Constructing (Co)inductive Types via Large Sizes

Bastiaan Laarakker
LIACS, Leiden University, The Netherlands

Daniël Otten
ILLC, University of Amsterdam, The Netherlands

Benno van den Berg
ILLC, University of Amsterdam, The Netherlands

Abstract

To ensure decidability and consistency of its type theory, a proof assistant should only accept terminating recursive functions and productive corecursive functions. Most proof assistants enforce this through syntactic conditions, which can be restrictive and non-modular. Sized types are a type-based alternative where (co)inductive types are annotated with additional size information. Well-founded induction on sizes can then be used to prove termination and productivity.

An implementation of sized types exists in `Agda`, but it is currently inconsistent due to the addition of a largest size. We investigate an alternative approach, where intensional type theory is extended with a large type of sizes and parametric quantifiers over sizes. We show that inductive and coinductive types can be constructed in this theory, which improves on earlier work where this was only possible for the finitely-branching inductive types. The consistency of the theory is justified by an impredicative realisability model, which interprets the type of sizes as an uncountable ordinal.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Sized Types, Parametricity, Realisability, Impredicativity, Constructive Ordinals, (Co)inductive Types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2026.21

Related Version This paper is based on the master's thesis of the first author, which was supervised by the second and third authors.
Full Version: `https://arxiv.org/abs/2602.18921` [28]

Funding *Benno van den Berg and Daniël Otten:* This publication is part of the project "The Power of Equality" (with project number OCENW.M20.380) of the research programme Open Competition Science M 2 which is financed by the Dutch Research Council (NWO).
Bastiaan Laarakker: This work was partially supported by the Dutch Research Council (NWO) under grant number VI.Veni.232.286 (ChEOpS).

Acknowledgements We are thankful to Andreas Abel, Jesper Cockx, and Andreas Nuyts for helpful discussions and ideas.

1 Introduction

Dependent type theory can be viewed as both a programming language and a foundation of mathematics. In particular, the Curry-Howard correspondence allows us to use dependently typed programming languages such as `Agda`, `Dedukti`, `Lean`, and `Rocq` as proof assistants. To ensure consistency, these languages should only accept total functions, so they put bounds on recursive calls. For example, if the function has an inductive domain (freely generated by constructors), then we know that the function terminates if it is only called recursively on a smaller input. Dually, if the function has a coinductive codomain (cofreely generated by destructors), then we know that the function is productive if its output is larger than that of its recursive calls. In both of these cases, we know that the function is total; however, totality can also be the result of a more intricate mix of induction and coinduction.

Syntactic Totality. Totality is usually enforced via syntactic checks for termination and productivity. By default, `Rocq` and `Lean` check for structural (co)recursion [24, 19, 42], while `Agda` and `Dedukti` use the more permissive size-change termination principle [29]. Both conditions rely on a syntactic comparison of the structural size of the input/output of the original function with the structural size of the input/output of the recursive call. However, programs with more complex patterns of (co)recursion are not easily defined in a way that satisfies these syntactic conditions. Moreover, syntactic methods are sensitive to slight reformulations of programs, and do not work well with higher-order programs [5].

Type-based Totality. This work is concerned with a type-based approach to enforcing totality: sized types. Sized types were originally introduced by Hughes, Pareto, and Sabry [26] as an alternative approach to both termination and productivity checking, in which these properties are guaranteed by the typing derivation instead of an inspection of syntax. In this approach, both inductive and coinductive types can be annotated with additional size information.

We can think of sizes as ordinals, and they come equipped with a well-founded ordering. In particular, for every size i and inductive type Ind we have an ordinal approximation Ind^i, which can be seen informally as the subtype of Ind consisting of terms that can be formed within i constructor steps. Dually, for every size i and coinductive type CoInd we have an ordinal approximation CoInd^i, which can be seen informally as the quotient of CoInd, where terms are identified if we cannot distinguish them within i destructor steps.

By defining functions on these annotated versions of the (co)inductive types, we introduce information about the sizes of the input/output directly in the type of the function. This allows for greater compositionality of functions, since size-information is retained across function calls. Moreover, it allows us to reduce both termination and productivity of recursive functions to well-founded induction on the size parameter.

Sized types, and similar approaches, have been proposed in different forms to aid termination/productivity checking [52, 16, 12, 2, 3]. The combination of sized types with dependent type theory has been studied as well [13, 38, 39, 7, 18]. Sized types have been implemented in `Agda` [4], and there have been informative practical uses, including: encoding regular languages coalgebraically [6], up-to techniques for bisimilarity [21], and constructing quotient-inductive types [23].

Consistency. Importantly, `Agda` does not allow pattern matching on sizes: they should have no influence on computation, and are only used to aid the termination checker. However, there are still significant limitations to both the current implementation of sized types, and to other proposals. One design question is how the full (co)inductive type $(\mathsf{Co})\mathsf{Ind}$ should relate to its ordinal approximations $(\mathsf{Co})\mathsf{Ind}^i$.

`Agda` introduces a special largest size ∞, which can be thought of as a closure ordinal: an ordinal large enough that the approximation has reached the full (co)inductive type, so that we can define $(\mathsf{Co})\mathsf{Ind} \coloneqq (\mathsf{Co})\mathsf{Ind}^\infty$. However, this approach leads to inconsistencies: `Agda` proves $\infty < \infty$, while any strict well-order $<$ can be proven to be irreflexive [46].

We employ a different way of recovering the full (co)inductive type from its ordinal approximations: using parametric quantifiers [37, 35, 34]. The intuition is that the quantifiers $\exists/\forall$ are versions of Σ/Π that restrict information: the type $\exists i.\, A(i)$ contains dependent pairs where the first component is ignored when checking equality, while the universal quantifier $\forall i.\, A(i)$ contains dependent functions that act uniformly on any input. With these quantifiers, we can define the full inductive type as $\mathsf{Ind} \coloneqq \exists i.\, \mathsf{Ind}^i$, and the full coinductive

type as $\mathsf{CoInd} \coloneqq \forall i.\, \mathsf{CoInd}^i$. This follows to our intuition: a term of an inductive type should be bounded by some possibly infinite height (where the bound is kept abstract), whereas a term of a coinductive type should be observable up to any depth (where the depth does not influence observations). Using Σ/Π instead of $\exists/\forall$ would make (Co)Ind too large, so our parametric quantifiers have the same effect as forbidding pattern matching on sizes.

To our knowledge, using parametric quantifiers for sizes was first described by Vezzosi [50], although it was framed in the context of guarded type theory [33, 10] to extend its applicability from coinductive to inductive types. It was later adapted by Nuyts, Vezzosi and Devriese in work on dependent type theory with parametric quantifiers [35, 34], where this parametric approach to sized types was described as a possible application. In their work, they also give a way to construct (co)inductive types: they construct all coinductive types but only the finitely-branching inductive types; this is because the type of sizes is interpreted as the set of natural numbers in their model. The interpretation of sizes as a larger ordinal has been explored [2, 18], but not in combination with parametricity for sizes, which is central to our construction of (co)inductive types.

Contribution. We show how all (co)inductive types can be encoded using sizes and parametric quantifiers. Concretely, we extend a version of Martin-Löf type theory – containing an empty type, unit type, boolean type, Σ-types, Π-types, $=$-types, one universe, function extensionality, and no primitive (co)inductive types – with two additional features: a type of sizes, and parametric quantifiers $\exists$ and $\forall$ over sizes. In our formulation, parametricity is internalised via the addition of axioms, and quantifying over small types will produce a small type although the type of sizes is large.

We show how well-founded recursion on sizes can be used to construct ordinal approximations for any (co)inductive type. Then, we use the parametric quantifiers to construct the full (co)inductive types: we construct both the initial algebra and the final coalgebra for any polynomial endofunctor. Our definitions do not use the uniqueness of identity proofs (UIP) axiom, and are therefore compatible with homotopy type theory [44].

We show the consistency of our theory by constructing a realisability model based on Hyland's effective topos [27]: small types are interpreted as partial equivalence relations, and large types as assemblies. This lets us interpret the type of sizes as an uncountable ordinal, which is needed to validate the axioms for the parametric quantifiers. This models a stronger type theory: one that is also impredicative and extensional and therefore satisfies UIP.

Importantly, we do not use impredicativity and extensionality in the syntax when constructing the ordinal approximations and full (co)inductive types. So, our constructions can be carried out in a weak theory, while our model validates a strong theory.

Related work. Beyond the work that we already mentioned it is worth noting the similarity between this approach to sized types and guarded type theory [33], as this was also the context in which this parametric approach to sizes was described [50]. Guarded type theory introduces a modality $\blacktriangleright$, called the later modality. This modality allows for the construction of guarded recursive types, in which the recursive variable only occurs under the guarding modality. However, the guarding modality by itself is not sufficient to construct coinductive types. Instead, the type system can be extended with so-called clock variables [10], which represent different time streams and can be used to define coinductive types via universal quantification over clocks. This construction requires quantification over clocks to be parametric, which is often expressed in the form of additional axioms [32, 15]. This is very similar to our approach, as we encode coinductive types using parametric universal quantification over sizes. However,

we also encode inductive types using existential quantification, whereas guarded type theory focuses on constructing coinductive types. The relation between sized types and guarded type theory has been studied, and it was shown that guarded recursion can be simulated in a simple type theory using sized types [49].

