Types are internal ∞-groupoids

Finster, Eric; Sozeau, Matthieu; Allioux, Antoine

DOI:
10.1109/LICS52264.2021.9470541

Types Are Internal ∞-Groupoids

Eric Finster
Cambridge University
Department of Computer Science
ericfinster@gmail.com

Antoine Allioux
Inria & IRIF, Université de Paris
France
antoine.allioux@irif.fr

Matthieu Sozeau
Inria & LS2N, Université de Nantes
France
matthieu.sozeau@inria.fr

***Abstract*—By extending type theory with a universe of definitionally associative and unital polynomial monads, we show how to arrive at a definition of *opetopic type* which is able to encode a number of fully coherent algebraic structures. In particular, our approach leads to a definition of ∞-groupoid internal to type theory and we prove that the type of such ∞-groupoids is equivalent to the universe of types. That is, every type admits the structure of an ∞-groupoid internally, and this structure is unique.**

I. Introduction

Homotopy Type Theory has brought a new perspective to intensional Martin-Löf type theory: the higher identity types of a type endow it with the structure of an ∞-*groupoid*, and ideas from homotopy theory provide us with a means to predict and understand the resulting tower of identifications. While this perspective has been enormously clarifying with respect to our understanding of the notion of proof-relevant equality, leading, as it has, to a new class of models as well as new computational principles, a number of difficulties remain in order to complete the vision of type theory as a foundation for a new, structural mathematics based on homotopy-theoretic and higher categorical principles.

Foremost among these difficulties is the following: how does one describe a well behaved theory of *algebraic structures* on arbitrary types? The fundamental difficulty in setting up such a theory is that, in a proof relevant setting, nearly all of the familiar algebraic structures (monoids, groups, rings, and categories, to take a few) become *infinitary* in their presentation. Indeed, the axioms of these theories, which take the form of a finite list of mere *properties* when the underlying types are sets, constitute additional *structure* when they are no longer assumed to be so. Consequently, in order to arrive at a well-behaved theory, the axioms themselves must be subject to additional axioms, frequently referred to generically as "coherence conditions". In short, in a proof relevant setting, it no longer suffices to describe the equations of an algebraic structure at the "first level" of equality. Rather, we must specify how the structure behaves *throughout the entire tower* of identity types, and this is an infinite amount of data. How do we organize and manipulate this data?

Similar problems have arisen in the mathematics of homotopy theory and higher category theory, and many solutions and techniques are known. Bafflingly, however, all attempts to import these ideas into plain homotopy type theory have, so far, failed. This appears to be a result of a kind of circularity: all of the known classical techniques at *some* point rely on set-level algebraic structures themselves (presheaves, operads, or something similar) as a means of presenting or encoding higher structures. Internally to type theory, however, we do not have recourse to such techniques. Indeed, without further hypotheses, we do not even expect that the most basic objects of the theory, types themselves, are presented by set-level structures. This leaves us in a strange position: absent a theory of algebraic structures, we have nothing to use to encode algebraic structures!

We suggest that a possible explanation for this phenomenon is the following: contrary to our experience with set-level mathematics, where an algebraic structure (i.e. a "structured set") can itself be defined just in terms of sets: underlying sets, functions, sets of relations and so on, when we pass to the world of homotopy theoretic mathematics, the notion of *type* and *structured type* are simply no longer independent of each other in the same way. Consequently, some primitive notion of structured type must be defined *at the same time* as the notion of type itself. The present work is a first attempt at rendering this admittedly somewhat vague idea precise.

We begin by imagining a type theory which, in addition to defining a universe $\mathcal{U}$ of *types*, defines at the same time a universe $\mathcal{S}$ of *structures*. Of course, we will need to be somewhat more precise about what exactly we mean by *structure*. Category theory suggests that one way of representing a structure is by the monad on $\mathcal{U}$ which it defines, so we might think of $\mathcal{S}$ as a universe of monads. In practice, however, it will be useful to restrict to a particularly well behaved class of monads, having reasonable closure properties, and for which we have a good understanding of their higher dimensional counterparts. We submit that a reasonable candidate for such a well-behaved collection is the class of *polynomial monads* [1].

We feel that this is an appropriate class of structures for a number of reasons. A first reason is that this class of monads arises quite naturally in type theory already: indeed, a large literature exists on the interpretation of inductive and coinductive types as initial algebras and terminal coalgebras for polynomial monads, and we consider our work as deepening and extending this connection. Furthermore, this class of algebraic structures enjoys some pleasant properties which make them particularly amenable to "weakening". For example, the very general approach to weakening algebraic structures developed by Baez and Dolan in [2] can be smoothly

978-1-6654-4895-6/21/$31.00

adapted to the polynomial case. While the cited work employs the language of symmetric operads, connections with the theory of polynomial functors were already described in [3], and moreover, recent work [4] has shown that, in type theory, we should expect symmetric operads to in fact be *subsumed* by the theory of polynomial monads.

The central intuition of Baez and Dolan's approach, is that each polynomial monad M determines its *own* higher dimensional collection of shapes (the M-opetopes) generated directly from the syntactic structure of its terms. They go on to introduce the notion of an *M-opetopic type* which is, roughly, a collection of well formed decorations of these shapes, and the notion of weak M-algebra is then defined as an M-opetopic type satisfying certain closure properties. In this sense, their approach differs from, say, approaches based on simplices, cubes or spheres in that the geometry is not fixed ahead of time, but adapted to the particular structure under consideration.

With these considerations in mind, our plan in the present work is to put the idea of a type theory with primitive structures to the test. What might it look like, and what might it be able to prove? In order to answer these questions, we will build a prototype of the theory [1] in the proof assistant Agda, and exploit the recent addition of *rewrite rules* [5] which permits us to extend definitional equality by new well-typed reductions. The use of such rewrites is necessary to ensure that our primitive structures are not subject to the same infinite regress of coherence conditions which has so far obstructed more naive attempts and defining such objects.

Concretely, we will introduce a universe $\mathbb{M}$,[2] whose elements we think of as codes for polynomial monads and describe the structures they decode to. Because we think of the objects of the universe $\mathbb{M}$ as primitives of our theory, on the same level as types, we allow ourselves the freedom to prescribe their computational behavior: in particular, we will equip them with definitional associativity and unit laws using the rewrite mechanism alluded to above. We emphasize that if structures are taken as defined in parallel with types, then this kind of definitional behavior should be no more surprising than, say, the definitional associativity of function composition.

We then show how the existence of our universe $\mathbb{M}$ has some strong consequences. In particular, it allows us to implement the Baez and Dolan definition of opetopic type alluded to above, and subsequently to define a number of weak higher dimensional structures. Among the structures which we are able to define using this technique are $\mathbb{A}_\infty$-monoids and groups, $(\infty, 1)$-categories and presheaves over them (in particular, our setup leads to a definition of *simplicial type*), and as a special case, ∞-groupoids themselves.

[1]The Agda source is available here: https://github.com/ericfinster/opetopic-types

[2]We mean by this notation to distinguish the universe $\mathbb{M}$ of this *particular* implementation from the generic idea of a universe of structures $\mathcal{S}$, the properties of which we expect to be refined by further investigation.

There arises, then, the problem of justifying the correctness of our definitions. In order to do so, we will take up the example of ∞-groupoids in some detail. Indeed, since the homotopical interpretation of type theory asserts that types should "be" ∞-groupoids, it seems natural to compare these two objects. Our main result is the following:

Theorem. *There is an equivalence*

$$\mathcal{U} \simeq \infty\text{-}\mathsf{Grp}$$

In other words, every type admits the structure of an ∞-groupoid in our sense, and that structure is unique. This theorem, therefore, can be regarded as a (constructive) internalization of the intuition provided by the various meta-theoretic results to this effect [6], [7].

A. Related Work

The so-called *coherence problem*, which is the main motivation for the present work, has been considered by a number of authors. We briefly compare our approach with two prominent other strains of thought.

1) Synthetic Structures: One way to avoid some of the problems posed by the definition of higher dimensional structures is to simply enlarge the collection of basic objects to include them. Such approaches may be described as synthetic, in that they do not reduce higher structures to more primitive objects in exactly the same way that homotopy type theory itself does not define ∞-groupoids in terms of sets. This point of view is often adopted, for example, in the research into *directed type theories*, whether they be aimed at specific structures like higher categories as in [9], or allow for more general directed spaces as in [8].

While our theory does indeed add some new primitive structures to type theory, we collect these structures in a universe and decode them into collections of ordinary types and maps between them. Moreover, we go on to use these additional strict structures to give analytic, internal definitions of higher structures.

2) Two-Level Type Theory: Perhaps the closest related work to the current approach is that of Two-Level Type Theory [10]. There it is advocated to add a second "level" to type theory with a set-truncated equality type which one can then use to make meta-theoretic statements about the inner level, whose objects are typically taken to be the homotopically meaningful ones. The two-level approach provides, thus, a great deal of generality and flexibility at the cost of restricting the applicability of homotopical interpretation of types to the inner theory. It is likely, for example, that our theory could be developed inside a two-level system and many of the rewrites we employ proven as theorems in the outer level. By constrast, our approach is, we feel, somewhat more economical, extending the theory with a specific set of rewrite rules, and pointing towards the possibility of a useful theory of higher structures without the need to restrict homotopical principles like univalence.

B. Preliminaries

The basis of our metatheory is the type theory implemented in Agda [11] which is an extension of the predicative part of Martin-Löf type theory [12]. Among the particular types that Agda implements, we shall use inductive types, records and coinductive records.

As such, we adopt a style similar to Agda code, writing $(x : A) \to B\,x$ for the dependent product (although we occasionally also employ the $\prod_{(x:A)} B\,x$ notation if it improves readability). We also make use of the implicit counterpart of the dependent products, written $\{x : A\} \to B\,x$. This allows us to hide arguments which can be inferred from the context and hence clarify our notation. Non-dependent functions are denoted $A \to B$, as usual. Functions enjoy the usual η conversion rule.

We shall make extensive use of coinductive record types, as well as copatterns for producing elements of these types. We write $\top$ for the empty record with a constructor $\mathsf{tt} : \top$. We write $\sum_{(x:A)} B\ x$ for the dependent sum as a record with constructor $\_,\_$ and projections fst and snd. Pairs which are not dependent are denoted $A \times B$.

We write $\bot$ for the empty type, using absurd patterns where appropriate, and writing $\bot$-elim for the unique function for $\bot$ to any type.

The identity type $\_ \equiv \_ : \{A : \mathcal{U}\}\ (x\ y : A) \to \mathcal{U}$ is an inductive type with one constructor $\mathsf{refl} : (x : A) \to x \equiv x$.

We shall make use of the notion of contractible type denoted $\mathsf{is\text{-}contr}$ whose center of contraction will be referred to as ctr. Other notions defined in the HoTT book [13] will be employed including equivalence of types denoted $X \simeq Y$, function extensionality denoted funext, as well as the Univalence Axiom.

We write $\mathcal{U}$ for the universe of small types, and $\mathcal{U}_1$ for the next universe when necessary.

