli and Daniel Gratzer. *Principles of dependent type theory*. 2025. URL: https://www.danielgratzer.com/papers/type-theory-book.pdf.

[AGJ14] Robert Atkey, Neil Ghani, and Patricia Johann. "A relationally parametric model of dependent type theory". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. 2014, pp. 503–515.

[AM13] Robert Atkey and Conor McBride. "Productive coprogramming with guarded recursion". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 197–208. ISSN: 0362-1340. DOI: 10.1145/2544174.2500597. URL: https://doi.org/10.1145/2544174.2500597.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. "Normalization by evaluation for sized dependent types". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017).

[Bir+12] Lars Birkedal et al. "First steps in synthetic guarded domain theory: step-indexing in the topos of trees". In: *Logical Methods in Computer Science* Volume 8, Issue 4, 1 (2012). ISSN: 1860-5974.

[Bir+16] Lars Birkedal et al. "Guarded Cubical Type Theory: Path Equality for Guarded Recursion". In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 23:1–23:17. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.23. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2016.23.

[Biz+16] Alevs Bizjak et al. "Guarded Dependent Type Theory with Coinductive Types". In: *Foundations of Software Science and Computation Structures*. Ed. by Bart Jacobs and Christof Löding. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 20–35. ISBN: 978-3-662-49630-5.

[CCD21] Simon Castellan, Pierre Clairambault, and Peter Dybjer. "Categories with Families: Unityped, Simply Typed, and Dependently Typed". In: *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*. Ed. by Claudia Casadio and Philip J. Scott. Cham: Springer International Publishing, 2021, pp. 135–180.

[Coh+18] Cyril Cohen et al. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9.

[Dan17] Nils Anders Danielsson. "Up-to techniques using sized types". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158131. URL: https://doi.org/10.1145/3158131.

[FPS20] Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. "Constructing Infinitary Quotient-Inductive Types". In: *Foundations of Software Science and Computation Structures*. Ed. by Jean Goubault-Larrecq and Barbara König. Cham: Springer International Publishing, 2020, pp. 257–276. ISBN: 978-3-030-45231-5.

[Gim95] Eduarde Giménez. "Codifying guarded definitions with recursive schemes". In: *Types for Proofs and Programs*. Ed. by Peter Dybjer, Bengt Nordström, and Jan Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 39–59.

[Hof97] Martin Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and P. Dybjer. Cambridge University Press, 1997, pp. 79–130.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. "Proving the correctness of reactive systems using sized types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 410–423. ISBN: 0897917693.

[JR97] Bart Jacobs and Jan J. M. M. Rutten. "A tutorial on (co)algebras and (co)induction". In: *Bulletin of The European Association for Theoretical Computer Science 62* (1997).

[KNX23] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. "Type-theoretic approaches to ordinals". In: *Theoretical Computer Science* 957 (2023), p. 113843. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2023.113843. URL: https://www.sciencedirect.com/science/article/pii/S0304397523001561.

[Ler24] Xavier Leroy. "Well-founded recursion done right". In: *CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages*. London, United Kingdom: ACM, 2024.

[Lim19] Jannis Limperg. "A Reflexive Graph Model of Sized Types". MA thesis. Chalmers University of Technology, 2019.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The size-change principle for program termination". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 81–92.

[LM91] Giuseppe Longo and Eugenio Moggi. "Constructive natural deduction and its omega-set interpretation". In: *Mathematical Structures in Computer Science* 1.2 (1991), pp. 215–254.

[Møg14] Rasmus Ejlers Møgelberg. "A type theory for productive coprogramming via guarded recursion". In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS '14. Vienna, Austria: Association for Computing Machinery, 2014. ISBN: 9781450328869. DOI: 10.1145/2603088.2603132. URL: https://doi.org/10.1145/2603088.2603132.

[Nak00] H. Nakano. "A modality for recursion". In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

[ND18] Andreas Nuyts and Dominique Devriese. "Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 779–788. ISBN: 9781450355834.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. "Parametric quantifiers for dependent type theory". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017).

[Odi89] P. G. Odifreddi. *Classical Recursion Theory*. 1989.

[Oos08] Jaap van Oosten. *Realizability, Volume 152: An Introduction to its Categorical Side*. San Diego, CA, USA: Elsevier Science, 2008. ISBN: 0444515844.

[Reu99] Bernhard Reus. "Realizability Models for Type Theories". In: *Electronic Notes in Theoretical Computer Science* 23.1 (1999). Tutorial Workshop on Realizability Semantics and Applications, pp. 128–158.

[Rey83] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism". In: *IFIP Congress*. 1983. URL: https://api.semanticscholar.org/CorpusID:19681636.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[VCW12] Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. "Stop When You Are Almost-Full". In: *Interactive Theorem Proving*. Ed. by Lennart Beringer and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 250–265. ISBN: 978-3-642-32347-8.

[Vez15] Andrea Vezzosi. "Guarded Recursive Types in Type Theory." Licentiate Thesis. Chalmers University of Technology, 2015.

[VW19] Niccolò Veltri and Niels van der Weide. "Guarded Recursion in Agda via Sized Types". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 32:1–32:19. ISBN: 978-3-95977-107-8.