Another method for guaranteeing termination is via user-defined well-founded orderings [30], which is possible in `Lean`, or using even weaker orderings [51]. `Rocq` also has the `equations` package [40, 31, 41] that extends the termination checker at the cost of losing computational behaviour. Although more flexible, these methods require significant manual work from the user.

Structure of the paper. Section 2 defines the theory in which we work, introducing the type of sizes and parametric quantification. In Section 3, we develop the encoding the approximations and for the full (co)inductive types by quantification over sizes. The consistency of the theory is justified in Section 4, which describes the main features of the realisability model. Finally, a summary and perspective on future work are given in Section 5.

2 The type theory

We highlight some features of the dependent type theory that we work in, which is given in its entirety in Appendix A. Our starting point is Martin-Löf type theory with the empty type ($\bot$), unit type ($\top$) and boolean type (Bool), together with dependent pair types (Σ), dependent function types (Π), and propositional equality types ($=$). For Σ and Π we include definitional η rules.

We also include a Tarski-style universe U, meaning that it comes with an explicit decoder El from terms of U to types, which we omit when possible for readability. The universe contains terms that act as codes for the basic types, and similar codes for $\Pi x : A.\, B(x)$, $\Sigma x : A.\, B(x)$ and $a =_A a'$ when both A and $B(x)$ have codes in U. For each type (former), we use a superscript $(-)^U$ to distinguish the code (term of U) from the type when needed, although we will often omit this for readability. We will call types with codes in the universe *small*, whereas general types are *large*. 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.

Although we do not work with univalence, we will make use of terminology and typical constructions from homotopy type theory [44]. In particular, the notion of an equivalence will be important to this work: we would like to say that a function $f : A \to B$ is an *equivalence*, written $f : A \simeq B$, if it has a *quasi-inverse*, which is a map $g : B \to A$ such that both compositions are pointwise equal to the identity. However, we want to ensure that the type stating that f is an equivalence a *mere proposition*, meaning that all terms of the type are equal. To this end, we also assume a coherence condition between the proofs of pointwise equality; see [44, Definition 4.2.1], where this definition is called a half-adjoint equivalence. We say that a type A is *contractible* when there exists exactly one term of type A up to propositional equality. We may also refer to terms of identity types as paths, and use operations defined by path induction, such as the action of a function $f : A \to B$ on paths and the transport operation.

We also include the *function extensionality* axiom as described in homotopy type theory [44], which states that for any two dependent functions $f, g : \Pi x : A.\, B(x)$, the canonical function $\mathsf{happly} : (f = g) \to \Pi x : A.\, (f\, x = g\, x)$ defined by path induction is an equivalence:

$$(f = g) \simeq \Pi x : A.\, (f\, x = g\, x). \qquad \text{(function extensionality)}$$

We call the quasi-inverse to happly obtained via this equivalence funext.

2.1 Sizes and parametric quantifiers

We extend the theory with a primitive type Size, which comes with a zero term and a successor operation:

$$\frac{}{\Gamma \vdash \mathsf{Size}\ \mathsf{type}} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Size}} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \uparrow i : \mathsf{Size}}$$

So far this looks a lot like the type of natural numbers; the difference in their behaviour will be the elimination principle, for which we first need to introduce some other notions. In particular, we assume a preorder (also known as quasi-order) relation $\leq$ on the type of sizes, which will be a small type. We encode this relation as a mere proposition, which means that for any two terms $p, q : i \leq j$ we have a propositional equality $p = q$. The ordering on sizes is encoded by the following rules:

$$\frac{\Gamma \vdash i : \mathsf{Size} \qquad \Gamma \vdash j : \mathsf{Size}}{\Gamma \vdash i \leq j\ \mathsf{type}} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_0\, i : 0 \leq i} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_{\mathsf{suc}}\, i : i \leq \uparrow i}$$

$$\frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_{\mathsf{refl}}\, i : i \leq i} \qquad \frac{\Gamma \vdash p : i \leq j \qquad \Gamma \vdash q : j \leq k}{\Gamma \vdash \mathsf{le}_{\mathsf{trans}}\, p\, q : i \leq k}$$

We let $i < j$ be notation for $\uparrow i \leq j$. To discuss the elimination and computation principles for sizes, we first need to add universal and existential quantifiers over sizes. These are impredicative: although the type of sizes is large, quantifying over small types will result in a small type:

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

These quantifiers will have very similar rules to Σ and Π; however, we will introduce additional axioms in Subsection 2.2 to internalise a notion of parametricity: sizes must be treated uniformly by quantifiers. As with $\Sigma i : \mathsf{Size}.\, A(i)$, terms of $\exists i.\, A(i)$ are pairs $\langle s, a \rangle^{\exists}$ where $s : \mathsf{Size}$ and $a : A(s)$. However, unlike Σ, the elimination principle of $\exists$ only allows elimination to a small type P:

$$\frac{\Gamma \vdash s : \mathsf{Size} \qquad \Gamma \vdash a : A}{\Gamma \vdash \langle s, a \rangle^{\exists} : \exists i.A(i)} \qquad \frac{\begin{array}{c}\Gamma, z : \exists i.A(i) \vdash P(z) : U \\ \Gamma, i : \mathsf{Size}, x : A \vdash p(i,x) : P(\langle i, x \rangle^{\exists})\end{array}}{\Gamma, z : \exists i.A(i) \vdash \mathsf{ind}_{\exists}(z, p(i,x)) : P(z)}$$

$$\frac{}{\Gamma \vdash \mathsf{ind}_{\exists}(\langle s, a \rangle^{\exists}, p(i,x)) \equiv p(s,a) : P(\langle s, a \rangle^{\exists})}$$

Consequently, we cannot define a projection to the first component, since the type Size is not small. Thus, the size given to the second component of a term of the existential type is inaccessible via the elimination principle and therefore kept abstract. For $\forall$, the rules are completely analogous to Π:

$$\frac{\Gamma, i : \mathsf{Size} \vdash a(i) : A(i)}{\Gamma \vdash \lambda^{\forall} i : \mathsf{Size}.\, a(i) : \forall i.A(i)} \qquad \frac{\Gamma \vdash f : \forall i.A(i) \qquad \Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash f\, s : A(s)}$$

$$\frac{}{\Gamma \vdash (\lambda^{\forall} i : \mathsf{Size}.\, a(i))\, s \equiv a(s) : A(s)} \qquad \frac{}{\Gamma \vdash (\lambda^{\forall} i : \mathsf{Size}.\, f\, i) \equiv f : \forall i.A(i)}$$

Since we will not allow case distinctions on sizes due to its limited elimination principle, it will not be possible to construct non-parametric functions in the syntax.

Here we have used $\langle \ldots \rangle^{\exists}$ and $\lambda^{\forall}$ to distinguish the pairs and functions from those in Σ-types and Π-types; however, when this does not lead to ambiguity we will drop these superscripts. In fact, we see $\Pi i : \mathsf{Size}.\, A(i) \simeq \forall i.\, A(i)$ where the maps are given by $h \mapsto \lambda^{\forall} i.\, h\, i$

and $h' \mapsto \lambda^{\Pi} i.\, h'\, i$. In particular, this shows that function extensionality for Π implies a function extensionality principle for $\forall$. However, the dual principle $\exists i.\, A(i) \simeq \Sigma i.\, A(i)$ can not be shown, since the elimination principle of $\exists$ does not allow us to build a function to Σ, and this equivalence is not true in the model of Section 4.

Another important notion will be "bounded" quantification over sizes. Recall that for sizes $i, j : \mathsf{Size}$, the type $i < j$ is a mere proposition. 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, as p is unique up to propositional equality. 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 type $\forall j < i.\, A(j)$.