In order to unclutter and clarify the presentation, we occasionally take liberties with the formal definitions, for example, silently inserting applications to functional extensionality when necessary, or reassociating Σ-types in order to avoid a proliferation of commas. Our formal development in Agda allows no such informalities to remain.

II. A Universe of Polynomial Monads

As we have explained in the introduction, type theory appears to lack the ability to speak about infinitely coherent algebraic structures, and our strategy for addressing this problem will be to distinguish a collection of such structures which we consider as defined by the theory itself. We do so using a common technique in the type theory literature: that of introducing a *universe*. We write $\mathbb{M} : \mathcal{U}$ for our universe, and we think of its elements as *codes for polynomial monads*. Just as a typical type theoretic universe has some collection of base types and some collection of type constructors, so we will populate our universe with a collection of "base monads" and "monad constructors". In other words: we will have a syntax of structures which parallels the syntax for types.

Typically, a universe of types $\mathbb{U}$ comes equipped with a decoding function $El : \mathbb{U} \to \mathcal{U}$. In the case of our universe of monads $\mathbb{M}$, each element $M : \mathbb{M}$ will decode not to a single type, but to a collection of types and type families equipped with some structure. We will use rewrite rules to specify the computational behavior of this structure.

A. Polynomial Structure

To begin, we first equip each $M : \mathbb{M}$ with an underlying *polynomial* or *indexed container* [14]. This is accomplished by postulating the following collection of decoding functions:

$$\begin{aligned}
&\mathsf{Idx} : \mathbb{M} \to \mathcal{U}\\
&\mathsf{Cns} : (M : \mathbb{M}) \to \mathsf{Idx}\,M \to \mathcal{U}\\
&\mathsf{Pos} : (M : \mathbb{M})\ \{i : \mathsf{Idx}\,M\} \to \mathsf{Cns}\,M\,i \to \mathcal{U}\\
&\mathsf{Typ} : (M : \mathbb{M})\ \{i : \mathsf{Idx}\,M\}\ (c : \mathsf{Cns}\,M i)\\
&\qquad \to \mathsf{Pos}\,M\,c \to \mathsf{Idx}\,M
\end{aligned}$$

Polynomials of this sort appear in the computer science literature as the "data of a datatype declaration". They can equivalently be seen as a way to describe the signature of an algebraic theory: concretely, the elements of $\mathsf{Idx}\,M$, which we refer to as *indices* serve as the sorts of the theory, and for $i : \mathsf{Idx}\,M$, the type $\mathsf{Cns}\,M\,i$ is the collection of operation symbols whose "output" sort is i. The type $\mathsf{Pos}\,M\,c$ then assigns to each operation a collection of "input positions" which are themselves assigned an index via the function Typ.

It follows that every monad M induces a functor $[\_] : (\mathsf{Idx}\,M \to \mathcal{U}) \to (\mathsf{Idx}\,M \to \mathcal{U})$ called its *extension* given by

$$[M]\,X\,i = \sum_{(c:\mathsf{Cns}\,M\,i)} (p : \mathsf{Pos}\,M\,c) \to X\,(\mathsf{Typ}\,M\,c\,p)$$

We may think of the value of this functor at a type family $X : \mathsf{Idx}\,M \to \mathcal{U}$ as the type of *constructors of M with inputs decorated by elements of X*. Indeed, we will frequently refer to a dependent function of the form

$$(p : \mathsf{Pos}\,M\,c) \to X\,(\mathsf{Typ}\,M\,c\,p)$$

where X is as above, as a *decoration* of c by elements of X.

B. Monadic Structure

Next, for each monad $M : \mathbb{M}$, we are going to equip the underlying polynomial of M with an algebraic structure: specifically, that structure required on the underlying polynomial so that the associated extension $[M]$ becomes a *monad*. In the case at hand, this takes the form of a pair of functions

$$\begin{aligned}
&\eta : (M : \mathbb{M})\ (i : \mathsf{Idx}\,M) \to \mathsf{Cns}\,M\,i\\
&\mu : (M : \mathbb{M})\ \{i : \mathsf{Idx}\,M\}\ (c : \mathsf{Cns}\,M\,i)\\
&\quad \to (\delta : (p : \mathsf{Pos}\,M\,c) \to \mathsf{Cns}\,M\,(\mathsf{Typ}\,M\,c\,p))\\
&\quad \to \mathsf{Cns}\,M\,i
\end{aligned}$$

which equip M with a multiplication and unit operation. We remark that the second argument δ of the multiplication μ is a decoration of c in the family $\mathsf{Cns}\,M$ of constructors, so that we can think of the input to this function as a "two-level" tree.

Crucial for what follows will be that the monads we consider are *cartesian* in the sense of [1]. Type theoretically,

the means we require each monad M to come equipped with equivalences

$$\begin{aligned}\mathsf{Pos}\, M\, (\eta\ M\, i) &\simeq \top\\ \mathsf{Pos}\, M\, (\mu\ M\, c\, \delta) &\simeq \textstyle\sum_{(p:\mathsf{Pos}\, M\, c)} \mathsf{Pos}\, M\, (\delta\, p)\end{aligned}$$

Since we are already modifying the definitional equality of our type theory, it may be tempting to require these equivalences definitionally by asserting that the type of positions reduces when applied to constructors of the appropriate form. However, this will not work: as we will see below, when we come to populate our universe with concrete monads and monad constructors, the equivalences we find are often in fact not definitional, even if they remain provable. As an alternative, we will equip each monad with introduction, elimination and computation rules for its positions which will in effect guarantee that we always have the required equivalence. Each monad definition will then be required to implement these rules in a manner consistent with the various required typing laws.

In the case of η, for example, we postulate introduction and elimination rules of the form

$$\begin{aligned}\eta\text{-pos} &: (M : \mathbb{M})\ (i : \mathsf{Idx}\, M) \to \mathsf{Pos}\, M\, (\eta\ M\, i)\\ \eta\text{-pos-elim} &: (M : \mathbb{M})\ (i : \mathsf{Idx}\, M)\\ &\to (X : (p : \mathsf{Pos}\, M\, (\eta\ M\, i)) \to \mathcal{U})\\ &\to (u : X\ (\eta\text{-pos}\, M\, i))\\ &\to (p : \mathsf{Pos}\, M\, (\eta\ M\, i)) \to X\ p\end{aligned}$$

with typing rule

$$\mathsf{Typ}\, M\, (\eta\ M\, i)\ p \rightsquigarrow i \qquad \text{(Typ-}\eta\text{)}$$

and computation rule

$$\eta\text{-pos-elim}\ \ M\, i\ X\ u\ (\eta\text{-pos}\, M\, i) \rightsquigarrow u$$

Notice these are exactly the rules for an inductively defined indexed unit type.[3] In particular, decorations of the constructor $\eta\, M\, i$ in a type family $X : \mathsf{Pos}\, M\, (\eta\, M\, i) \to \mathcal{U}$ are completely determined by a single element $x : X\, i$, a fact which we record in the following definition to reduce clutter below:

$$\eta\text{-dec}\, M\, X\, \{i\}\, x = \eta\text{-pos-elim}\, M\, i\, (\lambda\, \_ \to X i)\, x$$

Next, for the multiplication μ, our rules simply mimic the pairing and projections of the dependent sum. That is, we postulate an introduction rule

$$\begin{aligned}\mu\text{-pos} &: (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\}\ \{c : \mathsf{Cns}\, M\, i\}\\ &\to \{\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\, (\mathsf{Typ}\, M\, c\ p)\}\\ &\to (p : \mathsf{Pos}\, M\, c) \to (q : \mathsf{Pos}\, M\, (\delta\ p))\\ &\to \mathsf{Pos}\, M\, (\mu\ M\, c\ \delta)\end{aligned}$$

and elimination rules

$$\begin{aligned}\mu\text{-pos-fst} &: (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\}\ \{c : \mathsf{Cns}\, M\, i\}\\ &\to \{\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\, (\mathsf{Typ}\, M\, c\ p)\}\\ &\to \mathsf{Pos}\, M\ \ (\mu\ M\, c\ \delta) \to \mathsf{Pos}\, M\, c\\ \mu\text{-pos-snd} &: (M : \mathbb{M})\ \{i : \mathsf{Idx}\, M\}\ \{c : \mathsf{Cns}\, M\, i\}\\ &\to \{\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\, (\mathsf{Typ}\, M\, c\ p)\}\\ &\to (p : \mathsf{Pos}\, M\, (\mu\ M\, c\ \delta))\\ &\to \mathsf{Pos}\, M\, (\delta\ (\mu\text{-pos-fst}\, M\, p))\end{aligned}$$

with typing rule

$$\begin{aligned}&\mathsf{Typ}\, M\, (\mu\ M\, c\ \delta)\ p \rightsquigarrow \qquad \text{(Typ-}\mu\text{)}\\ &\qquad \mathsf{Typ}\, M\, (\delta\ (\mu\text{-pos-fst}\, M\, p))\\ &\qquad\qquad (\mu\text{-pos-snd}\, M\, p)\end{aligned}$$

and computation rules

$$\begin{aligned}\mu\text{-pos-fst}\, M\, (\mu\text{-pos}\, M\, p\ q) &\rightsquigarrow p \quad (\mu\text{-pos-fst})\\ \mu\text{-pos-snd}\, M\, (\mu\text{-pos}\, M\, p\ q) &\rightsquigarrow q \quad (\mu\text{-pos-snd})\\ \mu\text{-pos}\, M\, (\mu\text{-pos-fst}\, M\, p)\, (\mu\text{-pos-snd}\, M\, p) &\rightsquigarrow p \quad (\mu\text{-pos-}\eta)\end{aligned}$$

With the handling of positions in place, we can now state the unitality and associativity axioms for the monads in our universe. These take the form of reductions:

$$\begin{aligned}&\mu\ M\, c\ (\lambda\ p \to \eta\ M\, (\mathsf{Typ}\, M\, c\ p)) \rightsquigarrow c && (\mu\text{-}\eta\text{-r})\\ &\mu\ M\, (\eta\ M\, i)\ \delta \rightsquigarrow \delta\ (\eta\text{-pos}\, M\, i) && (\mu\text{-}\eta\text{-l})\\ &\mu\ M\, (\mu\ M\, c\ \delta)\ \epsilon \rightsquigarrow && (\mu\text{-}\mu)\\ &\quad \mu\ M\ c\ (\lambda\ p \to\ \ \mu\ M\, (\delta\ p)\ (\lambda\ q \to \epsilon\ (\mu\text{-pos}\, M\, p\ q)))\end{aligned}$$

Additionally, we must posit laws which assert that the constructors and eliminators for positions are compatible with these equations. We omit these for brevity, but the interested reader may consult the development for details.

While we will not undertake an extensive investigation of the meta-theoretic properties of our system in this article, we wish to pause briefly to make at least of few observations to justify its well-formedness. For example, there are critical pairs in the rewrite equations for the monad laws (between the first equation and the others) so we need to ensure confluence and termination.

