Formalizing Category Theory and Presheaf Models of Type Theory in Nuprl

April 16, 2018

1 Introduction

This article is the first in a series of articles that explain the formalization of a constructive model of *cubical type theory* in Nuprl. The main features of cubical type theory that will concern us at first are that in addition to the **Π** and **Σ** types of dependent functions and dependent pairs, there will be a new type $\mathbb{I}$ for the *interval*. When there are several variables `i,j,k:`$\mathbb{I}$ in the context, then those parameters range over (n-dimensional) *cubes*. The theory will also have *formulas* about these variables like `(i=0 and j=1)` or more complex formulas including clauses like `k = min(i,j)`. If we call the whole n-dimensional cube `X` and the subspace defined by the formula `S`, then if we add a new variable `v:`$\mathbb{I}$, we get spaces `X+ = X` $\times$ $\mathbb{I}$ and `S+ = S` $\times$ $\mathbb{I}$, and the subspace of `X+` for `v=0` is a copy of `X` we can call `X[0]`. If the subspace `S+` $\cup$ `X[0]` is a *homotopy retract* of the whole cube `X+` then any continuous function defined on `S+` $\cup$ `X[0]` can be extended to the whole space. Daniel Kan repeatedly used this *homotopy extension principle* in his development of homotopy theory. The principle is expressed synthetically in the rules of cubical type theory and to give a constructive model of cubical type theory we must produce the witnesses (i.e. the programs) for these extension operations. This is accomplished by endowing each type in the theory with a *composition operation* (from which the extension operation can be derived). Types with a composition operation are called *fibrant* (or said to satisfy the *Kan property*). When all the types in the *universe* are fibrant and the universe is itself fibrant then it will be possible to verify Voevodsky's *univalence axiom*. Showing that all the types are fibrant (especially the universe) is the most technically demanding part of construction of a constructive model of cubical type theory and is the topic of later articles in this series.

Eventually we will define the *paths* in a type `T` to be the functions of type $\mathbb{I} \rightarrow$ `T`, and a path with *endpoints* `a` and `b` will witness equality of `a` and `b`. This will give a new meaning for equality that has non-trivial computational content. It differs from the meaning of equality in Nuprl, and a constructive model of this theory is highly non-trivial.

The model we formalized is due to Bezem, Coquand, Cohen and Mörtberg but starts with a construction due to Hofmann and Streicher, the *presheaf models* of Martin Löf type theory. The point of the current article is to explain this construction and how it is formalized in Nuprl. A by-product of this explanation will be a sort of *Rosetta stone* relating the category-theoretic viewpoint and language to the purely type-theoretic viewpoint and language used in Nuprl. So, for example, a *presheaf* is seen to be the same as a *family of Nuprl types* together with a family of maps between them that satisfy certain equations. The category-theoretic language can say a lot with one word like *presheaf* while the type theoretic definitions are concrete (and understood by Nuprl tactics!). Those who already know the category theory can see how it is expressed in Nuprl, while those who know a type theory like Nuprl's can learn what the category theory means.

Formalizing category theory in Nuprl is fairly straightforward. Each category will be small—in the sense that it is a type in some Nuprl universe. But since Nuprl has an infinite cumulative hierarchy of universes that does not seem to limit the theory.

We have formalized basic concepts of category theory sufficient for our construction of a formal model of cubical type theory. For this we mainly needed the concept of a *presheaf*. So we need categories, functors, natural transformations, the opposite of a category, and a "category of sets". For the latter, we substitute the category of Nuprl types (in a universe). Because Nuprl's types are extensional, the definition of presheaf that we get has the properties needed to build constructive models of type theory and to prove things like the Yoneda lemma—that the Yoneda embedding is a full-faithful functor. We have also defined adjoint functors, monads, Kleisli category, groupoids, comma categories and many other concepts, and constructed examples such as the adjunction `FreeGroup -| ForgetGroup`.

Once we have formalized basic category theory, the first step in constructing a model of cubical type theory is to construct a presheaf model of basic Martin Löf type theory. This construction, due to Hofmann and Streicher, interprets the basic concepts of MLTT: *context, type, term* and its judgments, using for contexts presheaves over any base category `C`. This generic construction also defines the Π and Σ types and the basic terms for λ-abstraction and pairs with first and second components (it also gives an

interpretation of equality types but for cubical type theory we will use path types instead).

The model for cubical type theory starts with this basic construction and specializes it to use a particular `CubeCat` (the *cube category*) for the base category. In this document we discuss only the parts of the formalization that do not depend on the choice of base category. So, it spells out how we make the first steps of our formalization of cubical type theory.