Finally, we can give the elimination principle for the type Size, which comes in the form of a fixpoint operator, corresponding to well-founded induction on sizes:

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

The β-rule for fix ensures that $\mathsf{fix}\, f$ behaves as a fixpoint of the map $g \mapsto \lambda i.\, f\, i\, (\lambda j < i.\, g\, i)$:

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

We can view this as the unfolding rule of the fixpoint operator. The model we construct supports this β-rule as a definitional equality, because the model is extensional, meaning that propositional and definitional equality coincide. However, this definitional equality would lead to a loss of decidable type checking if allowed to unfold indefinitely. Thus, we opt to encode this as a propositional equality for now, and discuss alternatives in Section 5. Importantly, we can prove that, up to propositional equality, $\mathsf{fix}\, f$ is the *unique* term satisfying this unfolding, which we may view as its propositional η-rule:

▶ **Lemma 1** (Uniqueness of fixpoints)**.** *The following type is contractible for any term* $f : \forall i.(\forall j < i.A(j)) \to A(i)$*:*

$$\sum_{h : \forall i.A(i)} \forall i.\, (h\, i = f\, i\, (\lambda j < i.\, h\, j)).$$

Proof. We follow a well-known strategy [44, Theorem 5.4.7] [14, Lemma 3.2]. First, note that the type is inhabited by the pair $(\mathsf{fix} f, \mathsf{fix}^{\beta}\, f)$. For any other pair (g, g^{β}), we need to show $(\mathsf{fix} f, \mathsf{fix}^{\beta} f) = (g, g^{\beta})$. For this, it suffices to give a pointwise equality $e : \forall i.\mathsf{fix} f\, i = g\, i$ and a proof $e^{\beta} : (\mathsf{funext}(e))_*\, (\mathsf{fix}^{\beta} f) = g^{\beta}$. We take $e \coloneqq \mathsf{fix}(\lambda i.\, \lambda e'.\, p)$ where p is a concatenation of

$$\mathsf{fix} f\, i \overset{\mathsf{fix}^{\beta} f\, i}{=\!=} f\, i\, (\lambda j < i.\, \mathsf{fix} f\, j) \overset{\mathsf{ap}_{f\, i}(\mathsf{funext}(\lambda j < i.\, e'\, j))}{=\!=\!=} f\, i\, (\lambda j < i.\, g\, j) \overset{g^{\beta}\, i}{=\!=} g\, i.$$

To define e^{β}, it suffices to show that the following square commutes:

$$\begin{array}{ccc} \mathsf{fix} f\, i & \overset{\mathsf{fix}^{\beta} f\, i}{=\!=} & f\, i\, (\lambda j < i.\, \mathsf{fix} f\, j) \\ {\scriptstyle e\, i}\,\Big\| & & \Big\|\,{\scriptstyle \mathsf{ap}_{f\, i}(\mathsf{funext}(\lambda j < i.\, e\, j))} \\ g\, i & \underset{g^{\beta}\, i}{=\!=} & f\, i\, (\lambda j < i.\, g\, j). \end{array}$$

However, this is given precisely by the β-rule $\mathsf{fix}^{\beta}\, (\lambda i.\, \lambda e'.\, p)$. ◀

2.2 Type equivalences

We want to internalise parametricity for quantifying over sizes in our theory, which we do by adding additional equivalences to the theory as axioms. These equivalences describe how universal and existential quantification over sizes commute with the type constructors for small types. They will allow us to encode (co)inductive types in the calculus, and their consistency will be justified by the model we construct in 4.

Before introducing the additional axioms for the calculus, we show that two equivalences can be constructed in the calculus already. These correspond to swapping function arguments and elements of pairs.

▶ **Proposition 2.** *For small types A and $x : A, i : \mathsf{Size} \vdash B(x,i)$ we have the following equivalences:*

$$\Pi x : A.\, \forall i.B(x,i) \simeq \forall i.\, \Pi x : A.\, B(i,x), \qquad (\exists \text{ parametric over } \Sigma)$$
$$\Sigma x : A.\, \exists i.B(x,i) \simeq \exists i.\, \Sigma x : A.\, B(i,x). \qquad (\forall \text{ parametric over } \Pi)$$

We also require equivalences showing the remaining interactions between the quantifiers and Σ, Π, and $<$, for all *small types* A, $x : A, i : \mathsf{Size} \vdash B(x,i)$, and $i : \mathsf{Size} \vdash C(i)$. These are not constructible in the calculus, so we add these as axioms, similarly to what has been done in guarded type theory [32, 15].

$$\Pi x : A.\, \exists i.B(x,i) \simeq \exists i.\, \Pi x : A.\, \exists j < i.B(x,j), \qquad (\exists \text{ parametric over } \Pi)$$
$$\Sigma x : A.\, \forall i.B(x,i) \simeq \forall i.\, \Sigma x : A.\, B(i,x), \qquad (\forall \text{ parametric over } \Sigma)$$
$$\exists i.\, C(i) \simeq \exists i.\, \exists j < i.\, C(j), \qquad (\exists \text{ parametric over } <)$$
$$\forall i.\, C(i) \simeq \forall i.\, \forall j < i.\, C(j). \qquad (\forall \text{ parametric over } <)$$

More precisely, for each equivalence, a *canonical* map f can already be defined in one direction: for $\exists$ these are the right-to-left maps and for $\forall$ these are the left-to-right maps. Our axioms state that each such f forms an equivalence, giving us a quasi-inverse in the other direction.

Note that the first equivalence 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 for this is as follows: to go from right to left, there must be a limit of all sizes in the range of the dependent product. However, we cannot guarantee that the type B is monotone (also known as covariant) in Size, so we make use of $\exists j < i.B(x,j)$ instead, which is monotone in Size. Importantly, this axiom thus provides an implicit construction of limits of sequences of sizes in the theory, without explicitly adding a limit operator on sizes to the syntax.

With these parametricity axioms, we can also show how the quantifiers interact with fixed types such as the base types:

▶ **Proposition 3.** *For a small type A with $i \notin \mathsf{fv}(A)$, we have the following:*

$$\exists i.\, A \simeq A, \qquad (\exists \text{ parametric over a small fixed type})$$
$$\forall i.\, A \simeq A. \qquad (\forall \text{ parametric over a small fixed type})$$

Proof. We first prove both statements in the special case where $A \coloneqq \top$. For $\exists$, we use the fact that for any type B we have $\bot \to B \simeq \top$, and for $\forall$, we use function extensionality:

$$\exists i.\, \top \simeq \exists i.(\bot \to \exists j < i.\, \bot) \simeq \bot \to \exists i.\, \bot \simeq \top, \qquad \forall i.\, \top \simeq \top.$$

We use this special case and the equivalence $\Sigma x : A.\, \top \simeq A$ to prove the general versions:

$$\exists i.\, A \simeq \exists i.\, \Sigma x : A.\, \top \simeq \Sigma x : A.\, \exists i.\, \top \simeq \Sigma x : A.\, \top \simeq A,$$
$$\forall i.\, A \simeq \forall i.\, \Sigma x : A.\, \top \simeq \Sigma x : A.\, \forall i.\, \top \simeq \Sigma x : A.\, \top \simeq A.$$ ◀

3 Encoding (co)inductive types

In this section, we encode (co)inductive types in the theory by quantifying over sizes, taking inspiration from the approach of previous work [35, 50]. We encode these types by constructing initial algebras and terminal coalgebras for polynomial endofunctors; these functors denote the shape of a (co)inductive type. In particular, inductive types can be described as initial algebras for polynomial endofunctors [1, 11], while coinductive types can be described as terminal coalgebras for polynomial functors, as was done in the construction of coinductive types from inductive types [9].

3.1 Polynomial functors

As we will be especially interested in endofunctors on the "category" U of small types and functions, we will omit El for small types in this section. We will also be interested in the "category" of size-indexed small types, where the objects are functions $A : \mathsf{Size} \to U$, and a morphism between $A, B : \mathsf{Size} \to U$ is a map $f : \forall i.\, (A\, i) \to (B\, i)$, which we will write as $f : A \xrightarrow{i} B$.

▶ Remark. Because we do not assume uniqueness of identity proofs (UIP), U and $\mathsf{Size} \to U$ are not technically categories, but $(\infty, 1)$-categories [45]. So, when we talk about functors it would also be more proper to talk about $(\infty, 1)$-functors, which satisfy additional coherence conditions. Although we suspect that the functors we give here satisfy these coherences, we do not require these conditions for our purpose. Thus, our internal notions corresponds to those of *wild category theory* [14], although we will stick to the terms category and functor. For example, an endofunctor on U (a functor from U to U) consists of the following terms:

$$F : U \to U, \quad \text{and} \quad F_{A,B} : (A \to B) \to (FA \to FB) \text{ for all } A, B : U,$$

which satisfy

$$\phi_A : F_{A,A}(\mathsf{id}_A) = \mathsf{id}_{FA}, \quad \text{and} \quad \phi_{f,g} : F_{A,C}(g \circ f) = F_{B,C}\, g \circ F_{A,B} f.$$

We briefly recall polynomial endofunctors, and how they relate to inductive types (W-types) and coinductive types (M-types), which are the types of well-founded trees and non-well-founded trees respectively. Given a small type $A : U$ and a type $B : A \to U$, the polynomial endofunctor $P_{A,B}$ on the category of small types is defined as:

$$P_{A,B}(X) \coloneqq \Sigma a : A.\, (B(a) \to X),$$
$$P_{A,B}(f) \coloneqq \lambda \langle a, g \rangle.\, \langle a, f \circ g \rangle.$$

We can encode W-types and M-types as initial algebras and terminal coalgebras of polynomial endofunctors respectively. These W-types and M-types can in turn be used to encode all inductive and coinductive types. We give some examples of (infinitely-branching) inductive types encoded as W-types. More on W-types and M-types, their elimination principles, and their construction can be found in, for example, [1, 9].