Lemma 1 (Strong confluence for η and μ)**.** *The rewrite rules are strongly confluent [15], hence globally confluent.*

Proof. The rewrite system is strongly confluent using the rules (Typ-η) and (Typ-μ) and the associated reduction rules for μ-pos. We show the case for μ-η-r and μ-η-l. We omit M which is fixed here.

$$\begin{aligned}&\mu\ (\eta\ i)\ (\lambda\ p \to \eta\ (\mathsf{Typ}\, (\eta\ i)\ p)) \rightsquigarrow_{\mu\text{-}\eta\text{-r}} \eta\ i\\ &\mu\ (\eta\ i)\ (\lambda\ p \to \eta\ (\mathsf{Typ}\, (\eta\ i)\ p)) \rightsquigarrow_{\mu\text{-}\eta\text{-l}}\\ &\qquad (\lambda\ p \to \eta\ (\mathsf{Typ}\, (\eta\ i)\ p))\ (\eta\text{-pos}\, i) \rightsquigarrow_{\beta}\\ &\qquad \eta\ (\mathsf{Typ}\, (\eta\ i)\ (\eta\text{-pos}\, i)) \rightsquigarrow_{\text{Typ-}\eta} \eta\ i\end{aligned}$$

[3] In principle, we would also like to have an η-rule for the unit η, (that is, we would prefer the negative version as we have below for μ) but unfortunately this is not possible with the current implementation of rewriting in Agda.

The resolution of the μ-η-r/μ-μ pair can be found in the appendix of the extended version of this article [19, Lemma 3]. □

Proposition 1 (Termination of rewriting)**.** *All of the above rules form a terminating rewrite system.*

Proof. The μ-pos , η-pos and (Typ-η) rewrite rules are obviously terminating. For (Typ-μ), (μ-η-r), (μ-η-l) and (μ-μ), we need to use a dependency-pairs path ordering as introduced by [16] to verify termination. In particular for associativity, a lexicographic lifting of the subterm relation is not enough to verify (μ-μ)'s termination as we are going under binders and applying the δ and ϵ functions to subterms. This is a variant of the ordinal type eliminator proven to terminate in [16, Example 14, p11], which requires to ensure that the constructor types of our monads are inductively generated. All the monads considered in this article satisfy this. □

The instances of the μ and η operations for specific monads will themselves be defined by structural recursion on inductive datatypes and can be shown to respect the associativity and unitality laws prositionally. Results such as can be found in [5, Lemma 6.8], therefore, guarantee the consistency of the system. Furthermore, we conjecture that the rewrite system is strongly normalizing in conjunction with the definitional equality of Agda.

C. Populating the Universe

In the previous section, we described the generic structure associated to every monad $M : \mathbb{M}$. We now proceed to implement this structure in concrete cases, describing in each case the most salient features and omitting unnecessary details where we feel it will improve the presentation. Complete definitions can be found in the Agda formalization.

In the material which follows, we allow ourselves the freedom to use standard techniques such as inductive definitions and pattern matching during the definition of each monad. In practice, this agrees with the implementation: there, we first define all the necessary structure using ordinary Agda definitions and subsequently install rewrites which connect the decoding functions to their desired implementations. So for example, in order to define the indices of the identity monad (see below), we first make an ordinary Agda definition

$$\begin{aligned}&\mathsf{IdIdx} : \mathcal{U}\\ &\mathsf{IdIdx} = \top\end{aligned}$$

and then postulate the rewrite

$$\mathsf{Idx}\,\mathsf{Id} \rightsquigarrow \mathsf{IdIdx}$$

In the presentation which follows, we omit this auxiliary step and just write "=" when defining the structure associated to each monad.

1) The identity monad: We begin by adding a constant $\mathsf{Id} : \mathbb{M}$ to the universe to represent the *identity monad* (so named since its extension induces the identity monad on $\mathcal{U}$ up to equivalence). The polynomial part of Id decodes as follows:

$$\begin{aligned}\mathsf{Idx}\,\mathsf{Id} &= \top\\ \mathsf{Cns}\,\mathsf{Id}\,\mathsf{tt} &= \top\\ \mathsf{Pos}\,\mathsf{Id}\,\mathsf{tt} &= \top\\ \mathsf{Typ}\,\mathsf{Id}\,\mathsf{tt}\;\mathsf{tt} &= \mathsf{tt}\end{aligned}$$

Given the triviality of the associated polynomial, it is perhaps not surprising that its unit and multiplication are equally trivial. Indeed, they are given by:

$$\begin{aligned}&\eta\,\mathsf{Id}\,i = \mathsf{tt}\\ &\mu\,\mathsf{Id}\,\_\,\delta = \delta\,\mathsf{tt}\end{aligned}$$

We omit the remaining structure, which has a similar flavor.

2) The pullback monad: Our first monad constructor starts from a monad $M : \mathbb{M}$ and a family $X : \mathsf{Idx}\,M \to \mathcal{U}$ and refines the indices of M by additionally decorating the inputs and output of each constructor by elements of X. We refer to the resulting monad as the *pullback of M along X* (cf. [2, Section 2.4]). We implement this construction by first postulating a function

$$\mathsf{Pb} : (M : \mathbb{M})\,(X : \mathsf{Idx}\,M \to \mathcal{U}) \to \mathbb{M}$$

which adds the necessary code to our universe. We next define the polynomial part of $\mathsf{Pb}\,M\,X$ as follows:

$$\begin{aligned}&\mathsf{Idx}\,(\mathsf{Pb}\,M\,X) &&= \textstyle\sum_{(i:\mathsf{Idx}\,M)} X\ i\\ &\mathsf{Cns}\,(\mathsf{Pb}\,M\,X)\ (i,x) &&=\\ &&&\quad \textstyle\sum_{(c:\mathsf{Cns}\,M\,i)} \prod_{(p:\mathsf{Pos}\,M\,c)} X\,(\mathsf{Typ}\,M\,c\,p)\\ &\mathsf{Pos}\,(\mathsf{Pb}\,M\,X)\ (c,\nu) &&= \mathsf{Pos}\,M\,c\\ &\mathsf{Typ}\,(\mathsf{Pb}\,M\,X)\ (c,\nu)\ p &&= (\mathsf{Typ}\,M\,c\,p\,,\ \nu\ p)\end{aligned}$$

The unit for the pullback monad simply calls the unit of the underlying monad and decorates its input with the same value as its output:

$$\eta\ (\mathsf{Pb}\,M\,X)\ (i\,,x) = (\eta\ M\,i\,,\ \eta\text{-}\mathsf{dec}\,M\,X\,x)$$

As for the multiplication of the pullback monad, it again simply calls the multiplication of the underlying monad, this time decorating the result using the decorations of the second-level constructors, and forgetting the intermediate decoration. That is, we have

$$\mu\ (\mathsf{Pb}\,M\,X)\ (c\,,\nu)\ \delta = (\mu\ M\,c\ \delta'\,,\nu')$$

where

$$\begin{aligned}&\delta'\ p = \mathsf{fst}\ (\delta\,p)\\ &\nu'\ p = \mathsf{snd}\ (\delta\,(\mu\text{-}\mathsf{pos}\text{-}\mathsf{fst}\,p))\,(\delta\,(\mu\text{-}\mathsf{pos}\text{-}\mathsf{snd}\,p))\end{aligned}$$

The remaining structure is easily worked out from these definitions, and we omit the details.

3) The Slice Monad: The Baez-Dolan slice construction is at the heart of the opetopic approach: it is this construction which allows us to "raise the dimension" of the coherences in our algebraic structures. In our setting, it will take the form of a

monad constructor $\mathsf{Slice} : \mathbb{M} \to \mathbb{M}$. The basic intuition is that, for a monad $M : \mathbb{M}$, the monad $\mathsf{Slice}\, M$ may be described as the *monad of relations in* M. In order to realize this intuition, we have to find a way to encode the relations of M as some kind of data, just as the identity type encodes the relations in an ordinary type as data. This data will then serve as the constructors for the slice monad.

To begin, for a monad $M : \mathbb{M}$, let us define

$$\mathsf{Idx}\,(\mathsf{Slice}\, M) = \textstyle\sum_{(i:\mathsf{Idx}\, M)} \mathsf{Cns}\, M\, i$$

That is, the indices of the monad $\mathsf{Slice}\, M$ are exactly the constructors of the monad M. Next, we are going to capture the notion of *relation in* M with the help of a certain inductive family, defined as follows:

$$\begin{aligned}
&\texttt{data}\ \mathsf{Tree} : \mathsf{Idx}\,(\mathsf{Slice}\, M) \to \mathcal{U}\ \texttt{where}\\
&\quad \mathsf{lf}\ : (i : \mathsf{Idx}\, M) \to \mathsf{Tree}\ (i\,, \eta\ M\, i)\\
&\quad \mathsf{nd}\ : \{i : \mathsf{Idx}\, M\}\ (c : \mathsf{Cns}\, M\, i)\\
&\qquad \to (\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\, (\mathsf{Typ}\, M\, c\ p))\\
&\qquad \to (\epsilon : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Tree}\ (\mathsf{Typ}\, M\, c\ p, \delta\ p))\\
&\qquad \to \mathsf{Tree}\ (i\,, \mu\ M\, c\ \delta)
\end{aligned}$$

And we define $\mathsf{Cns}\,(\mathsf{Slice}\, M) = \mathsf{Tree}$.

The reader familiar with the theory of inductive types may recognize this as a modified form of the *indexed W-type* associated to a polynomial or indexed container. Here, as in that case, the elements of this type are *trees* generated by the constructors of the polynomial in question (the underlying polynomial of M, in the case at hand). The difference in the present setup is that our polynomial is equipped with a multiplication and unit, and we reflect this fact by indexing our trees not just over the indices (as is typically the case) but also over the constructors, applying the multiplication and unit as appropriate. The result is that we may view an element $\sigma : \mathsf{Cns}\,(\mathsf{Slice}\, M)\,(i\,, c)$ as "a tree generated by the constructors of M whose image under iterated multiplication is c". It is in this sense that this definition captures the *relations* in the original monad M.

We now turn to the rest of the structure required to complete the definition of $\mathsf{Slice}\, M$. Intuitively speaking, the positions of a tree σ will be its *internal nodes*. This can be accomplished by defining the positions by recursion on the constructors as follows:

$$\begin{aligned}
&\mathsf{Pos}\,(\mathsf{Slice}\, M)\,(\mathsf{lf}\ i) && = \bot\\
&\mathsf{Pos}\,(\mathsf{Slice}\, M)\,(\mathsf{nd}\ c\,\delta\,\epsilon) && =\\
&&& \top \sqcup \sum_{(p:\mathsf{Pos}\, M\, c)} \mathsf{Pos}\,(\mathsf{Slice}\, M)\,(\epsilon\, p)
\end{aligned}$$

In other words, if our tree is a leaf, it has no positions, and if it is a node, its positions consist of either the unit type (to record the current node) or else the choice of a position of the base constructor and, recursively, a node of the tree attached to that position.