2 Categories

A category has a set or class of objects. For objects x and y there is a set or class of arrows $x \to y$. For each object x there is an identity arrow $x \to x$, and arrows $x \to y$ and $y \to z$ can be composed to get an arrow $x \to z$. We replace 'set or class' by type (in a universe), so our formal definition in Nuprl has a universe *level parameter* i.

There are several ways to build structures in Nuprl's type theory, but to define categories we use the most straightforward one. We make a dependent product for the objects, arrows, identity, and composition, and then, using Nuprl's refinement type (also called the *set type*), form a subtype of the dependent product for which certain *identity* and *associativity* equations hold. We write either `Type{i}` or $\mathbb{U}${i} for the `i`th universe.

```
SmallCategory{i}  ==
  {cat:ob:Type{i}
    × arrow:ob → ob → Type{i}
    × x:ob → arrow(x, x)
    × x:ob → y:ob → z:ob
       → arrow(x,y)→ arrow(y,z)→ arrow(x,z) |
    let ob,arrow,id,comp = cat
    in ∀x,y:ob. ∀f:arrow(x,y). comp(x,x,y,id(x),f) = f
                               ∧ comp(x,y,y,f,id(y)) = f
    ∧ ∀x,y,z,w:ob. ∀f:arrow(x,y) ∀g:arrow(y,z). ∀h:arrow(z,w).
          comp(x,z,w,comp(x,y,z,f,g),h) =
          comp(x,y,w,f,comp(y,z,w,g,h))
  }
```

The four components of a category `C` are `ob(C)`, `arrow(C)` , `id(C)`, and `comp(C)`. To make a category we supply its four components using a `mk-cat` operator displayed as follows:

```
Cat(ob               =  ob;
```

```
    arrow(x,y)        =  arrow[x;y];
    id(a)             =  id[a];
    comp(u,v,w,f,g)   =  comp[u;v;w;f;g])
```

The expressions on the right of the equal signs are second order variables. The result is a category if the identity and associativity equations hold for the given expressions. For example the *discrete category* for a type `X` is

```
discrete-cat(X)  ==
Cat(ob= X; arrow(x,y)=  x=y; id(a)= ·; comp(u,v,w,f,g)= ·)
```

This is the category with objects `X` and arrows only between equal members of `X` (the equality type in Nuprl is inhabited only by `·`, sometimes written `Ax`).

A somewhat more interesting category is the category of types:

```
TypeCat{i}  ==
Cat(ob= Type{i};
   arrow(I,J)=  (I → J);
   id(I)= λx.x;
   comp(I,J,K,f,g)= (g o f) )
```

This category has a universe level parameter `i` and its objects are the Nuprl types in universe `i`, `I → J` is the Nuprl function type, and `g o f` is `λx.g(f(x))`. We use this category as a replacement for the category of sets.

The *opposite* of category `C` is

```
OpCat(C)  ==
Cat(ob= ob(C);
   arrow(I,J)=  arrow(C)(J,I);
   id(I)= id(C)(I)
   comp(I,J,K,f,g)= comp(C)(K,J,I,g,f) )
```

It simply reverses the direction of all the arrows.

The category of groups is:

```
Group  ==
Cat(ob= Group{i};
   arrow(G,H)=  MonHom(G,H);
   id(G)= λx. x;
   comp(I,J,K,f,g)= (g o f) )
```

`MonHom(G,H)` is the type of *monoid homomorphisms*, i.e. maps that preserve the group identity and group operation.

3 Functors

A *functor* F between categories C1 and C2 is a member of the following type:

```
Functor(C1;C2)  ==
  {FM:F:ob(C1) → ob(C2) ×
       (x:ob(C1) → y:ob(C1) →
          (arrow(C1) x y) → (arrow(C2) (F x) (F y))) |
     let F,M = FM in
     ∀x:ob(C1). M(x,x,id(C1)(x)) = id(C2)(F x)  ∧
     ∀x,y,z:ob(C1). ∀f:arrow(C1) x y.  ∀g:arrow(C1) y z.
      M(x,z,comp(C1)(x,y,z,f,g)) =
      comp(C2)(F(x),F(y),F(z,M(x,y,f),M(y,z, g))}
```