Given $A : U$ and $B : A \to U$, the associated W-type $\mathrm{W} x : A.\, B(x)$ of well-founded trees has a single constructor rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : B(a) \to \mathrm{W}x : A.\, B(x)}{\Gamma \vdash \mathsf{tree}(a, f) : \mathrm{W}x : A.\, B(x)}$$

Intuitively, the type A determines the shapes or constructors of nodes in a well-founded tree, and B determines the branching structure, i.e. arity of each constructor. The term $\mathsf{tree}(a, f)$ can be thought of as the tree with the root node labelled by a, and children given by f. For example, because the type of natural numbers has two constructors, zero and successor, it is encoded as a W-type by taking the shape to be the boolean type. The zero constructor takes no arguments and the successor constructor takes a single argument. Thus, the type $B : \mathsf{Bool} \to U$ determining the arity should defined by $B(\mathsf{ff}) = \bot$ and $B(\mathsf{tt}) = \top$.

More interestingly, infinitely-branching inductive types can also be encoded as W-types by letting B be infinite. One example is the inductive type Brw of Brouwer ordinals, which extends the constructors for the natural numbers with a limit constructor $\mathsf{lim} : (\mathbb{N} \to \mathsf{Brw}) \to \mathsf{Brw}$. Informally, assuming an existing type of natural numbers, this is encoded as the type $\mathrm{W}x : A.\, B$ where $A \coloneqq \top + \top + \top$ is a type with three terms, which we may suggestively call $\mathsf{zero}, \mathsf{succ}$, and lim. We can then define $B : A \to U$ by

$$B(\mathsf{zero}) = \bot, \quad B(\mathsf{succ}) = \top, \quad \text{and} \quad B(\mathsf{lim}) = \mathbb{N}.$$

Infinitely-branching well-founded trees are also essential in the model of constructive set theory given by Aczel [8]. In this model, the type of all sets is given by the W-type $W_{A:U} A$, where U is the type of the constructors, and the arity is given by the identity on U. The idea for this type is that each set can be viewed as a well-founded tree, where its direct successors are the elements of the set.

3.2 Size-indexed functors

In the next sections, we will develop our theory for general functors on small types and size-indexed small types, and show how polynomial functors fit in this framework. We will call endofunctors on the category of size-indexed small types *size-indexed endofunctors*. Following existing notation [35, 50], we define size-indexed endofunctors $\Diamond$ and $\Box$, corresponding to bounded existential and universal quantification.

▶ **Proposition 4.** *We have an endofunctor of the category* $\mathsf{Size} \to U$*:*

$$\Diamond : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U), \qquad \Diamond_{A,B} : (A \xrightarrow{i} B) \to (\Diamond A \xrightarrow{i} \Diamond B),$$
$$\Diamond\, A\, i \coloneqq \exists j < i.\, A\, j; \qquad \Diamond_{A,B}\, f\, i\, \langle j, a\rangle \coloneqq \langle j, f\, j\, a\rangle.$$

▶ **Proposition 5.** *We have an endofunctor of the category* $\mathsf{Size} \to U$*:*

$$\Box : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U), \qquad \Box_{A,B} : (A \xrightarrow{i} B) \to (\Box A \xrightarrow{i} \Box B),$$
$$\Box\, A\, i \coloneqq \forall j < i.\, A\, j; \qquad \Box_{A,B}\, f\, i\, g \coloneqq \lambda j.\, f\, j\, (g\, j).$$

We will write $\Diamond_i\, A$ for $\Diamond\, A\, i \coloneqq \exists j < i.\, A\, j$ and similarly $\Box_i\, A$ for $\Box\, A\, i \coloneqq \forall j < i.\, A\, j$.

Given a functor F on small types, we can make use of $\Diamond$ to "lift" F to the category of size-indexed types by precomposing with $\Diamond$:

$$F[\Diamond -] : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U), \qquad F[\Diamond -]_{A,B} : (A \xrightarrow{i} B) \to (F[\Diamond\, A] \xrightarrow{i} F[\Diamond\, B]),$$
$$F[\Diamond\, A]\, i \coloneqq F(\Diamond_i\, A) \coloneqq F(\exists j < i.\, A\, j); \qquad F[\Diamond\, f]_{A,B}\, i \coloneqq F_{\Diamond_i A, \Diamond_i B}(\Diamond_{A,B}\, f\, i).$$

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. Analogously, $F[\square -]$ is defined by precomposition with $\square$:

$$F[\square -] : (\mathsf{Size} \to U) \to (\mathsf{Size} \to U), \qquad F[\square -]_{A,B} : (A \xrightarrow{i} B) \to (F[\square\, A] \xrightarrow{i} F[\square\, B]),$$
$$F[\square\, A]\, i \coloneqq F(\square_i\, A) \coloneqq F(\forall j < i.\, A\, j); \quad F[\square\, f]_{A,B}\, i \coloneqq F_{\square_i A, \square_i B}(\square_{A,B}\, f\, i).$$

3.3 Inductive types

Since we encode inductive type as initial algebras, we recall the notion of an algebra for a endofunctor on U and define it for size-indexed functors.

▶ **Definition 6** (Algebra). *Given an endofunctor $F : U \to U$, an* F-algebra *is a pair (A, k_A) consisting of a type $A : U$ and a structure map $k_A : FA \to A$. A map of coalgebras $(h, s_h) : (A, k_A) \to (B, k_B)$ is a pair of a map $h : A \to B$ and a path $s_h : h \circ k_A = k_B \circ Fh$. We write F-Alg for the type of F-algebras. An* initial F-algebra *is an algebra $(\mu F, \mathsf{in})$, such that for any algebra (A, k_A), the following type is contractible:*

$$\sum_{h : \mu F \to A} (h \circ \mathsf{in} = k_A \circ Fh).$$

We define *size-indexed F-algebras* as algebras for $F[\diamond -]$. Explicitly, a size-indexed F-algebra is a pair (A, k_A) where $A : \mathsf{Size} \to U$ and $k_A : F[\diamond A] \xrightarrow{i} A$.

To encode size-indexed inductive types, we need to construct initial algebras for size-indexed functors. This is done in the following lemma, where we make use of the fixpoint operator:

▶ **Lemma 7.** *Let $F : U \to U$ be a functor. The size-indexed functor $F[\diamond -]$ has an initial algebra $(\mu^\diamond F, \mathsf{in}^\diamond)$, where the first component is given by:*

$$\mu^\diamond F\, i \coloneqq \mathsf{fix}(\lambda i.\lambda X.F(\diamond_i X))\, i.$$

Proof Sketch. We note that the propositional β-rule for the fixpoint operator gives us the following equality:

$$\begin{aligned}\mu^\diamond F\, i &\coloneqq \mathsf{fix}(\lambda i.\, \lambda X.\, F(\diamond_i X))\, i \\ &= (\lambda i.\, \lambda X.\, F(\diamond_i X))\, i\, (\lambda r < s.\, \mathsf{fix}(\dots)\, r) \\ &\equiv F(\diamond_i\, \mathsf{fix}(\lambda s.\, \lambda X.\, F(\diamond_s X))) \\ &=: F[\diamond(\mu^\diamond F)]\, i.\end{aligned}$$

The structure map $\mathsf{in}^\diamond : F[\diamond(\mu^\diamond F)] \xrightarrow{i} \mu^\diamond F$ is then given by the pointwise transport operation. Given an algebra (A, k_A), the term $\mathsf{fold}^\diamond\, A\, k_A : \mu^\diamond F \xrightarrow{i} A$ is also defined using the fixpoint operator, and can be shown to be an algebra morphism. Initiality then follows from the contractibility of fix described in Lemma 1, together with function extensionality. ◀

Thus, we have a way of constructing size-indexed types, which can be thought of as the ordinal approximations for our inductive types. But our main interest is in showing that F itself has an initial algebra that can be constructed via existential quantification over the initial $F[\diamond -]$-algebra. To this end, we first note the following properties:

▶ **Lemma 8.** *Every F-algebra (A, k_A) induces an $F[\diamond -]$-algebra (TA, k_{TA}), where the first component is $TA \coloneqq (\lambda i.A) : \mathsf{Size} \to U$. This extends to a functor $T : F$-$\mathsf{Alg} \to F[\diamond -]$-$\mathsf{Alg}$.*

Proof Sketch. We require a structure map $k_{TA} : F[\Diamond TA] \xrightarrow{i} TA$. Noting that $i \notin \mathrm{fv}(A)$, we can define the following auxiliary term:

$$\mathsf{extract} : \Pi A : U.\, \Diamond TA \xrightarrow{i} TA,$$
$$\mathsf{extract}_A\, i\, \langle j, a\rangle \coloneqq a.$$

This is well-defined because $TA\, j \equiv A$. As the name suggests, it allows us to extract a term away from the irrelevant size-index. This allows us to define $k_{TA}\, i \coloneqq k_A \circ F(\mathsf{extract}_A\, i)$, since $F(\mathsf{extract}_A\, i) : F[\Diamond_i TA] \to FA$.

Given a algebra map $f : (A, k_A) \to (B, k_B)$, we lift this to a map between size-indexed algebras $T\, f : (TA, k_{TA}) \to (TB, k_{TB})$ defined by $Tf \coloneqq \lambda i.f$. The fact that this forms an algebra map follows from the fact that $\mathsf{extract}_A$ is natural in A. ◀

We can also define a functor from size-indexed algebras to algebras by existential quantification on the carrier of the size-indexed algebra. However, this construction only works on the class of functors satisfying the following condition. As we will see, the equivalences we axiomatised ensure that this includes all polynomial functors.