Finally, the typing function $\mathsf{Typ}\,(\mathsf{Slice}\, M)\,\sigma\, p$ just projects out the constructor of M occurring at the node of σ specified by position p:

$$\begin{aligned}
&\mathsf{Typ}\,(\mathsf{Slice}\, M)\,(\mathsf{lf}\ i)\,()\\
&\mathsf{Typ}\,(\mathsf{Slice}\, M)\,(\mathsf{nd}\ \{i\}\, c\,\delta\,\epsilon)\,(\mathsf{inl}\,\mathsf{tt}) && = (i\,, c)\\
&\mathsf{Typ}\,(\mathsf{Slice}\, M)\,(\mathsf{nd}\ \{i\}\, c\,\delta\,\epsilon)\,(\mathsf{inl}\,(p\,, q)) && =\\
&&& \mathsf{Typ}\,(\mathsf{Slice}\, M)\,(\epsilon\, p)\, q
\end{aligned}$$

It remains to describe the unit and multiplication of the slice monad. In accordance with the general laws for monads, the unit constructor needs to have a unique position, and since the positions of a given tree are given by occurrences of constructors, this implies that the unit at a given constructor c should be the *corolla*, that is, a tree with one node consisting of c itself. Therefore we set:

$$\begin{aligned}
&\eta\,(\mathsf{Slice}\, M)\,(i\,, c) && =\\
&&& \mathsf{nd}\, c\,(\lambda\, p \to\ \eta\, M\,(\mathsf{Typ}\, M\, c\, p))\\
&&& \quad (\lambda\, p \to \mathsf{lf}\ (\mathsf{Typ}\, M\, c\, p))
\end{aligned}$$

Note that this definition would not be type correct without the assumption that M is definitionally right unital. A similar remark applies to the rest of the definitions of the slice monad in this section. Indeed, it is exactly the problem of completing the definition of the slice monad without any assumptions of truncation which motives the introduction of our monadic universe in the first place.

Let us now sketch the definition of the multiplication in the slice monad. As hypotheses, we are given a tree $\sigma : \mathsf{Cns}\,(\mathsf{Slice}\, M)(i\,, c)$ for some $i : \mathsf{Idx}\, M$ and $c : \mathsf{Cns}\, M\, i$, as well as a decoration

$$\phi : (p : \mathsf{Pos}\,(\mathsf{Slice}\, M)\,\sigma) \to \mathsf{Cns}\,(\mathsf{Slice}\, M)\,(\mathsf{Typ}\,(\mathsf{Slice}\, M)\,\sigma\, p)$$

In view of the preceding discussion, this means that ϕ assigns to every position of σ a tree which multiplies to the constructor which inhabits that position. The multiplication of $\mathsf{Slice}\, M$ may intuitively be described as "substituting" each of these trees into the node it decorates.

The definition of $\mu(\mathsf{Slice}\, M)$ will require an auxillary function γ with the following type:

$$\begin{aligned}
&\gamma : (M : \mathbb{M})\,\{i : \mathsf{Idx}\, M\}\,(c : \mathsf{Cns}\, M\, i)\\
&\quad \to (\sigma : \mathsf{Cns}\,(\mathsf{Slice}\, M)\,(i\,, c))\\
&\quad \to (\phi : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\,(\mathsf{Typ}\, M\, c\, p))\\
&\quad \to (\psi : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\,(\mathsf{Slice}\, M)\,(\mathsf{Typ}\, M\, c\, p\,, \phi\, p))\\
&\quad \to \mathsf{Cns}\,(\mathsf{Slice}\, M)(i\,, \mu\, M\, c\,\phi)
\end{aligned}$$

The intuition of this function is that γ *grafts* the tree specified by ψ onto the appropriate leaf of the tree σ (indeed, γ may be seen as an incarnation of multiplication in the *free* monad generated by the underlying polynomial of M). This function simply operates by induction and may be defined as follows:

$$\begin{aligned}
&\gamma\, M\,(\mathsf{lf}\ i)\,\delta\,\epsilon && = \epsilon\,(\eta\text{-}\mathsf{pos}\, M\, i)\\
&\gamma\, M\,(\mathsf{nd}\ c\,\delta\,\epsilon)\,\phi\,\psi && = \mathsf{nd}\ c\,\delta'\,\epsilon'
\end{aligned}$$

where we define

$$\phi'\, p\, q = \phi(\mu\text{-}\mathsf{pos}\, M\, c\,\delta\, p\, q)$$

$$\psi'\,p\,q = \psi(\mu\text{-pos}\,M\,c\,\delta\,p\,q)$$
$$\delta'\,p = \mu\,M\,(\delta\,p)\,(\phi' p)$$
$$\epsilon'\,p = \gamma\,M\,(\epsilon\,p)\,(\psi' p)$$

With this function in hand, we may complete the definition of the multiplication in the slice monad as

$$\mu\,(\mathsf{Slice}\,M)\,(\mathsf{lf}\ i)\,\phi = \mathsf{lf}\ i$$
$$\mu\,(\mathsf{Slice}\,M)\,(\mathsf{nd}\ c\,\delta\,\epsilon)\,\phi = \gamma\,M\,w\,\delta\,\psi$$

where we put

$$w = \phi\,(\mathsf{inl}\,\mathsf{tt})$$
$$\phi'\,p\,q = \phi\,(\mathsf{inr}\,(p\,,q))$$
$$\psi\,p = \mu\,(\mathsf{Slice}\,M)\,(\epsilon\,p)\,(\phi'\,p)$$

This definition then says that substitution is trivial on leaves, and when we are looking at a node, we first retrieve the tree living at this position (called w above), and then graft to it the result of recursively substituting in the remaining branches.

We refer the reader to the formalization for details on the remaining constructions handling positions in the slice monad.

D. Dependent monads

Since the notion of dependent type is one of the primitive aspects of Martin-Löf type theory, it is perhaps not surprising that we quickly find ourselves in need of a dependent version of our polynomial monads. We note there is a potential point of confusion here: while a dependent type can be thought of as a family of types dependent on a base type, a dependent monad in our sense is *not* a family of monads. Rather, it is a monad structure on dependent families of indices and constructors indexed over the indices and constructors of the base monad M. Put another way, under the equivalence between dependent types with *domain* A and functions with *codomain* A, our dependent monads over a base monad M correspond to monads M' equipped with a *cartesian homomorphism* to M. [4] The advantage of working in a dependent style, however, is that we do not need to axiomatize the notion of homomorphism using propositional equalities as it is encoded directly in the typing of the multiplication operator.

To begin, let us postulate, for each monad $M : \mathbb{M}$, a universe $\mathbb{M}{\downarrow}\,M : \mathcal{U}$ of *monads over* M.

$$\mathbb{M}{\downarrow} : \mathbb{M} \to \mathcal{U}$$

That is, for $M : \mathbb{M}$, the inhabitants of $\mathbb{M}{\downarrow}M$ are codes for monads equipped with a cartesian morphism to M. For this reason, when we are given a monad M and a dependent monad $M{\downarrow} : \mathbb{M}{\downarrow}\,M$, we often speak of the pair $(M\,, M{\downarrow})$ as a *monad extension*.

The decoding functions for dependent monads follow the same setup as the non-dependent case, simply repeating each of the definitions fiberwise. And since the dependent case resembles so closely the non-dependent one, we have attempted to systematically name dependent versions of the the monadic structure introduced above by appending a "↓" to the previously given name. For example, $\mathsf{Idx}{\downarrow}$ for the dependent version of the family Idx of indices.

As a first step, a dependent monad will be equipped with a polynomial lying over the base polynomial. This corresponds to the following three dependent families:

$$\mathsf{Idx}{\downarrow} : \{M : \mathbb{M}\} \to \mathbb{M}{\downarrow}\ M \to \mathsf{Idx}\ M \to \mathcal{U}$$
$$\mathsf{Cns}{\downarrow} : \{M : \mathbb{M}\}\ (M{\downarrow} : \mathbb{M}{\downarrow}\ M)\ \{i : \mathsf{Idx}\ M\} \to \mathsf{Idx}{\downarrow}\ M{\downarrow}\ i \to \mathsf{Cns}\ M\ i \to \mathcal{U}$$
$$\mathsf{Typ}{\downarrow} : \{M : \mathbb{M}\}\ (M{\downarrow} : \mathbb{M}{\downarrow}\ M) \to \{i : \mathsf{Idx}\ M\}\ \{i{\downarrow} : \mathsf{Idx}{\downarrow}\ M{\downarrow}\ i\} \to \{c : \mathsf{Cns}\ M\ i\}\ (c{\downarrow} : \mathsf{Cns}{\downarrow}\ M{\downarrow}\ i{\downarrow}\ c) \to \mathsf{Pos}\ M\ c \to \mathsf{Idx}{\downarrow}\ M{\downarrow}\ (\mathsf{Typ}\ M\ c\ p)$$

The reader will notice, however, that there is no analog of dependent positions. This is because we are modelling *cartesian* morphisms of monads, and therefore positions of a dependent constructor $c{\downarrow} : \mathsf{Cns}{\downarrow}\ M{\downarrow}\ i{\downarrow}\ c$ should be the same as those of the underlying constructor c. By working fiberwise, we can reflect this requirement directly in the type signature.

The monadic structure of a dependent monad simply operates fiberwise, following the multiplication in the base monad:

$$\eta{\downarrow} : \{M : \mathbb{M}\}\ (M{\downarrow} : \mathbb{M}{\downarrow}\ M) \to \{i : \mathsf{Idx}\ M\} \to \mathsf{Idx}{\downarrow}\ M{\downarrow}\ i \to \mathsf{Cns}{\downarrow}\ M{\downarrow}\ i\,(\eta\,M\,i)$$
$$\mu{\downarrow} : \{M : \mathbb{M}\}\ (M{\downarrow} : \mathbb{M}{\downarrow}\ M) \to \{i : \mathsf{Idx}\ M\}\ \{c : \mathsf{Cns}\ M\ i\} \to \{\delta : (p : \mathsf{Pos}\ M\ c) \to \mathsf{Cns}\ M\ (\mathsf{Typ}\ M\ c\ p)\} \to (i{\downarrow} : \mathsf{Idx}{\downarrow}\ M{\downarrow}\ i)\ (c{\downarrow} : \mathsf{Cns}{\downarrow}\ M{\downarrow}\ i{\downarrow}\ c) \to (\delta{\downarrow} : (p : \mathsf{Pos}\ M\ c) \to \mathsf{Cns}{\downarrow}\ M{\downarrow}\ (\mathsf{Typ}{\downarrow}\ M{\downarrow}\ c{\downarrow}\ p)\ (\delta\ p)) \to \mathsf{Cns}{\downarrow}\ M{\downarrow}\ i{\downarrow}\ (\mu\ M\ c\ \delta)$$

The fact that we require the multiplication of a family of dependent constructors to live over the multiplication of the base constructors (and similarly for the unit) is what guarantees the homomorphism property alluded to above.

