Reedy Types and Type Families Thereover

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Building on the unpublished ideas of C. McBride and ideas from "Displayed Type Theory and Semi-Simplicial Types" by A. Kolomatskaia and M. Shulman, we propose a novel extension for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,[1] virtual equipments,[2] (∞,1)-toposes once the Higher Observation Type Theory (HOTT)[3] is complete.

Finally, this approach should lead to a homoiconic[4] univalent type theory, i.e. one capable of representing its syntax as an inductive family and thus performing structural induction over it.

Why do we need advanced type families?

Finitary type families abstractly embody formalized languages. For example, consider the following simple language of arithmetic and logical expressions:[5]

```
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
```

If we allow generalized types as indexes, this approach scales up 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/1706.02866
[2] https://arxiv.org/abs/2210.08663
[3] https://ncatlab.org/nlab/show/higher+observational+type+theory
[4] https://homotopytypetheory.org/2014/03/03/hott-should-eat-itself/
[5] 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 in the language, and the type family `Tm : Ctx`$^{\mathtt{d}}$ of terms in a given context, where contexts are lists of types `Ctx := Ty`*. Definition of term substitution can be vastly simplified if we make the type `Ctx` of contexts fibered over the lax type `LaxNat` that enables context extension and selection of subcontexts.

In case of dependently typed languages, we'll have a type family `Ty : Ctx`$^{\mathtt{d}}$ of variable types available in a given context `c : Ctx`, and the type of contexts is an iterated dependent pair type

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

fibered over the Reedy type Δ that enables context extension and selection of subcontexts and respecting type dependencies, which is vital for the definition of the type family `Ty : Ctx`$^{\mathtt{d}}$ and the type family `Tm : (c : Ctx, Ty c)`$^{\mathtt{d}}$ of terms satisfying a given type in a given context. Bi-directionally typed languages (of computational type theories) require a separate fibered family `Rx : (c : Ctx)`$^{\mathtt{d}}$ `↓ (ty : Ty c, Tm ty)` of reducible expressions that synthesize their types.

1 Setting and basics

Our base theory will be the Higher Observational Type Theory with an infinite tower of cumulative universes `Type : Type`$^{+}$ `: Type`$^{++}$ `: ⋯` featuring $\Box$-modality-based polymorphism.

All universes will be closed under dependent product, dependent sum types, and quotient inductive types.

The simplest types of this kind are the finite datatypes (also known as enums) defined by enumerating their possible values:[6]

```
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:[7]

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

Each inductive type comes along with a dual typeclass:[8]

```
data Bool^R<this Y>(ifTrue  : Y,
                    ifFalse : Y)

data Possibly^R<X, this Y>(ifNothing : Y,
                           ifValue(x : X) : Y)
```

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

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

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

[8] 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. `Bool`$^{\mathtt{R}}$ `<: Type`, so every `T : Bool`$^{\mathtt{R}}$ is both a type and an instance of `Bool`$^{\mathtt{R}}$`<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.

Inhabitants of inductive types `x : T` can be converted into functions[9] (known as Church representations) that evaluate their by-case analysers: `x^c : ∀<Y : T^R> Y`:

```
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:

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

Non-finite inductive types admit (strictly positive) recursion in type definitions, allowing to introduce 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.

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

Type-valued functions on natural numbers `Y : Nat → U` can encode arbitrary predicates, and a dependent `Nat`-analyser $\texttt{Nat}^{\texttt{M}}\texttt{<Y>}$ encodes an induction motive: it establishes a proof of the 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, e.g. 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
  EmptySizedList : Vec<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
  Fst<size> : size⁺
  Nxt<size>(prev : size) : size⁺
```

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

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

data FinSet<T><size : Nat>(items : size → T)
  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.

2 Lax types: internalizing inverse 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 `ZeroEndingSequence` 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(n : Nat) : ℓType
  LaxNat(n) [m : Nat⟩ LaxNat(n + m)
  [n⟩ [m⟩ ↦ [(n +) m⟩
```

To each universe `U` we'll have an associated lax universe `$U` 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 : $U` 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 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). Putting everything together, lax types form *strictly associative inverse 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 `SizedZeroEndingSequence` as desired.

```
data SizedZeroEndingSequence : 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 type families are functions on their index type, so they must act on the extenders: they must map them either to identities or extenders between function results. If we deal with type-valued functions on lax types `S → U`, extenders can only be mapped to identities, but type families `Sᵈ` are more than type-valued functions: they allow mapping extenders to extenders between types which we define as follows: for types `X Y : U`, the type `X ↑ Y` is a pair of a function `b : Y → X` and domain extension operator `e : ∀<Z> (X → Z) → (Y → Z)` so that for every `f : X → Z`, we have equality by construction (definitional equality) `b ∘ e(f) = f`.

Let `F : Iᵈ` be a type family, and `e : s ↑ t` for some `s t : I`. Then `F(e) : ∀<Y> (F(s) → Y) → (F(t) → Y)`. We also have a dependently typed version.

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

Now we can fill in the gap in the definition of `ZeroEndingSizedSequence`. The type of the equality generator `f = append(f, 0)` does not typecheck yet, but we can decompose it into an application[10] `{ it = append(f, 0) } f` and apply the domain extension to the function part by applying `ZeroEndingSizedSequence n[extend 1⟩`:

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

  expand : EvZeroSeq 0[extend 1⟩ { Prepend(0, Zeros) = it } Zeros
```

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

3 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 simply called `Monoid<T>`.