▶ **Definition 9.** *Let F be an endofunctor on the category of small types. We say that F* weakly commutes *with the existential type if the following canonical map is an equivalence for any $A : \mathsf{Size} \to U$:*

$$\mathsf{can}_A : \exists i.\, F[\Diamond A]\, i \to F(\exists i.A\, i), \qquad \varphi_A : \forall i.\, (\exists j < i.A\, j \to \exists j.A\, j),$$
$$\mathsf{can}_A\, \langle i, a'\rangle \coloneqq F\, (\varphi_A\, i)\, a' \qquad \varphi_A\, i\, \langle j, a\rangle \coloneqq \langle j, a\rangle.$$

For functors that satisfy this property, we can define a functorial map $\exists : F[\Diamond -]\text{-}\mathsf{Alg} \to F\text{-}\mathsf{Alg}$ by lifting the existential quantifier to work on algebras.

▶ **Proposition 10.** *Given a functor F on the category of small types that weakly commutes with the existential quantifier, a size-indexed F-algebra (A, k_A) gives rise to an F-algebra $(\exists i.A\, i, k_{\exists A})$. Moreover, this operation is functorial.*

Proof Sketch. We first note that for any map $f : A \xrightarrow{i} B$, we can define a map between existentially quantified types as follows:

$$\exists f : \exists i.A\, i \to \exists i.B\, i,$$
$$\exists f\, \langle i, a\rangle \coloneqq \langle i, f\, i\, a\rangle.$$

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. We define the map $k_{\exists A} : F(\exists i.A\, i) \to \exists i.A\, i$ by

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

Given an algebra map $f : (A, k_A) \to (B, k_B)$, we define an algebra map $(\exists i.A\, i, k_{\exists A}) \to (\exists i.B\, i, k_{\exists B})$ as $\exists f : \exists i.A\, i \to \exists i.B\, i$. The fact that this constitutes an algebra morphism follows from can_A being natural in A. Thus, we have a functor $\exists : F[\Diamond -]\text{-}\mathsf{Alg} \to F\text{-}\mathsf{Alg}$. ◀

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 11.** *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:*

$$\mu F \coloneqq \exists i.(\mu^{\diamond} F)\, i.$$

Proof Sketch. The categorical intuition is that this follows from the fact that the functorial map $\exists : F[\diamond -]\text{-}\mathsf{Alg} \to F\text{-}\mathsf{Alg}$ of the previous proposition is the left adjoint of $T : F\text{-}\mathsf{Alg} \to F[\diamond -]\text{-}\mathsf{Alg}$. Since left adjoints preserve initial objects, we find that $\exists i.(\mu^{\diamond} F)\, i$ is the initial algebra of F, with the structure map $\mathsf{in} \coloneqq \exists\, \mathsf{in}^{\diamond} \circ \mathsf{can}^{-1}_{\mu^{\diamond} F}$, which is also an equivalence.

We can work this out in more detail: for any $A : \mathsf{Size} \to U$ and $B : U$, there is an isomorphism between maps of type $A \xrightarrow{i} TB$ and those of type $\exists i.A\, i \to B$, given by the currying isomorphism. This extends to an isomorphism between size-indexed algebra maps of type $(A, k_A) \to (TB, k_{TB})$ and algebra maps of type $(\exists i.A\, i, k_{\exists A}) \to (B, k_B)$. Thus, defining an algebra map $(\mu F, \mathsf{in}) \to (B, k_B)$ is equivalent to defining a size-indexed algebra map $(\mu^{\diamond} F, \mathsf{in}^{\diamond}) \to (TB, k_{TB})$. This map is uniquely given by the initiality of $(\mu^{\diamond} F, \mathsf{in}^{\diamond})$. ◀

Lastly, we show that all polynomial endofunctors on small types satisfy the condition of weakly commuting with the existential quantifier. The equivalences that we have axiomatised entail the following chain of equivalences, which corresponds precisely to the canonical map denoted in Definition 9, noting that $i \notin \mathrm{fv}(A)$ and $i \notin \mathrm{fv}(B(a))$ for $a : A$:

$$\begin{aligned}
\exists i.P_{A,B}(\diamond X)\, i &\coloneqq \exists i.\, \Sigma a : A.\, (B(a) \to \exists j < i.X\, j) \\
&\simeq \Sigma a : A.\, (\exists i.B(a) \to \exists j < i.X\, j) \\
&\simeq \Sigma a : A.\, (B(a) \to \exists i.\exists j < i.X\, j) \\
&\simeq \Sigma a : A.\, (B(a) \to \exists i.X\, i) =: P_{A,B}(\exists i.X\, i).
\end{aligned}$$

Since all polynomial functors (both finitely- and infinitely-branching) weakly commute with the existential quantifier, their initial algebra is constructed via existential quantification over the corresponding initial size-indexed algebra. This allows us to construct W-types via quantification over size-indexed types.

3.4 Coinductive types

The coalgebra construction is similar to that of algebras. We define size-indexed coalgebras and construct their terminal coalgebras using the fixpoint operator. This allows us to obtain terminal coalgebras for functors satisfying an analogous property: weakly commuting with the universal quantifier. This is done by showing that the universal quantifier lifts to a functor from size-indexed coalgebras to coalgebras, which is right adjoint to the inclusion map from coalgebras to size-indexed coalgebras. The equivalences allow us to show that all polynomial functors weakly commute with the universal quantifier. Further details can be found in the full version of this paper [28].

4 Realisability model

4.1 Assemblies and partial equivalence relations

In this realisability model, based on Hyland's effective topos, [27], types are interpreted as *assemblies*: sets with associated computability data in the form of a relation between a set of *realisers* and elements of the set. The intuition given by Reus [36] is that realisers can be seen as "codes" in a low-level language or a general model of computation. 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. Maps between such assemblies must be computable in this general model of computation.

The definition of an assembly is parameterised by a general model of computation, which is given by a partial combinatory algebra (pca) [22], providing the set of realisers. We will restrict ourselves to the most well-known pca, the first Kleene algebra $\mathcal{K}_1$, although the construction should work in general. Here, the realisers are the natural numbers, which enumerate the partial recursive functions $\{\varphi_n\}_{n\in\mathbb{N}}$. We define a partial operation $\cdot : \mathbb{N}\times\mathbb{N} \rightharpoonup \mathbb{N}$ on this set of realisers by $m\cdot n := \varphi_m(n)$, and we write $m\cdot n\downarrow$ when $\varphi_m(n)$ is defined.

▶ **Definition 12.** *An* assembly *is a pair $(X, \Vdash_X)$, where X is a set and $\Vdash_X \subseteq \mathbb{N}\times X$ is a relation such that for every $x\in X$ there exists some $n\in\mathbb{N}$ such that $n\Vdash_X x$. We will call $\Vdash_X$ the* realisability *relation of X. A morphism between assemblies $(X,\Vdash_X)$ and $(Y,\Vdash_Y)$ is a function $f : X\to Y$ such that there exists an $e\in\mathbb{N}$ (not part of the data) satisfying:*

$$\forall n\in\mathbb{N}.\,\forall x\in X.\,(n\Vdash_X x) \implies (e\cdot n\downarrow \text{ and } e\cdot n\Vdash_Y f(x)).$$

In this case, we say that f is tracked *by e. We denote the category of assemblies by* Asm.

Note that there is a full and faithful inclusion from sets to assemblies defined by assigning the largest realisability relation; we denote this by $\nabla X := (X, \mathbb{N}\times X)$.

Since morphisms between assemblies are functions that are tracked by a realiser, they must be sufficiently computable. This ensures that the dependent product can be small, even when quantifying over large types, which allows us to model the impredicative universal quantifier. It also allows us to enforce a notion of parametricity in the model via the inclusion from sets to assemblies. The intuition is that ∇X carries no meaningful computational data, making the objects of X indistinguishable when defining morphisms. Since we view the axioms that we added to the theory as a kind of parametricity, this construction allows us to enforce these axioms in the model.

Since we will interpret types as assemblies, we need a sufficiently "small" subcategory to interpret the small types. As a step towards this goal, we first consider the following subcategory of assemblies.

▶ **Definition 13.** *An assembly $(X,\Vdash_x)$ is called a* modest set *if every $n\in\mathbb{N}$ codes at most one $a\in X$:*

$$\forall n\in\mathbb{N}.\,(n\Vdash_X x_1, x_2) \implies (x_1 = x_2).$$

We denote the category of modest sets by Mod, which is a full subcategory of Asm. However, the size of the carrier set of a modest set is not bounded in any way, so we cannot equip the "set" of all modest sets with the structure of an assembly. Instead, we note that the elements of a modest set are not that important: we only need to know when two realisers encode the same element, as this lets us represent an object by the equivalence class of its realisers. This insight gives us a category that is equivalent to the modest sets, namely that of partial equivalence relations on the set of realisers.

