Higher Categorical Type Theory

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Building on the unpublished ideas of C. McBride and ideas from "Displayed Type Theory and Semi-Simplicial Types" by A. Kolomatskaia and M. Shulman, we propose a novel extension for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) that allows internalizing Reedy categories.

Indexing and fibering over shape 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 a 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 own syntax as a generalized inductive type and thus also performing structural induction over it.

1 Why do we need shape-indexed type families?

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

```
data ExpressionKind
  Numeric
  Logical

data Expr : ExpressionKind → *
  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
  And(a b : Expr Logical)  : Expr Logical
  Not(a : Expr Logical)    : Expr Logical
```

This approach scales up to languages that may declare and bind named entities (variables, constants, internal types) 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 that enabling type-driven design of correct-by-construction analysers and interpreters. This includes type checking, compilation, control flow analysis, as well as static analysis and abstract interpretation in general.

As for IDEs, inductive type families enable designing biparsers for those languages, i.e. parsers that maintain 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/

To represent languages with typed variables, one introduces the type `Ty` representing variable types in the language, and the type family `Tm : ↓Ctx` 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 shape type `LaxNat` that enables context extension and selection of subcontexts.

In case of dependently typed languages, the we'll have a type family `Ty : ↓Ctx` 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 shape 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` and the type family `Tm : ↓(c : Ctx, Ty c)` of terms of a given type in a given context.

Bi-directionally typed languages (computational type theories) also require a fibered type family `Redex : ↓(c : Ctx) ↑(ty : Ty c, Tm ty)` of reducible expressions that synthesize their types.

2 Setting and basics

Our base theory will be the Higher Observational Type Theory with an infinite tower of cumulative universes $* : *^+ : *^{++} : \cdots$ 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:

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

data Unit
  Point

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

We can generalize them to sum types by allowing infinite families of "possible values" parametrized by some other type:

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

Each inductive type comes along with a dual typeclass:

```
typeclass Bool^R<this Y>
  true  : Y
  false : Y

typeclass Possibly^R<X, this Y>
  nothing : Y
  value(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 evaluating their by-case analysers: `x^c : ∀<Y : T^R> Y`:

```
def True<Y : Bool^R>^c = Y.true
def False<Y : Bool^R>^c = Y.false

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

These functions are known as Church representations.

What if we want to return values of different types for `True` and `False`? If we have universes (types of types), we can first define a function from booleans into some universe `R : Bool → U` and then a dependent case analyser

```
typeclass Bool^M<this Y : Bool → *>
  true  : Y(True)
  false : 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`
  Next(pred : ℕ) `pred⁺`

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:

```
typeclass Nat^R<this Y>
  zero : 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 $\texttt{Nat}^{\texttt{M}}\texttt{<Y>}$ 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 by their generators, quotient inductive types may additionally contain constructors of identities between inhabitants.

This way we can define rational numbers and unordered collections:

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

data Collection<T>
  bagOf<n : FinType>(items : n → T)
  permute<n, items>(p : n!) : bagOf(items) = bagOf(items ∘ p)
```

where `n!` is the type of automorphisms on the type `n`, i.e. permutations in case of finite types.

That is, 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.

3 Type families and inverse categories

For a type `J : U` let $\texttt{J}^{\texttt{d}}$ denote the respective universe of type families indexed by `J`. A typical example is length-indexed lists:

```
data Vec<T> : Natᵈ
  nil : Vec<T> 0
  cons<n>(head : T, tail : Vec<T> n) : Vec<T> n⁺
```

Now consider the quotient inductive type of eventually-zero sequences:

```
data ZeroEndingSequence
  nil : ZeroEndingSequence
  append(prefix : ZeroEndingSequence, head : Nat)
  extend(f : ZeroEndingSequence) : f = append(f, 0)
```

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

```
shape LaxNat
  lax(n : Nat) : LaxNat
  lax(n) [m : Nat⟩ lax(n + m)
  [n⟩ [m⟩ ↦ [(n +) m⟩
```

To each universe `U` we'll have an associated shape universe `$U` occupied by the types like the one above. Inductive shape types are stratified directed counterparts of quotient inductive types. For every pair of their elements `x y : T` of a set-like type `T : U` there is a type `(x = y) : U` of identifications between `x` and `y`.

Shape types `S : $U` admit extender types instead: for every element `s : S`, there is a type family $\texttt{s↑ : P}^{\texttt{d}}$. We will write `s ↑ t` for `s↑ t`.

Quotient inductive types admit constructors of identities `x = y` between their elements. Shape types allow constructors 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).

This way, shape types form strictly associative inverse categories.

Every function we define on a shape type must have an action on all constructors, including extender constructors, which amounts to functoriality.

To have an example, let us define addition for `LaxNat`s:

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

With `LaxNat` we can transform `ZeroEndingSequence` into a type family:

```
data ZeroEndingSizedSequence : ↓LaxNat
  nil : ZeroEndingSizedSequence lax(0)
  append<n>(prefix : ZeroEndingSizedSequence n, head : Nat)
  : ZeroEndingSizedSequence (lax(1) + n)

  extend<n>(f : ZeroEndingSizedSequence n) : ???
```

Before we fill in the gap in the above definition, note that type families also seem to be functions on their index type, so they must act on the extender constructors: they must map extender constructors to identities or extenders between function results. If we deal with type-valued functions on shapes `S → U`, extenders can only be mapped to identities, but type families `S^d` 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 : I^d` 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 constructor `f = append(f, 0)` does not typecheck yet, but we can decompose it into an application `{ it = append(f, 0) } f` and apply the domain extension to the function part by applying `ZeroEndingSizedSequence n[extend 1⟩`:

```
data ZeroEndingSizedSequence : LaxNat^d
  nil : ZeroEndingSizedSequence lax(0)
  append<n>(prefix : ZeroEndingSizedSequence n, head : Nat)
  : ZeroEndingSizedSequence (lax(1) + n)

  extend<n>(f : ZeroEndingSizedSequence n)
  : ZeroEndingSizedSequence n[extend 1⟩ { it = append(f, 0) } f
```

4 Lax algebraic theories and shapes

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:

```
shape Monoid^P
  compose : LaxMonoid^P* → LaxMonoid^P

  expand(xs : LaxMonoid^P*, pr : Parenthesization xs.length)
  : 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 shape type with extenders instead of identities. Algebraic theories with extenders are known as lax algebraic theories.

```
shape LaxMonoid^P
  compose : LaxMonoid^P* → LaxMonoid^P

  compose(xs) [pr : Parenthesization xs.length⟩ (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.

5 Fibered types and direct categories

Many operations on containers have the following property: the shape of the resulting container only depends on the shapes of the arguments. For example, length of the list computed by `concatenate`, `map`, and `reverse` can be computed based on the lengths 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 `length`: `T* / length : 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:

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

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

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 : * ↓ *
  unit / Unit
  bool / Bool
  pi(X : V, Y : X → V) → V / ∀(x : X) Y(x)
```

We will use `|_|` as the default name for the fibering function unless it is explicitly named.

Fibered types allow formulating dependent extender types: for a type `X : U` and a fibered type `Y : Y' / X`, extenders `X ↑ Y` are terms of the type `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. On one hand, 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>`.

Above we only used the operator `( ᵈ)` on types `T : U` to denote type families `Tᵈ`, but this operator was actually introduced in "Displayed Type Theory and Semi-Simplicial Types" for all terms. Let us extend its definition to fibered types as follows.

For `Y : (F / f)ᵈ`, where `f : F → B`, and `x : F` let:

```
Y(x) : Bᵈ (f x) Y
```

The significance of this definition comes to light when we consider that inductive types can be self-fibered:

```
shape 𝔻 : * ↓ 𝔻
  fst / (Void / exfalso)
  snd / (Unit / { fst })
```

Here we define a type with two generators `fst` and `snd`, and a function `|x : 𝔻| : (* ↓ 𝔻)`, i.e. for every generator `c` we have a type `|c|` fibered over 𝔻. For `fst`, this type `|fst|` is empty. For `snd`, `|snd|` is a unit type together with a function mapping its unique element to `fst`.

Inductive self-fibered types form strictly associative direct categories. (TODO: Clarify)

A type family `Y : ↓(𝔻 / |·|)` indexed over this type satisfies the following typing rule:

```
Y(x : 𝔻) : (* ↓ 𝔻)ᵈ x Y
```

Since 𝔻only has two elements, we can split cases:

```
Y(fst) : (* ↓ 𝔻)ᵈ fst Y
Y(snd) : (* ↓ 𝔻)ᵈ snd Y
```

which in turn reduces to

```
Y(fst) : (* ↓ 𝔻)ᵈ (Void / { it }) Y
Y(snd) : (* ↓ 𝔻)ᵈ (Unit / { fst }) Y
```

further reducing to

```
Y(fst) : (∀(u / f : (Void / { it })) Y(f(u))) → 𝔻
Y(snd) : (∀(u / f : (Unit / { fst })) Y(f u)) → 𝔻
```

Product over empty domain is `Unit`, and the product over unit domain is just one element:

```
Y(fst) : Unit → 𝔻
Y(snd) : Y(fst) → 𝔻
```

So our type family is merely a dependent pair `Σ(T : 𝔻) (T → 𝔻)`!