Our dependent monads must also be equipped with equational laws making them compatible with the corresponding laws of the monads they live over. For example, the typing functions for $\eta{\downarrow}$ and $\mu{\downarrow}$ respect the indices of parameters, just as in the base case:

$$\mathsf{Typ}{\downarrow}\ M{\downarrow}\ (\eta{\downarrow}\ M{\downarrow}\ i{\downarrow})\ p \rightsquigarrow i{\downarrow}$$
$$\mathsf{Typ}{\downarrow}\ M{\downarrow}\ (\mu{\downarrow}\ M{\downarrow}\ c{\downarrow}\ \delta{\downarrow})\ p \rightsquigarrow \mathsf{Typ}{\downarrow}\ M\ (\delta{\downarrow}\ (\mu\text{-pos-fst}\downarrow M{\downarrow}\ p))\ (\mu\text{-pos-snd}\downarrow M{\downarrow}\ p)$$

[4] In fact, it is entirely possible to add a monadic form of dependent sum to the list of monad constructors of the universe $\mathbb{M}$ so that this statement becomes literally true. As we will not have need of this construction in the present article, however, we omit the details.

There are similar laws asserting the definitional associativity and unitality of the multiplication, but as these all follow exactly the same pattern, we omit the details here and refer the curious reader to the implementation.

We remark that, because their positions are the same, decorations of the dependent constructor $\eta\downarrow$ are essentially constant just as in the case of η, and there is therefore an analogous function η-dec$\downarrow$ generating such decorations from a single piece of data with a similar definition. This function occurs occasionally in the code below.

E. Populating the dependent universe

We now quickly describe dependent counterparts of the base monads and monad constructors of the previous section. As most of the definitions are routine and easily deduced from the absolute case, the presentation here is brief.

1) The identity monad: The dependent identity monad is parametrized by a type $A : \mathcal{U}$ and indexed over the identity monad Id. That is, we have a dependent monad constructor of the form

$$\mathsf{Id}\downarrow : \mathcal{U} \to \mathbb{M}\downarrow\, \mathsf{Id}$$

Its polynomial part is defined by

$$\begin{aligned}
&\mathsf{Idx}\downarrow\,(\mathsf{Id}\downarrow\, A)\,\mathsf{tt} && = A\\
&\mathsf{Cns}\downarrow\,(\mathsf{Id}\downarrow\, A)\,x\,\mathsf{tt} && = \top\\
&\mathsf{Pos}\downarrow\,(\mathsf{Id}\downarrow\, A)\,\mathsf{tt}\,\mathsf{tt} && = \top\\
&\mathsf{Typ}\downarrow\,(\mathsf{Id}\downarrow\, A)\,\{i\downarrow = x\}\ \mathsf{tt}\ \mathsf{tt} && = x
\end{aligned}$$

As in the base case, the multiplication and unit all take values in the unit type, making the structure essentially trivial.

2) The dependent pullback monad: Just as we can refine the indices of a base monad, so the dependent pullback monad allows us to refine the indices of a dependent monad. Its constructor takes the form

$$\begin{aligned}
&\mathsf{Pb}\downarrow : \{M : \mathbb{M}\}\ (M\downarrow : \mathbb{M}\downarrow\, M)\ \{X : \mathsf{Idx}\ \ M \to \mathcal{U}\}\\
&\quad \to (X\downarrow : \{i : \mathsf{Idx}\ \ M\} \to \mathsf{Idx}\downarrow\ \ M\downarrow\, i \to X\ \ i \to \mathcal{U})\\
&\quad \to \mathbb{M}\downarrow\ (\mathsf{Pb}\ \ M\ \ X)
\end{aligned}$$

Note that the family $X\downarrow$ may also depend on elements of the refining family X for the base monad. The underlying polynomial of the dependent pullback is then defined as follows:

$$\mathsf{Idx}\downarrow\ (\mathsf{Pb}\downarrow M\downarrow\, X\downarrow)\ (i,x) = \sum_{(i\downarrow : \mathsf{Idx}\downarrow\ M\downarrow\, i)} X\downarrow\ i\downarrow\ x$$

$$\mathsf{Cns}\downarrow\ (\mathsf{Pb}\downarrow M\downarrow\, X\downarrow)\ (i\downarrow, x\downarrow)\ (c,\nu) =$$

$$\sum_{(c\downarrow : \mathsf{Cns}\downarrow\ M\downarrow\, i\downarrow\, c)}\ \prod_{(p : \mathsf{Pos}\, M\, c)} X\downarrow\,(\mathsf{Typ}\downarrow M\downarrow c\downarrow p)\,(\nu\, p)$$

$$\mathsf{Typ}\downarrow\ (\mathsf{Pb}\downarrow M\downarrow X\downarrow)\ (c\downarrow, \nu\downarrow)\ p = \mathsf{Typ}\downarrow\ \ M\, c\downarrow p, \nu\downarrow p$$

with multiplicative structure following fiberwise the rules for the base pullback $\mathsf{Pb}\, M\, X$.

3) The dependent slice monad: Finally, the dependent slice monad extends the Baez-Dolan slice construction to the dependent case. Its monad constructor is typed as follows:

$$\mathsf{Slice}\downarrow\ : \{M : \mathbb{M}\}\,(M\downarrow : \mathbb{M}\downarrow\, M) \to \mathbb{M}\downarrow\ (\mathsf{Slice}\, M)$$

As for the absolute case, the indices are given by the dependent constructors. That is, we set

$$\mathsf{Idx}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)\ (i,c) = \sum_{i\downarrow : \mathsf{Idx}\downarrow\, M\downarrow\, i} \mathsf{Cns}\downarrow M\downarrow i\downarrow c$$

Similarly, the type of constructors $\mathsf{Cns}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)$ are trees lying over a tree in the base. This corresponds to the following (rather verbose) inductive type:

$$\begin{aligned}
&\texttt{data}\ \ \mathsf{Tree}\downarrow : \{i : \mathsf{Idx}\,(\mathsf{Slice}\, M)\} \to (i\downarrow : \mathsf{Idx}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)\\
&\qquad \to \mathsf{Tree}\, i \to \mathcal{U}\ \ \texttt{where}\\
&\mathrm{lf}\downarrow : \{i : \mathsf{Idx}\, M\}\ (i\downarrow : \mathsf{Idx}\downarrow\, M\downarrow i)\\
&\quad \to \mathsf{Cns}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)\,(i\downarrow, \eta\downarrow M\downarrow i\downarrow)\,(\mathsf{lf}\, i)\\
&\mathrm{nd}\downarrow : \{i : \mathsf{Idx}\, M\}\ \{c : \mathsf{Cns}\, M\, i\}\\
&\quad \to \{\delta : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M\,(\mathsf{Typ}\, M\, c\, p)\}\\
&\quad \to \{\epsilon : (p : \mathsf{Pos}\, M\, c)\\
&\qquad \to \mathsf{Cns}\,(\mathsf{Slice}\, M)\,(\mathsf{Typ}\, M\, c\, p, \delta\, p)\}\\
&\quad \to \{i\downarrow : \mathsf{Idx}\downarrow M\downarrow i\} \to (c\downarrow : \mathsf{Cns}\downarrow M\downarrow i\downarrow c)\\
&\quad \to (\delta\downarrow : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\downarrow M\downarrow (\mathsf{Typ}\downarrow M\downarrow c\downarrow p))\\
&\quad \to (\epsilon\downarrow : (p : \mathsf{Pos}\, M\, c)\\
&\qquad \to \mathsf{Cns}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)\,(\mathsf{Typ}\downarrow M\downarrow c\downarrow p, \delta\downarrow p))\\
&\quad \to \mathsf{Cns}\downarrow\,(\mathsf{Slice}\downarrow M\downarrow)\,(i\downarrow, \mu\downarrow M\downarrow c\downarrow \delta\downarrow)\,(\mathsf{nd}\, c\, \delta\, \epsilon)
\end{aligned}$$

The rest of the description of the dependent slice follows exactly the same pattern: duplicating the definitions and laws of the base case routinely in each fiber.

III. Opetopic Types

In this section, we show how to use the universes introduced above in order to implement Baez and Dolan's definition of *opetopic type* [2]. We go on to explain how to use this definition to capture the notion of *weak M-algebra*, and finish with some examples.

Definition 1. *An **opetopic type over a monad** M is defined coninductively as follow:*

$$\begin{aligned}
&\texttt{record}\ \ \mathsf{OpetopicType}\,(M : \mathbb{M}) : \mathcal{U}_1\ \ \texttt{where}\\
&\quad \mathcal{C}\ : \mathsf{Idx}\, M \to \mathcal{U}\\
&\quad \mathcal{R}\ : \mathsf{OpetopicType}\,(\mathsf{Slice}\,(\mathsf{Pb}\, M\, \mathcal{C}\,))
\end{aligned}$$

We see from the definition that an opetopic type consists of an infinite sequence of dependent families

$$\mathcal{C}\, X\,, \mathcal{C}\,(\mathcal{R}\, X)\,, \mathcal{C}\,(\mathcal{R}\,(\mathcal{R}\, X))\,, \ldots$$

whose domain is the set of indices of a monad whose definition incorporates all the previous families in the sequence. Given an opetopic type $X : \mathsf{OpetopicType}\, M$, we will often denote this sequence of dependent types more succinctly as just $X_0, X_1, X_2, \ldots$ since the destructor notation quickly becomes

quite heavy. We will use a similar convention for the series of monads $M = M_0, M_1, M_2 \ldots$ generated by the definition. That is, we have:

$$\begin{array}{ll} M_0 = M & X_0 = \mathcal{C}\, X : \mathsf{Idx}\, M \to \mathcal{U} \\ M_1 = \mathsf{Slice}\,(\mathsf{Pb}\, M_0\, X_0) & X_1 = \mathcal{C}\,(\mathcal{R}\, X) : \mathsf{Idx}\, M_1 \to \mathcal{U} \\ M_2 = \mathsf{Slice}\,(\mathsf{Pb}\, M_1\, X_1) & X_2 = \mathcal{C}\,(\mathcal{R}\,(\mathcal{R}\, X)) : \\ & \qquad \mathsf{Idx}\, M_2 \to \mathcal{U} \\ \vdots & \vdots \end{array} \tag{1}$$

Before describing the connection between opetopic types and weak M-algebras, let us give some examples of how to think of the resulting dependent families as "fillers" for a collection of "shapes" generated by the monad M. For concreteness, we will fix $M = \mathsf{Id}$ in our examples. Given $X : \mathsf{OpetopicType}\,\mathsf{Id}$, we can define the type of *objects* of X as simply

$$\begin{array}{l} \mathsf{Obj} : \mathcal{U} \\ \mathsf{Obj} = \mathcal{C}\, X\, \mathsf{tt} \end{array} \tag{2}$$

Next, after a single slice, X provides us with a type of *arrows* between any two objects which can be defined as follows:

$$\begin{array}{l} \mathsf{Arrow} : (x\, y : \mathsf{Obj}) \to \mathcal{U} \\ \mathsf{Arrow}\, x\, y = \mathcal{C}\,(\mathcal{R}\, X) \\ \qquad\qquad ((\mathsf{tt}\,, y)\,, (\mathsf{tt}, \eta\text{-}\mathsf{dec}\,\mathsf{Id}\,(\mathcal{C}\, X)\, x)) \end{array} \tag{3}$$

Furthermore, for a *loop* f in X, that is, an arrow with the same domain and codomain, X includes a family whose elements can be thought of as "null-homotopies of f", and which is defined by

$$\begin{array}{l} \mathsf{Null} : (x : \mathsf{Obj})\,(f : \mathsf{Arrow}\, x\, x) \to \mathcal{U} \\ \mathsf{Null}\, x\, f = \mathcal{C}\,(\mathcal{R}\,(\mathcal{R}\, X)) \\ \qquad\qquad ((((\mathsf{tt}\,, x)\,, (\mathsf{tt}, \eta\text{-}\mathsf{dec}\,\mathsf{Id}\,(\mathcal{C}\, X)\, x))\,, f)\,, \\ \qquad\qquad\quad \mathsf{If}\ (\mathsf{tt}\,, x)\,, \bot\text{-}\mathsf{elim}) \end{array}$$

More examples of shapes and filling families may be found in the development.