▶ **Definition 14.** *A* partial equivalence relation (PER) *is a symmetric and transitive relation $R\subseteq\mathbb{N}\times\mathbb{N}$. We define the following notation:*

$$\mathrm{dom}(R) := \{n\in\mathbb{N} \mid n\,R\,n\},\quad [n]_R := \{m\in\mathbb{N}\mid nRm\},\quad \mathbb{N}/R := \{[n]_R \mid n\in\mathrm{dom}(R)\}.$$

A morphism between PERs R and S is a function $f : \mathbb{N}/R\to\mathbb{N}/S$ such that $f([n]_R) = [e\cdot n]_S$ for some $e\in\mathbb{N}$, in which case we say that f is tracked by e. We denote this category by PER.

Note that a PER R is an equivalence relation on $\mathrm{dom}(R)$.

▶ **Lemma 15.** *There is an equivalence between the categories* PER *and* Mod.

Proof Sketch. Given a PER $R \subseteq \mathbb{N}\times\mathbb{N}$ the pair $\iota(R) \coloneqq (\mathbb{N}/R, \in)$ is a modest set. Conversely, given a modest set $(X, \Vdash_X)$, we can define a relation on the set of realisers as follows:

$$n_1 \sim_X n_2 \iff \exists x \in X.\, (n_1 \Vdash_X x) \wedge (n_2 \Vdash_X x).$$

This induces an equivalence between PERs and modest sets. ◀

In our model, we will use assemblies to interpret large types, while Lemma 15 allows us to make use of the category of PERs to interpret the small types, since every PER induces a modest set with a countable carrier. To describe our model, we make use of the categories with families (CwF) framework, which closely follows the syntactic structure of dependent type theory.

We recall the basic parts of which a CwF $\mathcal{C}$ consists:

- A category $\mathcal{C}$ of *semantic contexts* and context morphisms;
- For every semantic context $\Gamma \in \mathcal{C}$, a collection $\mathsf{Ty}(\Gamma)$ of *semantic types*;
- For every semantic context $\Gamma \in \mathcal{C}$ and semantic type $A \in \mathsf{Ty}(\Gamma)$, a collection $\mathsf{Tm}(\Gamma, A)$ *of semantic terms*;
- For every context morphism $f : \Delta \to \Gamma$, an operation $-[f] : \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Delta)$ for *semantic substitution* on types, and for $A \in \mathsf{Ty}(\Gamma)$, an operation $-[f] : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Delta, A[f])$ for semantic substitution on terms;
- For every semantic context $\Gamma \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Gamma)$, an extended context $\Gamma.A \in \mathcal{C}$.

The category of assemblies can be given the structure of a CwF, with semantic contexts given by assemblies. A semantic type $A \in \mathsf{Ty}(\Gamma)$ is given by a Γ-indexed family of assemblies $(A_\gamma)_{\gamma\in\Gamma}$. Given a semantic type $A \in \mathsf{Ty}(\Gamma)$, a semantic term $a \in \mathsf{Tm}(\Gamma, A)$ is given by a *dependent morphism* $a : (\gamma \in \Gamma) \to A_\gamma$, that is, a dependent function tracked by an $e \in \mathbb{N}$:

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

We will also write this as a family $(a_\gamma)_{\gamma\in\Gamma}$. The semantic substitution operation $-[f]$ on both types and terms is given by precomposing with f. Finally, context extension is given by dependent pairing (as in the ussual set-theoretic model).

We will omit the term "semantic", blurring the distinction between syntax and semantics. Further details on categories with families and the associated structures in assemblies can be found in, for example, [25, 36]. Assemblies support the interpretation of our basic types, dependent products and sums, and the identity type. In the following subsections, we highlight the relevant constructions.

4.2 Universe type

We give a brief description of how the universe type is interpreted in the model, as it will provide intuition on why we interpret the type of sizes in the same way. The universe type U in a context Γ is interpreted as a semantic type $U \in \mathsf{Ty}(\Gamma)$, meaning a Γ-indexed family of assemblies. Since the universe type is a non-dependent type, its interpretation should be a constant family. As we will want to interpret the small types using PERs, we will interpret U as the constant family of an assembly where the carrier set is the set of all PERs, which we also denote by PER. To construct an assembly out of PER, we use the inclusion from sets to assemblies, giving us ∇PER. Recall that the intuition is that ∇PER carries no meaningful computational data, making the PERs computationally indistinguishable when defining morphisms. The decoder El is interpreted by the inclusion of PERs into assemblies.

4.3 Type of sizes

In previous work [35], the type of sizes is interpreted as the set of natural numbers. By contrast, the impredicativity of our realisability model allows us to interpret the type Size as a large type, while still having universal quantification over Size with a small type live in U. 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: the set of all countable ordinals. We construct an assembly from the ordinal ω_1 by giving it the full realisability relation, just as we did for the interpretation of the universe type. This ensures that morphisms from $\nabla\omega_1$ must treat all sizes uniformly.

The reason for interpreting the type of sizes as ω_1 is that it allows us to justify the following equivalence:

$$\Pi x : A.\, \exists i.B(x,i) \simeq \exists i.\, \Pi x : A.\, \exists j < i.B(x,j). \qquad (\exists \text{ parametric over } \Pi)$$

Intuitively, the left-to-right direction of this equivalence states the following: given a dependent function that sends a term $a : A$ to a pair $\langle s, b\rangle : \exists i.B(a,i)$, there is a limit size α such that $s < \alpha$ for each $\langle s, b\rangle$ in the range of the dependent function. This requires limits of sequences of sizes to be a size as well. Note that, since A here is a small type, it is interpreted as a partial equivalence relation over $\mathbb{N}$, hence it has countably many equivalence classes. Since countable limits of countable ordinals are themselves countable, ω_1 suffices for our purpose.

The ordering on sizes is interpreted as the ordering on ω_1, interpreting $s_1 \leq s_2$ as the terminal singleton assembly when true, and the initial empty assembly otherwise. This interpretation validates the encoding of the ordering on sizes as a proposition in the theory.

4.4 Bounded quantification and the fixpoint operator

Recall that the notation $i : \mathsf{Size} \vdash \forall j < i.\, A(j)$ for bounded universal quantification is shorthand for $i : \mathsf{Size} \vdash \forall j.\, (j < i \to A(j))$. A term of this type is interpreted by a family of dependent morphisms, where the domain is bounded by the size in context, which confirms the syntactic intuition for this type.

Given a term $f : \forall i.(\forall j < i.A(j)) \to A(i)$, the fixpoint operator $\mathsf{fix}\, f : \forall i.A(i)$ is interpreted as a morphism, where the underlying dependent function is defined by well-founded induction on ω_1. The definition given by well-founded induction corresponds exactly to the β-rule in the theory. In order for $\mathsf{fix}\, f$ to be a morphism, it must be tracked by a realiser. The following fact, which states that a fixpoint operator exists in every partial combinatory algebra, allows us to find an appropriate realiser for the fixpoint operator.

▶ **Proposition 16** (Fixpoint operator in a pca [48]). *For every pca $(\mathcal{A}, \cdot)$, there exists a term $\mathsf{fix} \in \mathcal{A}$ such that the following hold for any $f, a \in \mathcal{A}$ we have $\mathsf{fix}\, f \downarrow$ and $\mathsf{fix}\, f\, a \simeq f\, (\mathsf{fix}\, f)\, a$.*

4.5 Universal and existential type

The assembly model supports the interpretation of general impredicative products. Although we do not add impredicative products to the theory in full generality, this does allow us to interpret the (impredicative) universal quantifier $\forall i.A(i)$ where $A : U$ in the model simply as a dependent product. The interpretation of sizes as $\nabla\omega_1$ then suffices to validate the equivalences for the universal quantifier.

To interpret the existential type $\exists i.A(i)$, we construct a quotient of the corresponding Σ-type that lives inside the universe. Thus, we should canonically construct a family of PERs from the semantic Σ-type, $\Sigma(\mathsf{Size}, A)$, which is a family of assemblies (the complete definition can be found in the full version [28]). The following lemma gives us a quotient construction resulting in a modest set.

▶ **Lemma 17.** *The inclusion functor* $I : \mathsf{Mod} \to \mathsf{Asm}$ *has a left adjoint* $\mathsf{M} : \mathsf{Asm} \to \mathsf{Mod}$.

Proof Sketch. For an assembly $(A, \Vdash_A)$, we define a modest set $\mathsf{M}(A) = (A/\sim_{\mathsf{M}}, \Vdash_{\mathsf{M}(A)})$ where $\sim_{\mathsf{M}}$ is the least equivalence relation satisfying the following: $\exists n \in \mathbb{N}.\, (n \Vdash_A a_1, a_2) \implies (a_1 \sim_{\mathsf{M}} a_2)$. The relation is defined by $n \Vdash_{\mathsf{M}(A)} [a]_{\sim_{\mathsf{M}}}$ iff $\exists a' \in [a]_{\sim_{\mathsf{M}}} (n \Vdash_A a')$. ◀

However, the universe consists of PERs rather than modest sets. Following Lemma 15, every modest set is isomorphic to a PER. Thus, we may treat $\mathsf{M}(A)$ as the PER that is isomorphic to it. Given a type $A \in \mathsf{Ty}(\Gamma)$, we can define its *U-truncation* $\|A\| \in \mathsf{Tm}(\Gamma, U)$ as the family of PERs $\|A\|_\gamma = \mathsf{M}(A_\gamma)$. Given a small type $A \in \mathsf{Tm}(\Gamma, U)$, this allows us to interpret the small type $\exists i.A(i) \in \mathsf{Tm}(\Gamma, U)$ as the U-truncation of the Σ-type:

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

This also allows us to interpret the pairing operation and elimination of existential types. Moreover, it validates the equivalences we have axiomatised in the theory related to the existential quantifier in the model. Further details of the model are given in the full version [28].

