Reedy Types and Dependent Type Families

Alexander Kuklev[1,2] ‹a@kuklev.com›
[1]Radboud University Nijmegen, Software Science
[2]JetBrains Research

Building on the unpublished ideas of C. McBride and ideas from "Displayed Type Theory and Semi-Simplicial Types"[1] by A. Kolomatskaia and M. Shulman, we propose a novel extension for univalent Martin-Löf Type Theories (MLTTs) for internalizing Reedy categories.

Indexing and fibering over Reedy types provide effective machinery to deal with syntaxes that include binding and become indispensable when internalizing the syntax and semantics of type theories themselves. In this way, we obtain convenient tooling and uniformly establish the existence of initial models for structures like weak ω-categories,[2] virtual equipments,[3] (∞,1)-toposes once the Higher Observational Type Theory (HOTT)[4] is complete.

Finally, this approach should lead to a homoiconic[5] univalent type theory, i.e. one capable of representing its syntax as an inductive family and thus performing structural induction over it.

1 Why do we need dependent type families?

Finitary type families abstractly embody formalized languages. For example, consider the following simple language of arithmetic and logical expressions:[6]

```
data ExpressionKind
  Numeric
  Logical

data Expr : ExpressionKind → Type
  Literal(n : Int)          : Expr Numeric
  Sum(a b : Expr Numeric)   : Expr Numeric
  Mul(a b : Expr Numeric)   : Expr Numeric
  Div(a b : Expr Numeric)   : Expr Numeric
  Pow(a b : Expr Numeric)   : Expr Numeric
  Neg(a : Expr Numeric)     : Expr Numeric
  Log(a : Expr Numeric)     : Expr Numeric

  Eq(a b : Expr Numeric)    : Expr Logical
  Lt(a b : Expr Numeric)    : Expr Logical
  Or(a b : Expr Logical)    : Expr Logical
  Not(a : Expr Logical)     : Expr Logical
```

Dependent type families allow scaling up this approach to languages with scoped binders (variables, type definitions) including general-purpose programming languages themselves.

Data types defined that way are inhabited by abstract syntax trees corresponding to finite expressions of the language, and they come with a recursive descent analysis operator enabling type-driven design of correct-by-construction analysers and interpreters facilitating robust type checking, compilation, static analysis, and abstract interpretation in general.

As for IDEs, inductive type families enable designing biparsers for those languages, parsers that maintain a one-to-one mapping between the source code and the respective annotated abstract syntax tree, enabling both fast incremental reparsing and mechanized refactoring.

[1] https://arxiv.org/abs/2311.18781
[2] https://arxiv.org/abs/1706.02866
[3] https://arxiv.org/abs/2210.08663
[4] https://ncatlab.org/nlab/show/higher+observational+type+theory
[5] https://homotopytypetheory.org/2014/03/03/hott-should-eat-itself/
[6] This paper is written in literate dependent Kotlin, see https://akuklev.github.io/kotlin/kotlin_literate.pdf. We use an Agda-like syntax for inductive definitions, except using angle brackets for type parameters and irrelevant function parameters, allowing to concisely introduce records as inductive types with a unique generator.

To represent languages with typed variables, one introduces the type `Ty` representing variable types of the language, and the type family `Tm (ctx : *)` of terms in a given context given by a list of types. Definition of term substitution can be vastly simplified if one recasts the type of contexts as a size-indexed type family `Ctx (size : LΔ)`, which requires the notion of lax types (such as `LΔ`) to enable context extension. In case of dependently typed languages, `Ty` is not a type, but a type family `Ty (c : Ctx)`, where the contexts are iterated dependent pairs

```
data Ctx
  Empty : Ctx
  Append(prefix : Ctx, head : Ty prefix)
```

To define substitution, we have to recast `Ctx` as a simplicial type family `Ctx (shape : Δ)`, which requires the notion of Reedy types (such as `Δ`) that enable selection of subcontexts. Bidirectionally typed languages split terms into a type family of normal forms satisfying a given type `Nf (c : Ctx, t : Ty c)` and a fibered family of reducible expressions that synthesize their types and normal forms `Rx : (c : Ctx)ᵈ ↓ (ty : Ty c, Nf (c, ty))`.

2 Setting and basics

Our base theory will be the Higher Observational Type Theory with a hierarchy of ΠΣ-closed cumulative universes `Type : Type⁺ : Type⁺⁺ : ⋯` featuring □-modality-based polymorphism,[7] adjoined by quotient inductive-inductive-recursive type definitions.