A. Weak Algebras and Fibrant Opetopic Types

We now wish to describe how an opetopic type $X : \mathsf{OpetopicType}\, M$ encodes the structure of a weak M-algebra. Before we begin, it will be convenient to adopt the following convention: recall that X consists of an infinite sequence of dependent types following the form of Equation 1. In the discussion which follows, instead of working with a fixed opetopic type X, we will rather just work with abstract type families $X_0, X_1, \ldots$ over monads $M = M_0, M_1, \ldots$ following the same pattern of dependencies. We will then freely add new families of the form X_i to our hypotheses as they become necessary. The advantage of working this way is that our definitions are parameterized over just that portion of the opetopic type which is necessary, as opposed to depending on the entire opetopic type X itself, and consequently, we will be able to reuse our definitions and constructions starting at any point in the infinite sequence generated by X.

We recall that for M a polynomial monad, an *M-algebra* consists of a *carrier family* $C : \mathsf{Idx}\, M \to \mathcal{U}$ together with a map

$$\alpha : \{i : \mathsf{Idx}\, M\} \to [M]\, C\, i \to C\, i$$

which satisfies some equations expressing the compatibility of α with the multiplication of M. Indeed, it is the need for a complete description of these equations in all dimensions which motivates the present work. Now, clearly the first dependent type $X_0 : \mathsf{Idx}\, M \to \mathcal{U}$ may serve as a carrier for an M-algebra structure. Let us now see what else this sequence of families provides us with.

After one iteration, we obtain a type family $X_1 : \mathsf{Idx}\, M_1 \to \mathcal{U}$, and unfolding the definition of the indices of the slice and pullback monads, we find that the domain of X_1 takes the form

$$\sum_{(i:\mathsf{Idx}\, M)} \sum_{(x:X_0 i)} \sum_{(c:\mathsf{Cns}\, M\ i)} (p : \mathsf{Pos}\, M\, c) \to X_0\,(\mathsf{Typ}\, M\, c\, p)$$

The elements of this type are 4-tuples $(i\,, x\,, c\,, \nu)$, and we now observe that the three elements i, c and ν are typed such that they are exactly the arguments of the hypothetical algebra map α introduced above. We may regard the family X_1, therefore, as a relation between triples $(i\,, c\,, \nu)$ and elements $x : X_0\, i$, and in order to define a map α, we only need to impose that this relation is functional in the sense that there is a *unique* x determined by any such triple. When this is the case, we will say that the family X_1 is *multiplicative*. That is, we define:

$$\begin{array}{l} \mathsf{is\text{-}mult}\ : \{X_0 : \mathsf{Idx}\, M_0 \to \mathcal{U}\}\,(X_1 : \mathsf{Idx}\, M_1 \to \mathcal{U}) \to \mathcal{U} \\ \mathsf{is\text{-}mult}\,\{X_0\}\, X_1 = \{i : \mathsf{Idx}\, M\}\,(c : \mathsf{Cns}\, M\, i) \\ \qquad \to (\nu : (p : \mathsf{Pos}\, M\, c) \to X_0\,(\mathsf{Typ}\, M\, c\, p)) \\ \qquad \to \mathsf{is\text{-}contr}\,(\sum_{x:X_0\, i} X_1\,(i\,, x\,, c\,, \nu)) \end{array}$$

Supposing we are given a proof $m_1 : \mathsf{is\text{-}mult}\, X_1$, we can define an algebra map α as above by

$$\alpha\,(c\,, \nu) = \mathsf{fst}\,(\mathsf{ctr}\,(m_1\, c\, \nu))$$

Furthermore, we will write

$$\alpha\text{-}\mathsf{wit}\,(c\,, \nu) = \mathsf{snd}\,(\mathsf{ctr}\,(m_1\, c\, \nu))$$

for the associated element of the relation $X_1(i\,, \alpha\,(c\,, \nu)\,, c\,, \nu)$ which witnesses this multiplication.

Let us now suppose that our sequence extends one step further, that is, that we are given a type family $X_2 : \mathsf{Idx}\, M_1 \to \mathcal{U}$ and a proof $m_2 : \mathsf{is\text{-}mult}\, X_2$. We now show how to use this further structure to derive some of the expected *laws* for the algebra map α we have just defined. As a first example, we expect α to satisfy a unit law: decorating a unit constructor with some element x and then applying α should return the element x itself. In other words, we expect to be able to prove

$$\begin{array}{l} \alpha^{\eta}\text{-}\mathsf{coh} : \{i : \mathsf{Idx}\, M\}(x : X_0\, i) \\ \qquad \to \alpha\,(\eta\, M\, i\,, \eta\text{-}\mathsf{dec}\, M\, X_0\, x) \equiv x \end{array}$$

To prove this equality, let us define the following function:

$$\begin{aligned}&\eta\text{-alg}_{m_2} : \{i : \mathsf{Idx}\, M\}(x : X_0\, i)\\ &\quad \to X_1\,((i\,,x)\,,(\eta\, M\, i, \eta\text{-dec}\, M\, X_0\, x))\\ &\eta\text{-alg}_{m_2} = \mathsf{fst}\,(\mathsf{ctr}\,(m_2\,(\mathsf{lf}\,(i\,,x))\,\bot\text{-elim}))\end{aligned}$$

Now we simply notice that the pairs

$$\mathsf{ctr}\,(m_1\,(\eta\, M\, i)\,(\eta\text{-dec}\, M\, X_0\, x)) \equiv (x\,, \eta\text{-alg}\, x)$$

must be equal as indicated, since they inhabit a contractible space. Projecting on the first factor gives exactly the desired equation.

We also expect our algebra map α to satisfy an equation expressing its compatibility with multiplication of the following form:

$$\begin{aligned}&\alpha^{\mu}\text{-coh} : \{i' : \mathsf{Idx}\, M\}\,(c' : \mathsf{Cns}\, M\, i)\\ &\quad \to (\delta' : (p : \mathsf{Pos}\, M\, c) \to \mathsf{Cns}\, M(\mathsf{Typ}\, M\, c\, p))\\ &\quad \to (\nu' : (p : \mathsf{Pos}\, M\, c')(q : \mathsf{Pos}\, M\, (\delta'\, p))\\ &\qquad\qquad \to X_0\,(\mathsf{Typ}\, M\ \ (\delta'\, p)\, q))\\ &\quad \to \alpha\,(\mu\, M\, c'\, \delta')\,(\lambda p \to \nu'\,(\mu\text{-pos-fst}\, p)\,(\mu\text{-pos-snd}\, p)) \equiv\\ &\qquad \alpha\, c'\,(\lambda p \to \alpha\,(\delta'\, p)\,(\nu'\, p))\end{aligned}$$

We note that this equation is simply the type theoretic translation of the familiar commutative diagram

$$\begin{array}{ccc} [M]\,[M]\,X_0 & \xrightarrow{\mu_{X_0}} & [M]\,X_0 \\ {\scriptstyle [M]\,\alpha}\downarrow & & \downarrow{\scriptstyle \alpha} \\ [M]\,X_0 & \xrightarrow[\alpha]{} & X_0 \end{array}$$

To prove this axiom, we use m_2 to define the following multiplication operation on elements of the family X_1:

$$\begin{aligned}&\mu\text{-alg}_{m_2} : \{i : \mathsf{Idx}\, M\}\,(c : \mathsf{Cns}\, M\, i)\\ &\quad \to (\nu : (p : \mathsf{Pos}\, Mc) \to X_0\,(\mathsf{Typ}\, M\, c\, p))\\ &\quad \to (\delta : (p : \mathsf{Pos}\, Mc)\\ &\qquad\qquad \to \mathsf{Cns}\,(\mathsf{Pb}\, M\, X_0)(\mathsf{Typ}\,(\mathsf{Pb}\, M\, X_0)\,(c,\nu)\, p))\\ &\quad \to (x_0 : X_0\, i)\,(x_1 : X_1\,(i\,, x_0\,, c\,, \nu))\\ &\quad \to (\bar{x} : (p : \mathsf{Pos}\, M\, c) \to X_1(\mathsf{Typ}\,(\mathsf{Pb}\, M\, X_0)(c\,,\nu)\,, \delta\, p))\\ &\quad \to X_1(i\,, x_0\,, \mu\,(\mathsf{Pb}\, M\, X_0)\,(c\,,\nu)\,\delta)\\ &\mu\text{-alg}_{m_2} = \mathsf{fst}\,(\mathsf{ctr}\,(m_2\,\sigma\,\theta))\end{aligned}$$

where

$$\sigma = \mathsf{nd}\,(c\,,\nu)\,\delta\,(\lambda p \to \eta\, M_1\,((\mathsf{Typ}\, M\, c\, p\,, \nu\, p)\,, \delta\, p))$$

is the two-level tree consisting of a base node (c, ν), as well as a second level of constructors specified by the decoration δ, and θ is the decoration of the nodes of σ by elements of X_1 defined by:

$$\begin{aligned}\theta\,(\mathsf{inl}\,\mathsf{tt}) &= x_1\\ \theta\,(\mathsf{inr}\,(p\,, \mathsf{inl}\,\mathsf{tt})) &= \bar{x}\, p\end{aligned}$$

Now instantiating our function $\mu\text{-alg}_{m_2}$ with arguments

$$\begin{aligned} c &= c' & x_1 &= \alpha\text{-wit}\,(c\,,\nu)\\ \nu\, p &= \alpha\,(\delta'\, p\,, \nu'\, p) & \delta\, p &= (\delta'\, p\,, \nu'\, p)\\ x_0 &= \alpha\,(c\,,\nu) & \bar{x}\, p &= \alpha\text{-wit}\,(\delta'\, p\,, \nu'\, p)\end{aligned}$$

we find that the pairs

$$\begin{aligned}&\mathsf{ctr}\,(m_1\,(\mu\, M\, c'\, \delta')\,(\lambda p \to \nu'\,(\mu\text{-pos-fst}\, p)\,(\mu\text{-pos-snd}\, p))) \equiv\\ &\quad (\alpha\,(c\,,\nu)\,, \mu\text{-alg}_{m_2}\, c\, \nu\, x_0\, x_1\, \delta\, \bar{x})\end{aligned}$$

again inhabit a contractible space, whereby their first components are equal, giving the desired equation.