5 Conclusion

We continue the development of sized types with parametric quantifiers over sizes. In previous work [50, 35], the type of sizes was interpreted in the semantics as the natural numbers. This means that the following axiom for small types was restricted to the case where A is finite:

$$\Pi x : A.\, \exists i.B(x,i) \simeq \exists i.\, \Pi x : A.\, \exists j < i.B(x,j). \qquad (\exists \text{ parametric over } \Pi)$$

Our main contribution is a model where the type of sizes is instead interpreted as a larger ordinal. Indeed, for infinite A, the left-to-right direction of the equivalence intuitively shows that limit ordinals exist. This model allows us to remove the restriction that A is finite, which enables us to define all (co)inductive types, not just the finitely-branching ones.

Our model is based on realisability instead of the reflexive graphs used in previous work [50, 35]. This semantics fits well with the intuition of parametric quantifiers: $\exists i.\, A(i)$ behaves like the union of the $A(i)$, while $\forall i.\, A(i)$ behaves like the intersection of the $A(i)$. The model also shows that our syntax is consistent with two additional principles: an impredicative universe of sets, and uniqueness of identity proofs. However, we do not use these properties in the syntax, which makes our construction of (co)inductive types more widely applicable and compatible with homotopy type theory.

Discussion and Future work. A main motivation for our work is obtaining a consistent implementation of sized types in proof assistants. Encoding (co)inductive types using quantifiers would allow the closure ordinal ∞ to be removed from `Agda`, which is a source of inconsistency of the current implementation. However, proof assistants should satisfy desirable computational properties like decidability and canonicity, and this requires computational behaviour for our parametricity axioms. So, to use our results for this purpose, we need to connect them with notions of internal parametricity that compute, such as those based on bridge types [35], which have also been developed in combination with cubical type theory [17, 47].

In addition, we need a computational version of the β-rule for the fixpoint operator. We have only assumed a propositional version of this rule because a naïve definitional version of this rule breaks normalisation due to unrestricted unfolding. For example, it would be

possible to disallow unfolding under binders, only allowing unfolding when a smaller size exists in context, or to ignore computation on sizes entirely. However, each of these options come with drawbacks and it is not clear to us what the most principled approach is.

Because we include suprema sizes, it is not generally possible to keep inequality decidable. In the current Agda implementation, $i < j$ is a judgement that is automatically checked when applying a function $f : (i : \mathsf{Size}_{<j}) \to A$. However, without decidability of inequality we need to ask the user for a term of the type $i < j$, which increases the amount of manual work. One partial solution is to have two versions of inequality: a propositional type $i < j$ and a definitional judgemental fragment $i \lll j$ (similar to the two versions of equality, $=$ and $\equiv$). Then we could add a term ineq so that $\mathsf{ineq} : i < j$ holds whenever $i \lll j$ (similar to the term refl where $\mathsf{refl} : a =_A a'$ holds whenever $a \equiv a'$). This would allow the user to fill in a simple term when the computer can decide the inequality, while still allowing more complicated terms when using suprema sizes.

On the semantic side, it is worth investigating whether we can generalise our realisability model. We restricted the model to the pca of partial recursive functions, as it allowed us to interpret the type of sizes as the first uncountable ordinal. In principle, it seems that the model could generalise to other pcas, as long as the type of sizes is interpreted as a sufficiently large ordinal: one of larger cardinality than the pca. The realizability model could also be constructed using more general notions of computation, such as the monadic combinatory algebras introduced in [20], to allow for an integration of sizes with effects.

Another question is whether we can model both sizes and the univalence axiom. A possibly suitable model for this is given by cubical assemblies [43], which form a model of a type theory with a univalent and impredicative universe. It seems likely that this construction can be extended to include sizes and parametric quantifiers in a similar way as in this work, but a more careful investigation is needed to confirm this.

We would also like to investigate how realisability models can model internal parametric quantifiers and related modalities such as irrelevance in dependent type theory more generally. The inclusion from sets to assemblies assigns a set the complete realisability relation, which intuitively makes elements computationally indistinguishable and enables parametricity for sizes. It seems reasonable that this construction would work for a more general notion of parametric quantifier in the theory.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics. doi:10.1016/j.tcs.2005.06.002.

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

3 Andreas Abel. Semi-continuous sized types and termination. *Log. Methods Comput. Sci.*, 4(2), 2008. doi:10.2168/LMCS-4(2:3)2008.

4 Andreas Abel. Miniagda: Integrating sized and dependent types. In *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, EPiC Series, pages 18–33. EasyChair, 2010. doi:10.29007/322Q.

5 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. *Electronic Proceedings in Theoretical Computer Science*, 77, February 2012. doi:10.4204/EPTCS.77.1.

6 Andreas Abel. Compositional coinduction with sized types. In *Coalgebraic Methods in Computer Science*, Lecture Notes in Computer Science, pages 5–10. Springer, 2016. doi:10.1007/978-3-319-40370-0_2.

7 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.*, 1(ICFP):33:1–33:30, 2017. doi:10.1145/3110277.

8 Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978. doi:10.1016/S0049-237X(08)71989-X.

9 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-Wellfounded Trees in Homotopy Type Theory. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17–30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.TLCA.2015.17.

10 Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. *SIGPLAN Not.*, 48(9):197–208, September 2013. doi:10.1145/2544174.2500597.

11 Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. *J. ACM*, 63(6), January 2017. doi:10.1145/3006383.

12 G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004. doi:10.1017/S0960129503004122.

13 Gilles Barthe, Benjamin Gregoire, and Fernando Pastawski. CIC$^-$: Type-based termination of recursive definitions in the calculus of inductive constructions. In *Math. Struct. in Comp. Science*, pages 257–271, November 2006. doi:10.1007/11916277_18.

14 Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CSL.2016.23.

15 Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 20–35. Springer, 2016. doi:10.1007/978-3-662-49630-5_2.

16 Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, Lecture Notes in Computer Science, pages 24–39. Springer, 2004. doi:10.1007/978-3-540-25979-4_2.

17 Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. *Log. Methods Comput. Sci.*, 17(4), 2021. doi:10.46298/LMCS-17(4:5)2021.

18 Jonathan H.W. Chan. Sized dependent types via extensional type theory. Master's thesis, University of British Columbia, 2022. doi:10.14288/1.0416401.

19 Jesper Cockx, Dominique Devriese, and Frank Piessens. Eliminating dependent pattern matching without K. *J. Funct. Program.*, 26:e16, 2016. doi:10.1017/S0956796816000174.

20 Liron Cohen, Ariel Grunfeld, Dominik Kirst, and Étienne Miquey. From partial to monadic: Combinatory algebra with effects. In *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, Birmingham, UK, July 14-20, 2025*, LIPIcs, pages 14:1–14:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.FSCD.2025.14.

21 Nils Anders Danielsson. Up-to techniques using sized types. *Proc. ACM Program. Lang.*, 2(POPL):43:1–43:28, 2018. doi:10.1145/3158131.

22 Solomon Feferman. A language and axioms for explicit mathematics. *Algebra and Logic*, pages 87–139, 1975.

23 Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. In *Foundations of Software Science and Computation Structures - 23rd International Conference, Proceedings*, Lecture Notes in Computer Science, pages 257–276. Springer, 2020. doi:10.1007/978-3-030-45231-5_14.

24 Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, volume 4060, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science. doi:10.1007/11780274_27.

25 Martin Hofmann. *Syntax and Semantics of Dependent Types*, pages 79–130. Publications of the Newton Institute. Cambridge University Press, 1997.

26 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, pages 410–423, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.240882.

27 J Martin E Hyland. The effective topos. In *Studies in Logic and the Foundations of Mathematics*, volume 110, pages 165–216. Elsevier, 1982.

28 Bastiaan Laarakker, Daniël Otten, and Benno van den Berg. Constructing (co)inductive types via large sizes. *CoRR*, abs/2602.18921, 2026. doi:10.48550/arXiv.2602.18921.

29 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001. doi:10.1145/360204.360210.

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

31 Cyprien Mangin and Matthieu Sozeau. Equations for hereditary substitution in leivant's predicative System F: A case study. *CoRR*, abs/1508.00455, 2015. arXiv:1508.00455.

32 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, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603132.

33 H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000. doi:10.1109/LICS.2000.855774.

34 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, pages 779–788, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209119.

35 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, August 2017. doi:10.1145/3110276.