The simplest types of this kind are the finite datatypes (also known as enums) defined by enumerating their possible values:[8]

```
data Bool
  False   `ff`
  True    `tt`

data Unit
  Point   `()`

data Void {}    // no elements at all
```

We can generalize them to sum types by allowing indexed families of possible values:[9]

```
data Possibly<X>
  Nothing
  Value(x : X)
```

Each inductive type comes along with a dual typeclass:[10]

```
data Boolᴿ<this Y>(ifTrue  : Y,
                   ifFalse : Y)

data Possiblyᴿ<X, this Y>(ifNothing : Y,
                          ifValue(x : X) : Y)
```

Instances of these typeclasses represent by-case analysis of the respective sum types.

Inhabitants of inductive types `x : T` can be converted into functions[11] (known as Church representations) that evaluate their by-case analysers: `xᶜ : ∀<Y : Tᴿ> Y`:

[7] https://akuklev.github.io/polymorphism

[8] Fancy aliases for plain identifiers can be introduced in backticks. See `kotlin_academic.pdf` for details.

[9] We omit the type of `X` in `Possibly<X>`, because parameter types can be omitted if inferrable.

[10] Typeclasses are introduced as records with a marked (by `this`), possibly higher-kinded, typal parameter, but turn into a subtype of their marked parameter's type, e.g. `Boolᴿ <: Type`, so every `T : Boolᴿ` is both a type and an instance of `Boolᴿ<T>`, which does not introduce ambiguities since types and families cannot have fields, while typeclass instances are records and consist from their fields. See `kotlin_typeclasses.pdf` for details.

[11] Result type in definitions can be omitted in assignment-style definitions as here.

```
def False<Y : Bool^R>^c = Y.ifFalse
def  True<Y : Bool^R>^c = Y.ifTrue

def Nothing<X, Y : Possibly^R<X>>^c        = Y.ifNothing
def   Value<X, Y : Possibly^R<X>>(x : X)^c = Y.ifValue(x)
```

What if we want to return values of different types for `True` and `False`? We can first define a function from booleans into types `Y : Bool → Type` and then a dependent case analyser

```
data Bool^M<this Y : Bool → Type>(ifTrue  : Y True,
                                  ifFalse : Y False)
```

To apply dependent case analysers to inhabitants of the respective type, we need a special operator called induction for reasons explained below:[12]

```
I-ind<Y : I^M> : ∀(x : I) Y(x)
```

Non-finite inductive types admit (strictly positive) recursion in type definitions, enabling such types as natural numbers, lists, and trees:

```
data Nat `ℕ`
  Zero `0`
  PosInt(pred : ℕ) `pred⁺`

data Int `ℤ` :> Nat
  NegInt(opposite : PosInt)   // So, Int is either Nat or NegInt

data List<T> `T*`
  EmptyList : T*
  NonEmptyList(head : T, tail : T*) : T*

data BinTree<T>
  Leaf
  Node(label: T, left : BinTree<T>, right : BinTree<T>)

data VarTree<T>
  Leaf
  Node(label: T, branches : VarTree<T>*)

data InfTree<T>
  Leaf
  Node(label: T, branches : Nat → InfTree<T>)
```

All above examples except infinitely branching trees are finitary inductive types, i.e. inductive types with the property that all of their generators have a finite number of parameters, and all these parameters are of finitary inductive types. Finitary inductive types may be infinite, but their inhabitants can be encoded by natural numbers or equivalently finite bit strings.

Finitary inductive types embody single-sorted languages. They are inhabited by abstract syntax trees corresponding to finite expressions of the language formed by their generators.

"Case analysis" for the type of natural numbers provides n-ary iteration operator:

```
data Nat^R<this Y>(base : Y,
                   next(p : Y) : Y)
```

Analysing a natural number `n` by `R : Nat^R<Y>` yields `n^c<R>() = (R.next)^n R.zero`, allowing to iterate arbitrary functions given number of times. In general, "case analysis" turns into "recursive descent analysis". For lists and trees we obtain the respective fold operators.

Type-valued functions on natural numbers `Y : Nat → U` can encode arbitrary predicates, and a dependent `Nat`-analyser `Nat^M<Y>` encodes an induction motive: it establishes a proof of the

[12] In Displayed Type Theory `I-ind<Y : I^M>(x : I)` can be defined as `x^cd<Y>`, and `I^M` as `I^Rd I^c`.

base case `Y(zero)` and the inductive step `Y(n) → Y(n⁺)`. Dependent case analysis operator turns induction motives into to proof the predicate for all natural numbers, that is why it is also known as induction operator. The presence of induction witnesses that inductive types contain only inhabitants that can be obtained by finite compositions of their generators. Which is also the reason why data types described in terms of their generators are called inductive types.