We may think of the functions $\eta\text{-alg}_{m_2}$ and $\mu\text{-alg}_{m_2}$ defined above as the nullary and binary cases of a multiplicative operation on the *relations* of our algebra structure. The key insight, as we have seen, is that this multiplicative structure encodes exactly the *laws* for the algebra map α defined one level lower. Similarly, if we are able to extend our sequence on *further* step to a family X_3 which is itself multiplicative, then we will be able to show that the operations $\eta\text{-alg}_{m_2}$ and $\mu\text{-alg}_{m_2}$ *themselves satisfy unit and associativity laws*, and this in turn encodes the "2-associativity" and "2-unitality" of the algebra map α. This motivates the following definition:

Definition 2. *An opetopic type X over a monad M is said to be **fibrant** if we are given an element of the following coinductively defined property:*

$$\begin{aligned}&\texttt{record}\ \mathsf{is\text{-}fibrant}\,\{M : \mathbb{M}\}\,(X : \mathsf{OpetopicType}\, M) : \mathcal{U}\\ &\quad \texttt{where}\\ &\qquad \mathsf{car\text{-}is\text{-}mult}\ : \mathsf{is\text{-}mult}\, M\,(\mathcal{C}\,(\mathcal{R}\, X))\\ &\qquad \mathsf{rel\text{-}is\text{-}fibrant}\ : \mathsf{is\text{-}fibrant}\,(\mathcal{R}\, X)\end{aligned}$$

Fibrant opetopic types, therefore, are our definition of infinitely coherent M-algebras, with the multiplicativity of the relations further in the sequence witnessing the higher dimensional laws satisfied by the structure earlier in the sequence.

B. Higher structures

We now use the preceding notions to define a number of coherent algebraic structures. A first example is that we obtain an internal definition of the notion of ∞-groupoid as follows:

Definition 3. *An ∞-**groupoid** is a fibrant opetopic type over the identity monad. That is,*

$$\infty\text{-Grp} = \textstyle\sum_{(X : \mathsf{OpetopicType}\,\mathsf{Id})}\, \mathsf{is\text{-}fibrant}\, X$$

We will attempt to justify the correctness of this definition in the sections which follow.

Next, it happens that the monad $\mathsf{Slice}\,\mathsf{Id}$ is in fact the monad whose algebras are monoids, and consequently, our setup leads naturally to the definition of an $\mathbb{A}_\infty$-type, that is, a type with a coherently associative binary operation.

Definition 4. *An $\mathbb{A}_\infty$-**type** is a fibrant opetopic type over the first slice of the identity monad.*

$$\mathbb{A}_\infty\text{-type} = \textstyle\sum_{(X : \mathsf{OpetopicType}\,(\mathsf{Slice}\,\mathsf{Id}))}\, \mathsf{is\text{-}fibrant}\, X$$

Furthermore, the notion of $\mathbb{A}_\infty$-group can now be defined by imposing an invertibility axiom. A classical theorem of homotopy theory asserts that the type of $\mathbb{A}_\infty$-groups is equivalent to the type of pointed, connected spaces via the loop-space construction. It would be interesting to see if the techniques of this article lead to a proof of this fact in type theory.

The notion of ∞-category can also be defined using this setup. Recall that an opetopic type over the identity monad Id has both a type of objects and a type of arrows (Equations 2 and 3). In the definition of ∞-groupoid above, the invertibility of the arrows in the underlying opetopic type is a consequence of the fact that the family of arrows is assumed to be multiplicative. Consequently, we obtain a reasonable notion of a *pre-∞-category* by simply dropping this assumption, and only requiring fibrancy after one application of the destructor $\mathcal{R}$:

$$\mathsf{pre\text{-}\infty\text{-}Cat} = \textstyle\sum_{(X : \mathsf{OpetopicType\ Id})} \mathsf{is\text{-}fibrant}\,(\mathcal{R}\,X)$$

The prefix "pre" here refers to the fact that this definition is missing a completeness axiom asserting that the invertible arrows coincide with paths in the space of objects, that is, an axiom of *univalence* in the sense of [17]. Such an axiom is easily worked out in the present setting, but as it would distract us slightly from the main objective of the present work, we will not pursue the matter here.

IV. The ∞-groupoid associated to a type

In this section, we use the machinery we have set up to produce an ∞-groupoid associated to any type and eventually prove it is unique. As a first step, we will need a source of opetopic types. Here is where the notion of dependent monad becomes important: we now show that every dependent monad gives rise to an opetopic type. The reason for this phenomenon is simple: since our dependent monad constructors mirror the monad constructors of the absolute case, any monad extension $(M\,, M{\downarrow})$ in fact gives rise to a *new* monad extension as follows:

$$\begin{aligned} M &\mapsto \mathsf{Slice}\,(\mathsf{Pb}\,M\,(\mathsf{Idx}{\downarrow}\,M{\downarrow})) \\ M{\downarrow} &\mapsto \mathsf{Slice}{\downarrow}\,(\mathsf{Pb}{\downarrow}\,M{\downarrow}\,(\lambda\,j\,k \to j \equiv k)) \end{aligned}$$

Notice how by pulling back along $\mathsf{Idx}{\downarrow}\,M{\downarrow}$, the identity type gives us a canonical family along which to apply the $\mathsf{Pb}{\downarrow}$ constructor. Iterating this construction, then, we find that associated to every monad extension $(M\,, M{\downarrow})$, is an infinite sequence

$$(M\,, M{\downarrow}) = (M_0\,, M{\downarrow}_0), (M_1\,, M{\downarrow}_1), (M_2\,, M{\downarrow}_2), \ldots$$

where $(M_{i+1}\,, M{\downarrow}_{i+1})$ is obtained from $(M_i\,, M{\downarrow}_i)$ by the above transformation.

The above construction provides us with our desired source of opetopic types. Formally, we define (using copattern notation)

$$\begin{aligned} &{\downarrow}\mathsf{OpType}\,M\,M{\downarrow} : \mathsf{OpetopicType}\,M \\ &\mathcal{C}\,({\downarrow}\mathsf{OpType}\,M\,M{\downarrow}) = \mathsf{Idx}{\downarrow}\,M{\downarrow} \\ &\mathcal{R}\,({\downarrow}\mathsf{OpType}\,M\,M{\downarrow}) = \\ &\qquad {\downarrow}\mathsf{OpType}\,(\mathsf{Slice}\,(\mathsf{Pb}\,(\mathsf{Idx}{\downarrow}\,M{\downarrow}))) \\ &\qquad\quad (\mathsf{Slice}{\downarrow}\,(\mathsf{Pb}{\downarrow}\,M{\downarrow}\,(\lambda\,j\,k \to j \equiv k))) \end{aligned}$$

Specializing to the case of the identity monad, we obtain the following:

Definition 5. *For a type $A : \mathcal{U}$, the **underlying opetopic type of** A is defined to be the opetopic type associated to the dependent identity monad determined by A. That is, the opetopic type*

$${\downarrow}\mathsf{OpType}\,\mathsf{Id}\,(\mathsf{Id}{\downarrow}\,A)$$

in the notation of the previous paragraph.

In order to show that every type A determines an ∞-groupoid in our sense, our next task is to show that this opetopic type is in fact fibrant.

A. Algebraic Extensions

Let $M : \mathbb{M}$ and $M{\downarrow} : \mathbb{M}{\downarrow}$. We will say that the extension $(M, M{\downarrow})$ is *algebraic* if we have a proof

$$\begin{aligned} &\mathsf{is\text{-}algebraic} : (M : \mathbb{M})\,(M{\downarrow} : \mathbb{M}{\downarrow}) \to \mathcal{U} \\ &\mathsf{is\text{-}algebraic} = \{i : \mathsf{Idx}\,M\}\,(c : \mathsf{Cns}\,M\,i) \\ &\quad \to (\nu : (p : \mathsf{Pos}\,M\,c) \to \mathsf{Idx}{\downarrow}\,M{\downarrow}\,(\mathsf{Typ}\,M\,c\,p)) \\ &\quad \to \mathsf{is\text{-}contr}\left(\sum_{(i{\downarrow} : \mathsf{Idx}{\downarrow}\,M{\downarrow})}\ \sum_{(c{\downarrow} : \mathsf{Cns}{\downarrow}\,M{\downarrow}\,i{\downarrow})} \mathsf{Typ}{\downarrow}\,M{\downarrow}\,c{\downarrow} \equiv \nu\right) \end{aligned}$$

An algebraic extension should be thought of as roughly analogous to a generalized kind of opfibration: if we think of the constructors as generalized arrows between their input indices and output, then the hypothesis says we know a family of lifts over the source of our constructor, and the conclusion is that there exists a unique "pushforward" consisting of a lift over the output as well as a constructor connecting the two whose typing function agrees with the provided input lifts. Such a hypothesis is one way of encoding an M-algebra, which motivates the name for this property. See [18, Section 6.3].

The main use of the notion of algebraic extension is the following lemma, whose proof is entirely straightforward:

Lemma 2. *Suppose the pair $(M\,, M{\downarrow})$ is an algebraic extension. Then the relation $\mathsf{Idx}{\downarrow}\,M{\downarrow}_1$ is multiplicative.*

Consequently, just as dependent monads are a source of opetopic types, algebraic extensions can be thought of as a source of multiplicative relations. Hence if we want to prove fibrancy of the opetopic type associated to a monad extension, we will need to know which of the extensions in the generated sequence are algebraic. Our main theorem is that after a single iteration of the slice construction, *every* monad extension becomes algebraic. That is

Theorem 1. *Let $(M\,, M{\downarrow})$ be a monad extension. Then slice extension $(M_1\,, M{\downarrow}_1)$ is algebraic.*

A proof can be found in the extended version of this article [19]. The importance of the theorem is that it has the following immediate corollaries:

Corollary 1. *Let $(M\,, M{\downarrow})$ be an algebraic extension. Then the opetopic type $\downarrow\mathsf{OpType}\,M\,M{\downarrow}$ is fibrant.*

Proof. The base case of the coinduction is Lemma 2 and the coinductive case is covered by Theorem 1. □

Corollary 2. *There is a map $\Gamma : \mathcal{U} \to \infty\text{-}\mathsf{Grp}$.*

Proof. Let $A : \mathcal{U}$ be a type. A short calculation shows that the monad extension $(\mathsf{Id}\,, \mathsf{Id}{\downarrow}\,A)$ is algebraic. The result therefore follows from Corollary 1. □

B. Uniqueness of the Groupoid Structure

We now turn to the task of showing the map $\Gamma : \mathcal{U} \to \infty\text{-}\mathsf{Grp}$ is an equivalence. Observe that there is a forgetful map $\Upsilon : \infty\text{-}\mathsf{Grp} \to \mathcal{U}$ which is given by extracting the type of objects (Equation 2) from the opetopic type underlying a groupoid $G : \infty\text{-}\mathsf{Grp}$. It is readily checked that the composite $\Upsilon \circ \Gamma$ is definitionally the identity, and so what remains to be shown is that any $G : \infty\text{-}\mathsf{Grp}$ is equivalent to Γ applied to its type of objects.