F has two components ob(F) and arrow(F), where ob(F) maps objects of C1 to objects of C2, and arrow(F) maps arrows of C1 to arrows of C2. The functor must map identity arrows in C1 to identity arrows in C2 and map the composition of arrows in C1 to composition of arrows in C2. We display ob(F)(x) as F(x) and display arrow(F)(x,y,a) as F(x,y,a). To construct a functor we use mk-functor which is displayed

```
functor(ob(a)= ob[a];
       arrow(x,y,f)= arrow[x;y;f] )
```

where the expressions on the right of the equal signs are second order variables. For example, the identity functor (which we display as 1) is functor(ob(x)=x; arrow(x,y,a)=a). Composition of functors F and G is the functor

```
functor(ob(x)= G(F(x)); arrow(x,y,a)= G(F(x),F(y),F(x,y,a)))
```

The identity functor and functor composition satisfy the equations needed to define the *category of categories*

```
CatCat{i}  ==
Cat(ob= SmallCategory{i};
   arrow(A,B)=  Functor(A,B);
   id(A)= 1;
   comp(A,B,C,F,G)= functor-comp(F,G) )
```

This category is a member of the type SmallCategory{i+1}.

A functor F ∈ Functor(C,D) is *full and faithful* if for any x,y ∈ob(C), arrow(F) is a bijection between arrow(C)(x,y) and arrow(D)(F(x),F(y)).

4 Natural Transformations

A *natural transformation* between two functors `F` and `G` in `Functor(C,D)` is a function that assigns to each object `A` in category `C` an arrow in category `D` between `F(A)` and `G(A)` for which a certain diagram commutes (i.e. a certain *naturality* equation holds). Thus a natural transformation is a member of the type:

```
nat-trans(C;D;F;G)  ==
 {trans:A:ob(C) ⟶ arrow(D)(F(A), G(A)) |
   ∀A,B:ob(C). ∀g:cat-arrow(C)(A,B).
   comp(D)(F(A),G(A),G(B),trans(A),G(A,B,g))
   = comp(D)(F(A),F(B),G(B),F(A,B,g),trans(B))
 }
```

A natural transformation `T` is simply a function $\lambda x.T(x)$, but we use a special operator `mk-nat-trans` displayed as `x |⟶ T[x]` to tell the system to typecheck it as a natural transformation. For example, the identity natural transformation is

```
identity-trans(C;D;F) ==  A |⟶ id(D)(F(A))
```

Composition of natural transformation `t1 ∈ nat-trans(C;D;F;G)` with natural transformation `t2 ∈ nat-trans(C;D;G;H)` is:

```
t1 o t2 ==  A |⟶ comp(D)(F(A), G(A), H(A), t1(A), t2(A))
```

This composition operator is really `trans-comp(C;D;F;G;H;t1;t2)`, but we "hide" the parameters `C,D,F,G` and `H` in the display form and display only `t1 o t2`.

The identity natural transformation and the composition operation satisfy the equations needed to define the *category of functors*

```
FUN(C1;C2)  ==
 Cat(ob= Functor(C1;C2)
    arrow(F,G) = nat-trans(C1;C2;F;G)
    id(F) = identity-trans(C1;C2;F)
    comp(F,G,H,t1,t2)=  t1 o t2 )
```

5 Presheaves

A *presheaf* over a category `C` is a functor from `OpCat(C)` to the category of sets. We substitute the category of types (in universe `i`) for the category of sets. Thus

```
Presheaf(C){i} == Functor(op-cat(C);TypeCat{i})
```

This is a type in Nuprl universe `i+1`. The presheaves over `C` form a category

```
Presheaves(C){i} == FUN(op-cat(C);TypeCat{i})
```

This is a member of the type `SmallCategory{i+1}`.

Since a presheaf is a functor, to construct one we must give the two components. We write this

```
Presheaf(Set(I) = S[I]
        Morphism(I,J,f,rho) = morph[I,J,f,rho]
        )
```

The expression `S[I]` specifies the "set" (i.e. the type) assigned to object `I` from category `C`. The expression `morph[I,J,f,rho]` specifies how to map the set `S[I]` to the set `S[J]` when there is an arrow `f:J` $\rightarrow$ `I` in category `C` (the arrow is reversed because the presheaf is a functor from the opposite of `C`) by giving the image of `rho` $\in$ `S[I]` under the mapping. This map is called the *restriction map* and we display it as simply `f(rho)`, but it really has parameters `H,I,J,f,rho` where `H` is the presheaf.