While ordinary inductive types are freely generated, quotient inductive types additionally contain generators of identities between their inhabitants, so we can define rational numbers:

```
data Rational(num : Int, den : PosInt) `ℚ`
  expand<num, den>(n : PosInt) : Rational(num, den) = Rational(num · n, den · n)
```

Here, in addition to listing generators, we require that some actions on generators (expanding the fraction or permuting list elements) must be irrelevant for all predicates and functions defined on these types.

An inductive definition may simultaneously define a family of types dependent on one another. This is not limited to finite families: we can allow type families indexed by an arbitrary type `J`:

```
data SizedList<T> : Nat → Type           // Also known as Vec T n in Agda
  EmptySizedList : SizedList<T> 0
  NonEmptySizedList<n>(head : T, tail : SizedList<T> n) : SizedList<T> n⁺
```

This way we can also introduce finite types of a given size (used as an implicit conversion):

```
data operator asType : Nat → Type        // Also known as Fin n in Agda
  Fst<size> : asType size⁺
  Nxt<size>(prev : size) : asType size⁺
```

Now we can use numbers as types which come in handy for advanced collections:

```
data HList<T : Nat → Type><n : Nat>(items : n → T n)    // Heterogeneous lists

data Collection<T><size : Nat>(items : size → T)        // Finite multisets
  permute<size, items>(p : size!) : Collection(items) = Collection(items ∘ p)

data FinSet<T><size : Nat>(items : size → T)            // Finite sets
  multipermute<n, m, items>(inj : n ↣ m) : FinSet(items) = FinSet(items ∘ inj)
```

where `T!` is the type of automorphisms (permutations) of the type `T`, `X ↣ Y` the type of injections.

3 Lax types: injective Reedy categories

Consider the quotient inductive type of eventually-zero sequences:

```
data EvZeroSeq
  Zeros : EvZeroSeq
  Prepend(head : Nat, tail : EvZeroSeq)

  expand : Prepend(0, Zeros) = Zeros
```

As we have seen above, we can turn the type of lists to a size-indexed type family over `Nat`, but we cannot make `EvZeroSeq` into a type family over `Nat` because `extend` generates equality between "lists" of different sizes. We need a "lax" index type instead of `Nat`:

```
data LaxNat `LΔ` : ℓType
  LaxNat(n : Nat)                    // Ordinary constructor LaxNat(n)
  LaxNat(n) [m : Nat⟩ LaxNat(n + m)  // Higher directed constructor [m⟩
  [n⟩ [m⟩ ↦ [(n +) m⟩                // This reduction is the action of [n⟩ on [m⟩
```

To each universe `U` we'll have an associated lax universe `ℓU` occupied by the types like the one above. Lax inductive types are stratified directed counterparts of quotient inductive types.

Ordinary types `T : U` admit types `(x ≃ y) : U` of identifications between their elements `x y : T`, written `(x = y) : Prop` for strict data types. Similarly, lax types `S : ℓU` admit extender types: for every element `s : S`, there is a type family `s↑ : Pᵈ`. We will write `s ↑ t` for `s↑ t`.

Quotient inductive types admit generators of identities `x = y` between their elements. Lax types allow generators of extenders like `s [n⟩ t` that generate inhabitants of the type `s ↑ t`. Sources of extenders must be structurally smaller than (or equal to) their targets to enable typechecking. Whenever we define an extender `s [n⟩ t` , we must also define how it acts on all possible extenders `e : t ↑ t'` yielding some `[f n⟩ : s ↑ t'`. This action must be given by some function `f` to ensure associativity by construction (because function composition is), and the action of level extenders `e : s ↑ s` must be given by the `id` function. Putting everything together, lax types form *strictly associative inverse (= injective Reedy) categories*.