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's 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 the lax formulation provides an explicitly confluent system of rules making the theory stratified. Stratifiability of the sort algebra is necessary for generalized algebraic theories to have explicit syntactic free models and an effective model structure on the category of their models.

4 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 ↓ 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"[11] by N. Ghani et al.

Type families `T : Xᵈ` can be fibered over type families `Y : Xᵈ`. For this case, we'll 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>`.

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

5 Matryoshka types: internalizing direct categories

So far we only applied the operator ($^{\mathsf{d}}$) to types `T : U`, but this operator has been introduced in Displayed Type Theory for all terms, including type families `F : B`$^{\mathsf{d}}$ 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 ($^{\mathsf{d}}$) 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 𝔻 : Type ↓ 𝔻^d
  Fst / Void
  Snd / data
    Dep : |Snd| Fst
```

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

Let us now consider a type family `Y : (𝔻 / |·|)`$^{\mathsf{d}}$. 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 $\mathbb{D}$, so $|\mathtt{Snd}|^{\mathsf{d}}$ 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 Δ2⁺ : Typeℓ    // Shorter notation for T : T ↓ T^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 Δ2⁺ : Typeℓ
  El1 / Void
  El2 / data
    Dep : |El2| El1
  El2 / data
    Dep1 : |El3| El1
    Dep2 : |El3| El2 Dep1
```

For the whole thing to typecheck indexes of the types `|x|` should be structurally smaller than `x`. As you now see, such types form strictly associative direct 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 inductive type families `Prototype : V^d`. Categories themselves have the vocabulary

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

A foundational infinite example is the semi-simplicial shape type

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

Type families over Δ are known as semi-simplicial types and represent infinite sequences of sequentially dependent types

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

6 Reedy types: internalizing Reedy categories

Reedy categories are a joint generalization of direct and inverse categories, and can be represented by lax matryoshka inductive types which we'll call reedy types from now on.

In particular, we can add extenders to Δ to ensure that functions on `Tn` can be also applied to `Tn+1`. Let us start with an incomplete definition:

```
data Δ  : ℓTypeℓ
  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'⟩
```

Extenders define type families on a fibered type, so they have to specify action on selectors. In this way, we'll specify intertwining identities between selectors and extenders (i.e. face and degeneracy maps as they are known for geometric shapes). For the complete definition with detailed description, see Appendix I.

Type families on Δ are the infamous simplicial types essential for dependently typed theories.

7 Categories as models of a reedy prototype

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

```
data Cell2 : ℓTypeℓ
  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> extends Monoid<Ts.Ob> {}
data LaxMonoidalCat<this Ts : Cat> extends LaxMonoid<Ts.Ob> {}
```

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.

8 Displayed algebraic structures

The other nice thing is that since we have defined categories as models for an inductive type, we automatically have the typeclass of displayed categories, and all algebraic typeclasses are instances of it:

```
Group : Cat^d
Ring : Cat^d
Cat : Cat^d
```

Furthermore, we can iterate, and thus `Cat^d : Cat^dd`, etc. And since constructions and proofs also can be lifted, any statement we have proven for all small categories `prf<C : Cat>` also can be applied to displayed categories, say like the category `Grp : Cat^d` of all groups and the category of all categories `Cat : Cat^d` itself.

9 Universes of models are model categories with proarrows

Displayed models for inductive types have the form

```
data ℕ^Rd<M : ℕ^R, this Ts : M^d>(
  base : Ts(M.base),
  next : ∀{n : M} Ts(n) → Ts(M.next n)
)
```

allowing to define the type of motives `ℕ^M` for the induction operator `ℕ-ind`:

```
def ℕ^M<this P : ℕ^d> = ℕ^Rd<ℕ, P>

ℕ-ind<P : ℕ^M> : ∀(n) P(n)
```

For each model `M : ℕ^R`, the inhabitants `Pm : ℕ^Rd<M>` are promorphisms (many-to-many correspondences, sometimes also called weak homomorphisms) on `M` with the target given by

```
def Pm.target : ℕ^R<M × Pm>
  zero: (M.zero, Pm.base M.base)
  next: { n : M, x : (Pm n) ↦ (M.next n, Pm.next (M.next n) x) }
```

We can single out homomorphisms as the univalent (= many-to-one) promorphisms

```
def isUniv<src : ℕ^R, pm : ℕ^Rd src> = ∀{n} isContr(pm n),
```

making the type of `ℕ`-models into a ∞-precategory (Segal type), which turns out to be a ∞-category (Complete Segal type) due to a well-known fact that the equivalences `(≃)<ℕ^R>` of `ℕ`-models correspond to their isomorphisms.

Categories of models also carry a model structure that coincides with the one given by extenders between types and fibered types for ordinary universes `U` which can be seen as universes of models for the empty theory. For lax and/or generalized algebraic theories, they may exhibit non-invertible higher morphisms and thus form weak ω-categories. In particular, we expect to have an infinite typeclass hierarchy

```
ωCat : ωCat^d : ωCat^dd : ⋯
```

Together with □-modality based approach to polymorphism,[12] we expect to have a satisfying solution to all size issues arising in ordinary and higher category theory. In fact, we hope that the presented type theory is capable of eventually formalizing the nLab[13] in its entirety.

[12] https://akuklev.github.io/polymorphism
[13] 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 lax universes `𝓤` using an approach modelled after "Type Theory for Synthetic ∞-categories" by E. Riehl and M. Shulman.

Besides lax inductive types, lax universes are also populated by large types equipped with appropriate structure. As we have seen above, not only Reedy types are equipped with extenders and selectors. It also applies to 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.