36 Bernhard Reus. Realizability models for type theories. *Electronic Notes in Theoretical Computer Science*, 23(1):128–158, 1999. Tutorial Workshop on Realizability Semantics and Applications. doi:10.1016/S1571-0661(04)00108-2.

37 John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. North-Holland.

38 Jorge Luis Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 233–242, 2013. doi:10.1109/LICS.2013.29.

39 Jorge Luis Sacchini. Linear sized types in the calculus of constructions. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 169–185, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07151-0_11.

40 Matthieu Sozeau. Equations: A Dependent Pattern-Matching Compiler. In *Interactive Theorem Proving*, volume 6172, pages 419–434. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-642-14052-5_29.

41 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341690.

42 David Thibodeau. *An intensional type theory of coinduction using copatterns*. PhD Thesis, McGill University Montréal, QC, Canada, 2020.

43 Taichi Uemura. Cubical assemblies, a univalent and impredicative universe and a failure of propositional resizing. In *24th International Conference on Types for Proofs and Programs, TYPES 2018, Braga, Portugal, June 18-21, 2018*, LIPIcs, pages 7:1–7:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2018.7.

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

45 Benno Van Den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London mathematical society*, 102(2):370–394, 2011.

46 Twan van Laarhoven. Well-founded induction on Size is inconsistent. https://github.com/agda/agda/issues/3026, 2018. Agda GitHub repository, Issue #3026.

47 Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. Internal and observational parametricity for cubical Agda. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632850.

48 Jaap van Oosten. *Realizability, Volume 152: An Introduction to its Categorical Side*. Elsevier Science, San Diego, CA, USA, 2008.

49 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)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2019.32.

50 Andrea Vezzosi. *Guarded Recursive Types in Type Theory*. Licentiate thesis, Chalmers University of Technology, 2015.

51 Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. Stop when you are almost-full. In *Interactive Theorem Proving*, pages 250–265, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

52 Hongwei Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15:91–131, 2002. doi:10.1023/A:1019916231463.

A The type theory

The symbol $\equiv$ denotes definitional equality in the theory for both types and terms. We make use of the following forms of judgements:

$$\vdash \Gamma\ \mathsf{ctx}, \qquad \Gamma \vdash A\ \mathsf{type}, \qquad \Gamma \vdash A \equiv B\ \mathsf{type}, \qquad \Gamma \vdash a : A, \qquad \Gamma \vdash a \equiv b : A.$$

For brevity, we omit the standard rules stating that definitional equality is a congruence relation with respect to all type- and term formers. We may also assume weakening and substitution rules with respect to all judgements, which are both admissible [25].

The rules for context formation are given as follows:

$$\frac{}{\vdash \diamond\ \mathsf{ctx}} \qquad \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{ctx}}$$

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

The rules for the empty-, unit-, and boolean type are as follows, where their elimination and computation principles are left implicit.

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

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

The rules for Π-types include definitional β- and η-rules.

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

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

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

Similarly to Π-types, Σ-types also come with β- and η-rules. The η-rule for Σ-types is defined via its destructors, where π_1 and π_2 are the projections derived from the elimination principle.

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, x : A \vdash B(x) \ \mathsf{type}}{\Gamma \vdash \Sigma x : A.\, B(x) \ \mathsf{type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash \langle a, b \rangle : \Sigma x : A.\, B(x)}$$

$$\frac{\begin{array}{c}\Gamma, z : \Sigma x : A.\, B(x) \vdash P(z) \ \mathsf{type} \\ \Gamma, x : A, y : B \vdash p(x, y) : P(\langle x, y \rangle)\end{array}}{\Gamma, z : \Sigma x : A.\, B(x) \vdash \mathsf{ind}_\Sigma(z, p(x, y)) : P(z)}$$

$$\frac{\begin{array}{c}\Gamma, z : \Sigma x : A.\, B(x) \vdash P(z) \ \mathsf{type} \\ \Gamma, x : A, y : B \vdash p : P(\langle x, y \rangle)\end{array}}{\Gamma, x : A, y : B \vdash \mathsf{ind}_\Sigma(\langle x, y \rangle, p) \equiv p(x, y) : P(\langle x, y \rangle)}$$

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

Propositional identity types are defined with the standard induction principle.

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

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

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

The universe type is closed under the base types, propositional equality, and the type constructors. The definitional type equalities for the small types ensure that they lift to the proper type.

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

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

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

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

$$\frac{\Gamma \vdash A : U \qquad \Gamma \vdash a : \mathrm{El}\, A \qquad \Gamma \vdash a' : \mathrm{El}\, A}{\Gamma \vdash a =^U_A a' : U}$$

$$\begin{aligned} \mathrm{El}\,(\bot^U) &\equiv \bot & \mathrm{El}\,(\Sigma^U x : A.\, B(x)) &\equiv \Sigma_{x:\mathrm{El}\, A}(\mathrm{El}\, B(x)) \\ \mathrm{El}\,(\top^U) &\equiv \top & \mathrm{El}\,(\Pi^U x : A.\, B(x)) &\equiv \Pi_{x:\mathrm{El}\, A}(\mathrm{El}\, B(x)) \\ \mathrm{El}\,(\mathsf{Bool}^U) &\equiv \mathsf{Bool} & \mathrm{El}\,(a =^U_A a') &\equiv (a =_{\mathrm{El}\, A} a') \end{aligned}$$

The type of sizes is defined with constructors for zero and successor. The ordering on sizes is defined as a mere proposition, with constructors for reflexivity and transitivity.

$$\frac{}{\Gamma \vdash \mathsf{Size}\ \mathsf{type}} \qquad \frac{\Gamma \vdash i, j : \mathsf{Size}}{\Gamma \vdash\ i \leq j : U} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Size}} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \uparrow i : \mathsf{Size}}$$

$$\frac{\Gamma \vdash i : \mathsf{Size} \qquad \Gamma \vdash j : \mathsf{Size}}{\Gamma \vdash i \leq j\ \mathsf{type}} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_0\, i : \mathrm{El}\,(0 \leq i)} \qquad \frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_{\mathsf{suc}}\, i : \mathrm{El}\,(i \leq \uparrow i)}$$

$$\frac{\Gamma \vdash i : \mathsf{Size}}{\Gamma \vdash \mathsf{le}_{\mathsf{refl}}\, i : \mathrm{El}\,(i \leq i)} \qquad \frac{\Gamma \vdash p : i \leq j \qquad \Gamma \vdash q : j \leq k}{\Gamma \vdash \mathsf{le}_{\mathsf{trans}}\, p\, q : \mathrm{El}\,(i \leq k)}$$

$$\frac{\Gamma \vdash p : i \leq j \qquad \Gamma \vdash q : i \leq j}{\Gamma \vdash \mathsf{le}_{\mathsf{eq}}\, p\, q : \mathrm{El}\,(p = q)}$$

The type of the existential quantifier over sizes is defined as follows. Note that the elimination principle is limited to small types.

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

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

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

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

The universal quantifier is defined analogously to the Π-type, only as an impredicative quantifier.

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

$$\frac{\Gamma, i : \mathsf{Size} \vdash a(i) : A(i)}{\Gamma \vdash \lambda^{\forall} i : \mathsf{Size}.\, a(i) : \mathrm{El}\,(\forall i.A(i))} \qquad \frac{\Gamma \vdash f : \mathrm{El}\,(\forall i.A(i)) \qquad \Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash f\, s : \mathrm{El}\, A(s)}$$

$$\frac{\Gamma, i : \mathsf{Size} \vdash a(i) : \mathrm{El}\, A(i) \qquad \Gamma \vdash s : \mathsf{Size}}{\Gamma \vdash (\lambda^{\forall} i : \mathsf{Size}.\, a(i))\, s \equiv a(s) : \mathrm{El}\, A(s)} \qquad \frac{\Gamma \vdash f : \mathrm{El}\,(\forall i.A(i))}{\Gamma \vdash (\lambda^{\forall} i : \mathsf{Size}.\, f\, i) \equiv f : \mathrm{El}\,(\forall i.A(i))}$$

Finally, the fixpoint operator on sizes, corresponding to well-founded induction, is defined as follows. Note that the associated β-rule is propositional.

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

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

Function extensionality is encoded as in [44]: given $f, g : \Pi x : A.\, B(x)$, there is a map

$$(f = g) \to \Pi x : A.\, (f(x) = g(x)),$$

and the function extensionality axiom states that this map is an equivalence. Similarly, we note that for small types A, $B(i, x)$, and $C(i)$ there are canonical maps, for which the parametricity axioms state that each such map is an equivalence:

$$\begin{aligned}
&\Pi x : A.\, \exists i.B(x, i) \leftarrow \exists i.\, \Pi x : A.\, \exists j < i.B(x, j),\\
&\Sigma x : A.\, \forall i.B(x, i) \to \forall i.\, \Sigma x : A.\, B(i, x),\\
&\qquad \exists i.\, C(i) \leftarrow \exists i.\, \exists j < i.\, C(j),\\
&\qquad \forall i.\, C(i) \to \forall i.\, \forall j < i.\, C(j).
\end{aligned}$$