Every function we define on a lax type must have an action on all generators, including extender generators, mapping them either to identities or extenders between results (functoriality).
To have an example, let us define addition for `LaxNat`s:

```
def add : LaxNat² → LaxNat
  (LaxNat(n), LaxNat(m)) ↦ LaxNat(m + n)
  (n[k⟩, m) ↦ add(n, m) [k⟩
  (n, m[k⟩) ↦ add(n, m) [k⟩
```

Let us denote universes of `J`-indexed type families by `Jᵈ` instead of `J → Type`. It does not make any difference ordinary types `J : U` , but for lax types it provides additional flexibility required to introduce `SizedEvZeroSeq` as desired.

```
data SizedEvZeroSeq : LaxNatᵈ
  Zeros : SizedEvZeroSeq LaxNat(0)
  Prepend<n>(head : Nat, tail : SizedEvZeroSeq n) : EvZeroSeq (LaxNat(1) + n)

  expand : ???
```

Before we fill in the gap in the above definition, note that inductive families `F : Jᵈ` also act on on extenders `e : s ↑ t` for `s t : J`. Terms of `F(e)` are generators of the type `F(s) ↑ F(t)`, whose elimination rule provides domain substitution for functions defined on `F(s)`:

```
F(e)ᶜ : ∀<Y> (F(t) → Y) → (F(s) → Y)     // We also have a dependent version:

F(e)ᶜᵈ: ∀<Y : F(t)ᵈ> (∀(x : F(t)) Y(x)) → (∀(x : F(s)) F(e) Y)(x))
```

Now we can fill in the gap in the definition of `SizedEvZeroSeq`. The type of the equality generator `Prepend(0, Zeros) = Zero` does not typecheck because its left- and right-hand sides have different types. Let us rewrite its type by abstracting an anonymous function and immediately applying it:[13] `{ Prepend(0, Zeros) = it } f`. Now we can apply the extender `(SizedEvZeroSeq LaxNat(0)[1⟩)ᶜ` to the function. This way we change the domain of the function from `SizedEvZeroSeq LaxNat(1)` to `SizedEvZeroSeq LaxNat(0)` and apply it to `Zeros`.

```
data SizedEvZeroSeq : LaxNatᵈ
  Zeros : SizedEvZeroSeq LaxNat(0)
  Prepend<n>(head : Nat, tail : SizedEvZeroSeq n) : SizedEvZeroSeq (LaxNat(1) + n)

  expand : (SizedEvZeroSeq LaxNat(0)[1⟩)ᶜ { Prepend(0, Zeros) = it } Zeros
```

[13] Anonymous functions are written like `{ n : Int ↦ n + 1 }` or `{ it + 1 }`. Types can be omitted if inferrable.

4 Lax algebraic theories

Models of single-sorted algebraic theories arise as dual typeclasses for quotient inductive types we will call prototypes of those theories. Monoids arise as models for the following type:

```
data Monoid^P
  e : Monoid^P
  (∘) : Monoid^P → Monoid^P → Monoid^P

  unitorL : x = e ∘ x
  unitorR : x = x ∘ e
  associator : (x ∘ y) ∘ z = x ∘ (y ∘ z)
```

The dual typeclass `Monoid^PR<T>` will be automatically called `Monoid<T>`. The canonical examples of monoids are lists under concatenation (free monoids) and endomorphisms under composition:

```
object List<T> : Monoid(EmptyList, (++))          // Implicitly resolvable instances
object Endo<T> `T↻`: Monoid<T → T>(id, (∘))       // of typeclasses are introduced as
object Auto<T> `T!`: Group<T ⇔ T>(id, (∘), ( ⁻)) // companion objects of typeformers
```

We can also provide an unbiased definition for monoids, where the composition operation is not taken to be binary, but can have any finite arity including zero for the neutral element `e`. Let us introduce several types:

```
data PTree<T>
  Leaf(label : T)
  Node(branches : PTree<T>*)

data SizedPTree<T> : ℕ^d
  Leaf(label : T) : SizedPTree<T> 1
  Node<sizes : ℕ*>(branches : HList<T> sizes) : SizedPTree<T> (sum sizes)
```

A `pr : Parenthesization(n : ℕ)` is just a `SizedPTree<Unit> n` that acts on lists `xs : T*` turning them into respective trees `pr(xs) : PTree<T>`.

Now we can proceed to the definition of an unbiased monoid:

```
data Monoid^P
  compose : Monoid^P* → Monoid^P

  expand(xs : Monoid^P*, pr : Parenthesization xs.size)
  : compose(xs) = (pr(xs) ▸map compose)
```

If we can orient equalities so they map structurally smaller terms to structurally larger ones, we can reformulate the theory as a lax type with extenders instead of identities. Algebraic theories with extenders are known as lax algebraic theories.

```
data LaxMonoid^P : ℓType
  compose : LaxMonoid^P* → LaxMonoid^P

  compose(xs) [pr : Parenthesization xs.size⟩ (pr(xs) ▸map compose)

  [pr⟩ [pr'⟩ ↦ [expand (pr' ∘) p⟩
```

When mapping into ordinary types, extenders can only be mapped into identities, so exchanging identities for extenders does not affect set-like models, but enforces coherence in non-truncated models as their functoriality must hold by construction. When mapping into lax types, lax theories have additional lax models as we will see below in the lax monoidal category example.