With self-fibered index types we can define dependent pairs as dependent function types. Signatures of theories with dependent sorts can be expressed as finite direct categories, so first-order and higher-order theories with dependent sorts can be expressed as type classes parametrized by such families. Algebraic theories with dependent sorts can be expressed via inductive type families indexed over a finite self-fibered index type S. In particular, categories are models of an algebraic theory over the shape

```
data Cell2⋆ : * ↑ Cell2⋆
  Ob  / (Void / exfalso)
  Mor / (Bool / { Ob })
```

To deal with ∞-categories, one can introduce a shape type `Cell` containing cell types of every dimension `n : Nat`.

Dually to our lax natural numbers, we can introduce a self-indexed type `Δ⋆`:

```
data Δ⋆ : * ↑ Δ⋆
  simplex(n : Nat) / ((Σ(m) Fin(m) → Fin(n)) / simplex(m))
```

Type families over Δ are semi-simplicial types. Type families over thin (i.e. with at most one arrow between any two inhabitants) self-indexed types are also known as very dependent types.

6 Combining extenders and selectors: Reedy categories

Most notably, we can combine extenders (degeneracy maps) and selectors (face maps) yielding strictly associative Reedy categories like the simplicial category Δ:

```
shape Δ  : * ↑ Δ
  simplex(n : Nat) / ((Σ(m) Fin(m) → Fin(n)) / simplex(m))

  simplex(n)[m : Nat, f : Fin(m) → Fin(n)⟩ simplex(m) / (intertwining identities)
  [m, f⟩ [m', f'⟩ ↦ [m', { it ∘ f } f'⟩
```

Type families on Δ are the infamous simplicial types, which are vital for defining the syntax of dependent typed theories.

7 Categories as models of a shape-indexed prototype

Let us revisit the category signature shape, adding an extra extender:

```
shape Cell2 : * ↑ Cell2
  Ob  / (Void / exfalso)
  Mor / (Bool / { 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 Cat^P : Cell2^d
  id{o : Cat^P Ob} : (Cat^P Mor)(o, o)
  (►){x, y, z} : (Cat^P Mor)(x, y) → (Cat^P Mor)(y, z) → (Cat^P Mor)(x, z)

  unitorL{x, y} : ∀(f : (Cat^P Mor)(x, y)) f = id ► f
  unitorR{x, y} : ∀(f : (Cat^P 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:

```
typeclass Cat<this Ts : Cell2^d>
  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.Ob^n → 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:

```
typeclass MonoidalCat<Ts : Cat> extends Monoid<Ts.Ob> {}
typeclass LaxMonoidalCat<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^d^d`, 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

```
typeclass ℕ^R^d <M : ℕ^R, this Ts : |M| → *>
  zero : Ts(M.zero)
  next : ∀{n : M} Ts(n) → Ts(M.next n)
```

allowing do define the type of induction motives and the induction operator:

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


ℕ-ind<P : ℕ^M>(n : ℕ) : P(n)
```

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

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

We can single out homomorphisms as the functional (= many-to-one) promorphisms `Σ(src : ℕ^R<T>, pm : ℕ^R^d src) (f : ∀(n) (m : (pm n)) × ∀(n' : pm n) n ≃ m`, making the type of ℕ-models into a ∞-precategory (Segal type), which turns out to be a ∞-category (Complete Segal type) as it is well-known that the equivalences `(≃)<ℕ^R>` of ℕ-models correspond to their isomorphisms.

Categories of models also carry a weak model structure. Models of lax algebraic theories and dependently sorted algebraic theories can also have directed higher structure, and in general form weak ω-categories. In particular, we expect to have an infinite typeclass hierarchy

```
ωCat : ωCat^d : ωCat^d^d : ⋯
```

Together with □-modality based approach to polymorphism,[5] 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[6] in its entirety.

[5] https://akuklev.github.io/polymorphism
[6] http://ncatlab.org

10 (Pre)sheaf universes as categories: Pursuing stacks

As we have seen above, not only inductive shapes have the notion of extenders and selectors (i.e. are weak model categories); universes do as well. It is not hard to see that it also applies to universes of fibered types, universes of type families ("presheaf universes"), and universes of fibered type families. Conjecturally, it also applies to sheaf universes.

As we have seen above, it also applies to universes of models for any given algebraic theory, including infinitary algebraic theories with dependent sorts and their generalized form as long their sort algebras are stratified. In fact, in all of these cases, the categories `C` are also equipped with proarrows ("multivalued morphisms") $s^d\ t$ for each $s\ t : C$.

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 shape universes `$U` using an approach modelled after "Type Theory for Synthetic ∞-categories" by E. Riehl and M. Shulman.

As universes of lax or dependently sorted algebraic theories carry non-invertible higher morphisms, ultimately we shall be pursuing stacks.