For example, the *representable presheaf* for `X` $\in$ `ob(C)` is

```
Yoneda(X) ==
Presheaf(Set(I) = arrow(C)(I,X)
        Morphism(I,J,f,a) = comp(C)(J,I,X,f,a)
        )
```

This presheaf assigns to each object `I` $\in$ `C` the set of arrows `I` $\rightarrow$ `X`. Given an arrow `f:J` $\rightarrow$ `I`, composition with `f` maps an arrow `a` $\in$ `I` $\rightarrow$ `X` to an arrow `b` $\in$ `J` $\rightarrow$ `X`, and the equations necessary for this to define a presheaf hold.

We call this presheaf `Yoneda(X)` because it is the first component of the *Yoneda embedding*:

```
Functor(ob(X) = Yoneda(X)
        arrow(X,Y,f) = A |→ λ g. comp(C)(A,X,Y,g,f)
        )
```

For any category `C`, this defines a functor from `C` to the category of presheaves over `C`. The *Yoneda lemma* states that this functor is full and faithful. The proof of this lemma in Nuprl was straightforward. It does make use of some of the *extensional* properties of Nuprl's type theory, which we will discuss briefly in the next section.

6 Extensional reasoning in Nuprl

There are at least two different meanings for the adjective *extensional* in type theories. One is *function extensionality* and the other is sometimes phrased as "*propositional equality is definitional equality*". We explain how Nuprl's type theory is extensional in both of these senses.

Every Nuprl proof is built up (by using *tactics*) as a tree of *primitive inferences* that are instances of the *rules*. A rule matches the current *goal sequent* with a given *goal pattern* and then, using some (possibly empty) list of *parameters* (supplied by the tactic), generates the instances of the *subgoal patterns*. The rule is true when the *truth* of any instance of the goal follows from the truth of the instances of the subgoals. The formal definition of correctness for Nuprl rules thus depends on the formal definition of *truth* of a Nuprl sequent. This in turn depends of the formal definition of the Nuprl type system. All of this has been formalized in Coq and is well beyond the scope of this document. Here we want to show the two rules that are true for the Nuprl type system and show that it is extensional in both senses.

Function Extensionality

```
H ⊢ f = g ∈ (x:A ⟶ B)

    BY  functionExtensionality  !parameter{i:l}  u

    H  u:A  ⊢  f(u) = g(u) ∈ B[u/x]
    H ⊢  A = A ∈ Type{i}
```

This rule says that to prove `f` is equal to `g` in a (dependent) function type `x:A ⟶ B` it is enough to prove that the domain `A` is a type and that for every `u:A`, `f(u)` and `g(u)` are equal in `B[u/x]` (the proof of that subgoal will also establish that `B[u/x]` is a type). This, the mathematical definition of function equality, is true because of the formal definition of the (dependent) function type in the Nuprl type system. Note that all items in the generated subgoals come from matching the variables `H, f, g, x, A`, and `B` with the the goal, except for the universe level `i` and the auxiliary variable `u` that are supplied as parameters. In this rule, all the judgments are equality types which have no constructive content in Nuprl. Hence, no extract terms are specified (more precisely, the default extract `Ax` is used).

Type Extensionality If types `A` and `B` are provably equal (i.e. *propositionally equal*) and `t ∈ A` is it true that `t ∈ B`? In *intensional* type theories

this is usually not true unless A and B are *definitionally* equal. When that is not so, and p is the proof of A = B, some sort of coercion function like transport(p,t) ∈ B is needed. In Nuprl we can prove that t ∈ B without applying any coercion.

This follows from the more general rule shown here:

```
H  x:A, J  ⊢  C  ext  t

    BY  hyp_replacement  #j  B  !parameter{i:l}

    H  x:B, J  ⊢  C  ext  t
    H  x:A, J  ⊢  A = B ∈ Type{i}
```

In this rule, the parameter #j is the hypothesis number of the declaration x:A in the context. Because the conclusion C may have constructive content, the *extract term* t is specified. In this case the rule says that the term extracted from the proof of the original goal will be the term extracted from the proof of the first subgoal. The hyp_replacement rule says that in any context if type A is provably equal to type B (in some universe i) then we can replace a declaration x:A in the context with the alternate declaration x:B (where B is supplied as a parameter). Not only is the original goal true when the alternate goal is true, but the extract term t is the same because extracts are terms in an untyped programming language.

Subtype reasoning Another distinctive feature of Nuprl's type theory is that the *subtype relation* A ⊆ B is defined by λx.x ∈ A →B and is a proposition (i.e. a type) whenever A and B are types.[1] The Nuprl library contains many lemmas about subtypes and reasoning about subtypes is a signifcant part of the Auto tactic.