Lax formulation has an advantage even if we're only interested in set-like models as it provides an explicitly confluent system of rules making the theory stratified. Stratifiability of the sort algebra is necessary for generalized algebraic theories to have syntactic free models and an effective model structure on the category of their models.

5 Fibered types

Many operations on containers have the following property: the shape of the resulting container only depends on the shapes of the arguments. For example, size of the list computed by `concatenate`, `map`, and `reverse` can be computed based on the sizes of the input lists.

To account for that, let us introduce a notion of fibered types and functions between them, namely the functions with the property described above.

A fibered type is given by a pair of a type `E` and a function `f : E → B` written as `E / f`. We will denote the type of such terms as `E ↓ B` and occasionally `(e : E) ↓ B(e)` in case of dependent functions.

Fibered types above some base type `B : U` form a type family `↓B` and `E ↓ B := ↓B E` is just a reverse application:

```
data ↓B : Uᵈ
  (E : U) / (f : E → B) : E ↓ B
```

For example, we can take the type of lists `T*` and the function `size`: `T* / size : T* ↓ ℕ`.

A function between fibered types is a pair of functions `(f / b) : (X / p) → (Y / q)`, so that the following square commutes by construction:

```
X --[f]--> Y
|p         |q
V          V
A --[b]--> B
```

Consider a few examples of functions on fibered types:

```
def reverse<T> / id  : (T* / size)  → (T* / size)
def concat<T>  / add : (T* / size)² → (T* / size)
def flatten<T> / sum : (T* / size)* → (T* / size)

def map<X, Y>(f : X → Y) / id : (X* / size)  → (Y* / size)
```

Inductive-recursive definitions are mutually dependent definitions of an inductive type and a recursive function on that type. Such definitions naturally generate a fibered type.

```
data V : ↓Type
  MyUnit / Unit
  MyBool / Bool
  MyPi(X : V, Y : X → V) / ∀(x : X) Y(x)
```

We will use `|_|` as the default name for the fibering function unless it is explicitly named. A similar notion of fibered types in that sense was first introduced in "Fibred Data Types"[14] by N. Ghani, L. Malatesta; F. Nordvall Forsberg, and A. Setzer.

Type families `T : Xᵈ` can be fibered over type families `Y : Xᵈ`. For this case, we will introduce the notation `(x : X)ᵈ ↓ Y(x)`. Unless `X : U` is a shape, it is equivalent to `∀(x : X) (U ↓ Y(x))`.

Fibered types allow introducing dependent extender types: for a type `X : U` and a fibered type `Y : Y' / X`, extenders `X ↑ Y` are terms `e : ∀<Z : Xᵈ> (∀(x : X) Z(x)) → (∀(y : Y') Z(|y|))` so that `{ |e(f(it))| } = f` by construction.

Σ-type former is tightly connected to fibered types. For every type family `Y : Bᵈ`, we have the fibered type `Σ'Y / fst : ΣY ↓ B`. On the other hand, `Σ<J : U> : Jᵈ → U` maps type families into types, so for every J we have a fibered type `Jᵈ / Σ<J>`.

[14] https://doi.org/10.1109/LICS.2013.30

6 Matryoshka types: projective Reedy categories

So far we only applied the operator ($^{\mathtt{d}}$) to types `T : U`, but this operator has been introduced in Displayed Type Theory for all terms, including type families `F : B`$^{\mathtt{d}}$ for some `B : U`

```
F^d : B^d → U
F^d(E : B^d) = ∀<i> (F i) → E i
```

Let us now extend the definition of ($^{\mathtt{d}}$) to fibered types:

```
(X / |·|)^d : ∀(x : X) (|x|^d Y)^d
```

Now let us introduce matryoshka types fibered over type families indexed by themselves:

```
data SΔ1 : ↓SΔ1^d
  Fst / Void
  Snd / data
    Dep : |Snd| Fst
```

Here we define a type with two generators `Fst` and `Snd`, and for each one a type family `|x| : SΔ1`$^{\mathtt{d}}$. In this case, `|Fst|` is empty and `|Snd|` contains a unique element `Dep : |Snd| Fst`.