Unwinding the definitions, we find that we are faced with the following problem: suppose we are given a monad extension $(M\,, M{\downarrow})$ as well as a opetopic type X : $\mathsf{OpetopicType}\,M$. Under what hypotheses can we prove that $X \simeq_o \downarrow\mathsf{OpType}\,M\,M{\downarrow}$ (where $\simeq_o$ denotes an appropriate notion of equivalence of opetopic types)? A first remark is that the opetopic type $\downarrow\mathsf{OpType}\,M\,M{\downarrow}$ is completely determined by the algebraic structure of the dependent monad $M{\downarrow}$. Therefore, at a minimum, we must assume that the data of the opetopic type X is equivalent to the data provided by $M{\downarrow}$ wherever they "overlap".

To see what this means concretely, let us begin at the base of the sequence, writing $Z = \downarrow\mathsf{OpType}\,M\,M{\downarrow}$ to reduce clutter. Now, the family $Z_0 : \mathsf{Id}\mathsf{x}\,M \to \mathcal{U}$ is, by definition, given by the family of dependent indices $\mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}$ of the dependent monad $M{\downarrow}$. On the other hand, without additional hypotheses, the opetopic type X only provides us with some abstract type family $X_0 : \mathsf{Id}\mathsf{x}\,M \to \mathcal{U}$. Clearly, then, we will need to assume an equivalence $e_0 : (i : \mathsf{Id}\mathsf{x}\,M) \to \mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}\,i \simeq X_0\,i$ in order to have any chance to end up with the desired equivalence of opetopic types.

Moving on to the next stage, here we find that Z_1 is given by the dependent indices

$$\mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}_1 : \mathsf{Id}\mathsf{x}\,M_1 \to \mathcal{U}$$

of the first iteration of the dependent slice-pullback construction. Unfolding the definition, these are of the form

$$(\mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}_1)\,(i\,, j\,, c\,, v) = \sum_{(j : \mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}\,i)} \sum_{(r : j \equiv j')} \sum_{(d : \mathsf{Cns}{\downarrow}\,M{\downarrow}\,c)} (\mathsf{Typ}{\downarrow}\,M{\downarrow}\,d \equiv \nu)$$

With the 4-tuple $(i\,, j\,, c\,, v)$ as in Equation 4. We notice that much of the data here is redundant: by eliminating the equality r and the equality relating ν to the typing function of d, we find that the dependent indices are essentially just dependent constructors of $M{\downarrow}$, slightly reindexed. In other words, a dependent equivalence

$$e_1 : (i : \mathsf{Id}\mathsf{x}\,M_1) \to \mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}_1 \simeq_{e_0} X_1$$

over the previous equivalence e_0 amounts to saying that the relations of the family X_1 "are" just the dependent constructors of $M{\downarrow}$ (again, reindexed according to the typing of their input and output positions). As this is again part of the data already provided by the dependent monad $M{\downarrow}$, we will additionally need to add such an equivalence to our list of hypotheses.

To recap: assuming the equivalences e_0 and e_1 amounts to requiring that the first two stages of the opetopic type X are equivalent to the indices and constructors of the dependent monad $M{\downarrow}$, respectively. What structure remains? Well, the dependent constructors of $M{\downarrow}$ are equipped with the unit and multiplication operators $\eta{\downarrow}$ and $\mu{\downarrow}$. But now, recall from Section III-A that if the family of relations X_1 extends further in the sequence to a family X_2 and we have a proof m_2 : $\mathsf{is\text{-}mult}\,X_2$, then the family X_1 can be equipped with a multiplicative structure given by the functions $\eta\text{-}\mathsf{alg}_{m_2}$ and $\mu\text{-}\mathsf{alg}_{m_2}$ defined there. This is the case in the current situation, if we assume that the opetopic type X is fibrant (in fact, we only need assume that $\mathcal{R}\,X$ is fibrant to make this statement hold). Therefore, the last piece of information in order that X "completely agrees" with the dependent monad $M{\downarrow}$ is that the equivalence e_1 is additionally a *homomorphism*, sending $\eta{\downarrow}$ to $\eta\text{-}\mathsf{alg}_{m_2}$ and $\mu{\downarrow}$ to $\mu\text{-}\mathsf{alg}_{m_2}$. Our theorem now is that this data suffices to prove an equivalence of opetopic types:

Theorem 2. *Suppose $(M, M{\downarrow})$ is a monad extension and X : $\mathsf{OpetopicType}\,M$ an opetopic type such that $\mathcal{R}\,X$ is fibrant. Moreover, suppose we are given the data of*

- *An equivalence $e_0 : (i : \mathsf{Id}\mathsf{x}\,M) \to \mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}\,i \simeq X_0\,i$*
- *An equivalence $e_1 : (i : \mathsf{Id}\mathsf{x}\,M_1) \to \mathsf{Id}\mathsf{x}{\downarrow}\,M{\downarrow}_1 \simeq_{e_0} X_1$ over e_0*
- *Proofs that $s : \eta{\downarrow}M{\downarrow} \equiv_{e_0,e_1} \eta\text{-}\mathsf{alg}_{m_2}$ and $t : \mu{\downarrow}M{\downarrow} \equiv_{e_0,e_1} \mu\text{-}\mathsf{alg}_{m_2}$*

Then there is an equivalence of opetopic types

$$X \simeq_o \downarrow\mathsf{OpType}\,M\,M{\downarrow}$$

We have taken some liberties in the presentation of this theorem (strictly speaking, we have not stated precisely in what sense the second equivalence e_1 is "over" the equivalence e_0, nor precisely what equality is implied by symbol the $\equiv_{e_0,e_1}$) but these omissions can be made perfectly rigorous by standard techniques, and we feel the statement above conveys the essential ideas perhaps more clearly than a fully elaborated statement, which would require a great deal more preparation, not to mention space. See the appendix of the extended version of this article for a proof [19, Theorem 2].

We at last obtain our desired equivalence:

Theorem 3. *The map*

$$\Gamma : \mathcal{U} \to \infty\text{-}\mathsf{Grp}$$

is an equivalence.

Proof. Given $G : \infty\text{-}\mathsf{Grp}$, we let $A : \mathcal{U}$ be its type of objects. We now apply Theorem 2 with $M = \mathsf{Id}$ and $M{\downarrow} = \mathsf{Id}{\downarrow}A$. We may take e_0 to be the identity. The equivalence e_1 is a consequence of [13, Theorem 5.8.2] and the required equalities are a straightforward calculation. □

V. Conclusion

We have presented an approach to defining higher coherent structures in homotopy type theory by equipping type theory with a primitive set of structures collected into a universe $\mathbb{M}$ of polynomial monads, and demonstrated that this approach can be used to prove non-trivial theorems about these structures. In this brief final section, we compare some related approaches and survey some of the possible directions and applications.

A. Future Directions

1) Symmetric Structures: A natural class of structures which escapes the capabilities of our current approach is that of *symmetric structures*, that is, those which incorporate higher analogs of commutativity. Examples would include $\mathbb{E}_\infty$ groups and monoids, symmetric monoidal categories, and general ∞-operads and their algebras.

2) Higher Category Theory: As we have seen, one higher structure which *is* amenable to treatment by our methods is that of an ∞-category. An obvious point to follow up on, then, is how much of the well developed theory of ∞-categories can be formalized in this manner.

3) A General Theory of Structures: As we have mentioned in the introduction, we see the present work as a first step towards a general theory of types and structures. And though we feel certain that at least some of the ideas of the present work will carry over to such a theory, a complete picture of the basic principles remains to be understood. Moreover, a careful investigation of the interaction of our techniques with univalent implementations of type theory (such as *cubical* type theory) also remains for future work.

Accompanying such a general theory, we anticipate a deeper investigation of the meta-theoretic properties of our proposed approach. For example, the Agda implementation is limited by the expressivity of rewrite rules, and complicated by the explicit universe construction, while a proper extension of MLTT would allow for the investigation of meta-theoretic properties like decidability of type checking and strong normalization using techniques like normalization-by-evaluation (and potentially settling the conjecture of II-B). Furthermore, we have not touched at all on the potential models of our system, a topic which we feel deserves careful attention.

References

[1] N. Gambino and J. Kock, "Polynomial functors and polynomial monads," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 154, no. 1, p. 153–192, 2013.

[2] J. C. Baez and J. Dolan, "Higher-dimensional algebra iii. n-categories and the algebra of opetopes," *Advances in Mathematics*, vol. 135, no. 2, pp. 145–206, 1998.

[3] J. Kock, A. Joyal, M. Batanin, and J.-F. Mascari, "Polynomial functors and opetopes," *Advances in Mathematics*, vol. 224, no. 6, pp. 2690–2737, 2010.

[4] D. Gepner, R. Haugseng, and J. Kock, "∞-operads as analytic monads," *arXiv preprint arXiv:1712.06469*, 2017.

[5] J. Cockx, N. Tabareau, and T. Winterhalter, "The Taming of the Rew: A Type Theory with Computational Assumptions," *Proceedings of the ACM on Programming Languages*, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02901011

[6] B. Van Den Berg and R. Garner, "Types are weak ω-groupoids," *Proceedings of the london mathematical society*, vol. 102, no. 2, pp. 370–394, 2011.

[7] P. L. Lumsdaine, "Weak ω-categories from intensional type theory," in *International Conference on Typed Lambda Calculi and Applications*. Springer, 2009, pp. 172–187.

[8] E. Riehl and M. Shulman, "A type theory for synthetic ∞-categories," *arXiv preprint arXiv:1705.07442*, 2017.

[9] P. R. North, "Towards a directed homotopy type theory," *Electronic Notes in Theoretical Computer Science*, vol. 347, pp. 223–239, 2019.

[10] D. Annenkov, P. Capriotti, and N. Kraus, "Two-level type theory and applications," *CoRR*, vol. abs/1705.03307, 2017. [Online]. Available: http://arxiv.org/abs/1705.03307

[11] Agda development team, "Agda 2.6.1.1 documentation," 2020. [Online]. Available: https://agda.readthedocs.io/en/v2.6.1.1/

[12] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1975, vol. 80, pp. 73–118.

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

[14] T. Altenkirch, N. Ghani, P. G. Hancock, C. McBride, and P. Morris, "Indexed containers," *J. Funct. Program.*, vol. 25, 2015. [Online]. Available: https://doi.org/10.1017/S095679681500009X

[15] G. Huet, "Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems," *J. ACM*, vol. 27, no. 4, p. 797–821, Oct. 1980. [Online]. Available: https://doi.org/10.1145/322217.322230

[16] F. Blanqui, G. Genestier, and O. Hermant, "Dependency pairs termination in dependent type theory modulo rewriting," *CoRR*, vol. abs/1906.11649, 2019. [Online]. Available: http://arxiv.org/abs/1906.11649

[17] B. Ahrens, K. Kapulkin, and M. Shulman, "Univalent categories and the rezk completion," *Mathematical Structures in Computer Science*, vol. 25, no. 5, pp. 1010–1039, 2015.

[18] T. Leinster, *Higher operads, higher categories*. Cambridge University Press, 2004, no. 298.

[19] A. Allioux, E. Finster, and M. Sozeau, "Types are internal ∞-groupoids (extended version)," *arXiv preprint*, 2021.