Let us now consider a type family `Y : SΔ1 / |·|)^d`. Let us first apply it to `Fst`:

```
Y(Fst) : (|Fst|^d Y)^d
Y(Fst) : (|Void|^d Y)^d
Y(Fst) : (Unit)^d
Y(Fst) : Type
```

So, `Y(Fst)` is just any type. Now let us apply it to `Snd`:

```
Y(Fst) : (|Snd|^d Y)^d
```

`|Snd|` is itself a type family fibered over `SΔ1`, so |Snd|$^{\mathtt{d}}$ expects an argument of the same type as `|Snd|` and morally reduces to the "dependent function type" `∀<xs> (|x| xs) → Y xs` (not a valid expression as `xs` is not a single argument, but a telescope).

Fortunately, `|Snd|` is nonempty for only one argument, namely `Fst`, so we have

```
Y(Snd) : (Y(Fst))^d
```

Thus, our type family is merely a dependent pair `Σ(T : Type) (T → Type)`. We can now define dependent types as type families. Let us try a more complex example:

```
data SΔ2 : ↓SΔ2^d
  El1 / Void
  El2 / data
    Dep : |El2| El1
  El2 / data
    Dep : |El3| El2 ??
```

We run into a problem: `|El3|` is a type family over a fibered type, so `|El3| El2` expects yet another argument, and it should be of the type `|El3| El1`. We have no other way but to create a suitable element:

```
data SΔ2 : ↓SΔ2^d
  El1 / Void
  El2 / data
    Dep : |El2| El1
  El2 / data
    Dep1 : |El3| El1
    Dep2 : |El3| El2 Dep1
```

Typechecking requires, indexes of the types `|x|` to be be structurally smaller than `x`. As we now see, such types form *strictly associative direct (= projective Reedy) categories*.

Vocabularies `V` of theories with dependent sorts can be expressed as finite matryoshka types, theories being typeclasses of families `Carrier : V^d`. Algebraic theories with dependent sorts are typeclasses dual to type families `Prototype : V^d`. Categories themselves have the vocabulary

```
data Cell2₊ : ↓Cell2₊ᵈ
  Ob  / Void
  Mor / data
    Source : |Mor| Ob
    Target : |Mor| Ob
```

The canonical infinite example is the type of abstract semi-simplices

```
data SΔ : ↓SΔᵈ
  Zero / Void
  Next(s : SΔ) / data
    Prev(p : |s|) : |Next(s)| p
    Last : |Next(s)| s Prev(s)
```

Type families over SΔ are known as semi-simplicial type families, infinite type telescopes

```
(T₁ : Type,
 T₂(x₁ : T₁) : Type,
 T₃(x₁ : T₁, x₂ : T₂ x₁) : Type,
 T₄(x₁ : T₁, x₂ : T₂ x₁, x₃ : T₃ (x₁, x₂)) : Type,
 ...)
```

As we have done with natural numbers, we can define an implicit conversion from semi-simplices to types, yielding their truncated versions `Unit`, `SΔ1`, `SΔ2`, etc. This way we can define a dependent version of heterogeneous lists and sequences:

```
data DList<T : SΔᵈ><n : SΔ>(items : n → T n)
data DSeq<T : SΔᵈ>(head : T Zero                                  // Actually,
                   tail : DSeq<T Next(Zero) { Last ↦ head }>)    // a codata type
```

Dependent sequences are also known as very-dependent functions[15] on `Nat`. The construction of `SΔ` can be replicated for any inductive type `J` fibering all generators over their recursive arguments. Such very-dependent function types can be used to model communication protocols.

7 Reedy types

Reedy categories are allowed to have both injective and projective arrows, and can be represented by lax matryoshka inductive types, which we will from now on call Reedy types `T : яU`. In particular, we can add extenders to `SΔ` to ensure that functions on `Tₙ` can be also applied to `Tₙ₊₁`. Let us start with an incomplete definition:

```
data Δ : яType
  Zero / Void
  Next(s : Δ) / data
    Prev(p : |s|) : |Next(s)| p
    Last : |Next(s)| s Prev(s)

  Zero[n : Nat⟩ (n⁺ᶜ Next)(Zero)
  Next(s)[n : Nat, f : Fin(n⁺ + (s as ℕ)) → Fin(s as ℕ)⟩ (n⁺ᶜNext)(s)
  [n⟩ [n', f'⟩ ↦ [n', f'⟩
  [n, f⟩ [n', f'⟩ ↦ [n', { it ∘ f } f'⟩
```

[15] Jason J. Hickey. Formal objects in type theory using very dependent types (1996)

Extenders define type families on a fibered type, so they have to specify action on selectors. In this way, we will specify intertwining identities between selectors and extenders (i.e. face and degeneracy maps as they are known for geometric shapes). Here is a complete definition:

```
data Δ  : яType
  Zero / Void
  ...TODO: Not just write, but make the stuff typecheck!
```

```
data operator asType : Δ^d
  ...TODO: Not just write, but make the stuff typecheck!
```

Type families on `Δ` are the infamous simplicial type families that allow mutual definition of types and contexts when describing dependently typed theories:

```
data Ty : (ΣΔ Ctx)^d
  ...language-specific
```

```
def Ctx : Δ → Type
  Zero ↦ Unit
  Next(s) ↦ Ty s this(s)
```

Then we can define terms, telescopes, and substitution:

```
data Tm : (ΣΣΔ Ctx Ty)^d
  ...language-specific
```

```
data Tel : (ΣΔ Ctx)^d
  EmptyTel : Tel Zero
  Append<c>(prefix : Tel c, t : Ty c) : Tel (Next(c), {
    Next(c) ↦ t
    else ↦ c
  })
```

```
def applyTy<c>(tm : Ty c, args : Tel c) : Ty Zero ()
def applyTm<c, t>(tm : Tm (c, t), args : Tel c) : Tm (Zero, (), applyTy(t, c))
```

If we postulate the rules of the theory as identites in `Ty`, we might run into unsolvable coherency issues. Fortunatelly, computational type theories can be presented bi-directionally, i.e. instead of `Tm` we will have a free inductive type `Nf : (ΣΣΔ Ctx Ty)^d` of normal forms satisfying a given type, and a fibered family of reducible expressions that synthesize their types and normal forms `Rx : (c : Ctx)^d ↓ (ty : Ty c, Nf (c, ty))`.

The putative Higher Observational Type Theory has additional complexity: it needs higher dimentional substitutions, requiring a cubical type family of judgement sorts.

8 Categories as models of a reedy prototype

Let us revisit the category vocabulary, adding an extra extender:

```
data Cell2 : ЯType
  Ob  / Void
  Mor / data
    Source : |Mor| Ob
    Target : |Mor| Ob

  Ob [よ〉 Mor / ff
```

Just like we defined a monoid prototype above, we can define a prototype for categories as an indexed quotient-inductive type family:

```
data Catᴾ : Cell2ᵈ
  id<o : Catᴾ Ob> : (Catᴾ Mor)(o, o)
  (►)<x, y, z> : (Catᴾ Mor)(x, y) → (Catᴾ Mor)(y, z) → (Catᴾ Mor)(x, z)

  unitorL<x, y> : ∀(f : (Catᴾ Mor)(x, y)) f = id ► f
  unitorR<x, y> : ∀(f : (Catᴾ Mor)(x, y)) f = f ► id
  associator<f, g, h> : (f ► g) ► h = f ► (g ► h)
```

The dual typeclass is precisely the usual definition of a category:

```
data Cat<this Ts : Cell2ᵈ>(
  id<o> : Ts.mor(o, o),
  (►)<x, y, z> : Ts.mor(x, y) → Ts.mor(y, z) → Ts.mor(x, z)

  ... subject to unitality and associativity
)
```

Yoneda extender induces equivalence between isomorphism and equivalence for objects:

```
∀<x, y> (a ≃ b) ≃ Σ(f : Ts.mor(x, y)
                    g : Ts.mor(y, x)) (f ► g = id) and (f ► g = id)
```

But more importantly, it imposes functoriality on functions between categories:

```
f : ∀<Xs Ys : Cat> Xs.Ob  → Ys.Ob
g : ∀<Xs Ys : Cat> Xs.Obⁿ → Ys.Ob    // for any type n
h : ∀<Xs Ys : Cat> Xs.Ob* → Ys.Ob    // for any monadic container
```

Applying these functions to the embeddings `o[よ〉` one obtains their action on morphisms, which must commute with `Cat`-structure, i.e. compositions.

This way we can even introduce monoidal (or lax monoidal) structure on categories as follows:

```
data MonoidalCat<this Ts : Cat>(m : ∀<i> Monoid<Ts i>)
data LaxMonoidalCat<this Ts : Cat>(m : ∀<i> LaxMonoid<Ts i>)
```

In fact, we can lift any typeclass `C<this T>` to `J`-indexed type families by

```
data (C ↗ J)<this T : Jᵈ>(c : ∀<i> C<T i>)
```

Exactly as we did for monoids, we can proceed to derive an unbiased definition a lax prototype. To our understanding, lax categories are precisely the virtual double categories, "the natural place in which to enrich categories". Since we now can describe weak ω-categories algebraically, it is worth studying if categories weakly enriched in ω-categories are ω-categories themselves.

9 Displayed algebraic structures and parametricity

We already have the `Monoid` typeclass, so let us define their category. First, we need a notion of monoid homomorphisms, which can be given by a "function class":

```
data MonoidMorphism<X Y : Monoid>(this apply : X → Y, ...axioms)
```

So far we need a type `Ob` of objects rather than a typeclass, so let us define the category of small monoids (ones with carriers inside the first universe `Type`):

```
object Monoid : Cat<{Ob ↦ Monoid, Mor ↦ MonoidMorphism}>(id, (∘))
```

The □-based approach to polymorphism[16] allows automatically deriving categories `Monoid⁺` of `Type⁺`-sized monoids, `Monoid⁺⁺` and so on, and transferring proofs and constructions upwards this hierarchy. With the display operator `( ᵈ)` we can do even better. It turns a typeclass like `Cat` into a displayed typeclass, a typeclass of typeclasses. In this way, we can introduce a companion object making the typeclass of `Monoids` into a (size-agnostic) displayed category:

```
object Monoid : Catᵈ<{Ob ↦ Monoid, Mor ↦ MonoidMorphism}>(...)
```

Homomorphisms can be defined uniformly for all algebraic theories. A type class `T` is called algebraic if it is a dual for some inductive type `Tᴾ` called its prototype. Given an instance `X : T` of an algebraic typeclass, let us consider the typeclass `Tᵈ<X>`. Its instances consist of a type family indexed by elements of `X` (a multivalued function on `X`) and an instance `Y : T` on its values. In other words, its instances are all possible promorphisms `X ↠ Y` (many-to-many homomorphisms) on `X`. Ordinary homomorphisms are the univalent (= many-to-one) promorphisms:

```
data Hom<X : T>(this pm : Tᵈ X, ∀(x) isContr(pm x))
```

This way we can uniformly derive the category of models for every algebraic theory:

```
object Monoid : Catᵈ
object Group : Catᵈ
object Ring : Catᵈ
object Cat : (Cat ↗ Cell2)ᵈ
```

The last line requires some explanation: The typeclass `Cat` itself is a typeclass of families over `Cell2`. To obtain the typeclass of such type family classes, we must lift the typeclass `Cat` to families over `Cell2` (as described in the previous section) and build a displayed type. This process can be iterated yielding `Cat : (Cat ↗ Cell2)ᵈ : ((Cat ↗ Cell2)ᵈ ↗ Cell2)ᵈ : ···`

Lifting parametric proofs and constructions upwards such hierarchies can be achieved by generalizing the operation `↗` to typeclasses of `S`-indexed families, so unary □-parametricity can be generalized to its `S`-ary form. In this way, we expect to have a satisfying solution to all size issues arising in ordinary and higher category theory.

Exactly as type universes `Type⁺ⁿ`, universes of models for algebraic theories are not merely categories: they come with an inbuilt notion of promorphisms `(X ↠ Y)` and distinguished families of fibrations `X ↓ Y` and extensions `X ↑ Y`. Lax and/or dependently sorted algebraic theories exhibit non-invertible higher morphisms and thus form weak ω-categories. With this amount of coherent structure, our theory should be capable of formalizing the nLab.[17]

[16] https://akuklev.github.io/polymorphism
[17] http://ncatlab.org

10 Future work

So far we have only considered dependent type formers valued in ordinary types, and type families (valued in universes as categories), but it should be possible to introduce broader dependent type formers in directed universes `ЯU` using an approach modelled after "Type Theory for Synthetic ∞-categories"[18] by E. Riehl and M. Shulman.[19]

Besides Reedy types, higher directed universes `ЯType⁺` and upwards are also populated by large types equipped with appropriate structure: ordinary universes, universes of algebraic structures, universes of type families ("presheaf universes"), and conjecturally also sheaves which can be presented as fibered model-valued families.

Since universes of lax algebraic theories exhibit higher morphisms, ultimately we shall be pursuing stacks.

[18] https://arxiv.org/abs/1705.07442
[19] https://rzk-lang.github.io/