The main way that type extensionality is used in Nuprl is via the lemma subtype-rel-equal:

```
∀[A,B:Type]. A ⊆ B  supposing A = B
```

The proof of this lemma uses the hyp_replacement rule.

For any lemma proved in Nuprl, the system can tell us which lemmas and which primitive rules were used in its proof. For the Yoneda lemma, we found that the functionExtensionality rule was used and the

[1] Membership t ∈ T is just the equality type t = t ∈ T, but it is an interesting feature of Nuprl that sometimes (as when t is λx.x) an equality is well-formed whether it is true or not.

`subtype-rel-equal` lemma was used. So the Nuprl proof of the Yoneda lemma uses both kinds of extensional reasoning. We have not investigated whether both are absolutely necessary for this proof because we are working in Nuprl, so there is little point in knowing whether the natural proof of a fact can be redone to avoid using some rule or other.

7 Presheaf models of Martin Löf Type Theory

Martin Löf type theory (MLTT) has the following primitive concepts expressed as *judgements* of the formal theory.

- $(H \vdash)$ says that H is a well-formed *context*.
- $(H \vdash T)$ says that T is a well-formed *type* in context H.
- $(H \vdash t{:}T)$ says that t is a well-formed *term* of type T in context H.

A typical rule of MLTT would be that if $(H \vdash)$ and $(H \vdash T)$ then $(H, x{:}T \vdash)$ provided that x is a fresh variable. This rule says that we can add new declarations to a well-formed context, so starting with the *empty context* (which is well-formed) contexts are built up as lists of declarations where each type is well-formed in the preceding context. The well-formed types and terms in such a context are certain syntactic expressions mentioning the declared variables.

To model such a theory exactly we would have to formally define the syntax of the expressions and define free and bound variables, α-equality, and substitution. To avoid having to do this work, there is an alternate *name-free* syntax for MLTT. In this version rather than add $x{:}T$ to context H, we merely add type T to get the context $H.T$ and instead of the expression x in context $H, x{:}T$ a special term q refers to the last declaration of the context $H.T$. Rather than substitutions we use *context maps* $\sigma : H \to G$. We can apply such a context map σ to a type T to get $(T)\sigma$ and to a term t to get $(t)\sigma$. If $G \vdash T$ then $H \vdash (T)\sigma$ and if $G \vdash t{:}T$ then $H \vdash (t)\sigma{:}(T)\sigma$.

There is a polymorphic context map $p : (H.T) \to H$. Thus, the variables x, y, z in a context like $x : A,\ y : B,\ z : C$ correspond to the terms $((q)p)p,\ (q)p,\ q$ in the context $A.B.C$. If we think of q as zero and p as successor, then $((q)p)p$, $(q)p$, and q correspond to the numbers $2, 1, 0$ which we can see as *DeBruijn indices* that replace the bound variables x, y, z.

To build a model of this name-free version of MLTT we must start by giving the meanings of $(H \vdash)$, $\sigma : H \to G$, $(H \vdash T)$, $(H \vdash t : T)$, $(H.T)$, $(T)\sigma$, $(t)\sigma$, p, and q. After that we define the function and product types

(the Π and Σ types) and associated terms, and prove that these definitions satisfy thirty-nine rules of basic MLTT that we give later (at the end of this section and in section 8).

Context and context map It was realized by Hofmann and Streicher that for any base category `C` we can get a model of MLTT where context $(H \vdash)$ means that H is a presheaf over `C` and a context map $\sigma : H \to G$ is a natural transformation from presheaf H to presheaf G.

If we think about the Nuprl formalization of presheaf, we see that a presheaf over category `C` is a *family of Nuprl types* indexed by the objects in `C` together with a family of maps between these Nuprl types indexed by the arrows in `C` such that identity and composition are preserved.

We usually use letters `I, J, K` ... for objects of category `C`. Then if `H` is a presheaf over `C` then `H(I)` is a Nuprl type. We use letters α, ρ for members of a type like `H(I)` or `H(J)`, but in Nuprl syntax we have to write `alpha` or `rho`. An *object* of preheaf `H` is a pair of type `I:ob(C)` × `H(I)`, so it is a pair `<I, rho>` where `rho` ∈ `H(I)` (pairs in Nuprl are displayed with angle brackets). There is a natural way to define the arrows so that the objects of a presheaf `H` form a category, called the *category of elements of the presheaf.*

Types in a context One way to define the meaning of `H` ⊢ `T` for a presheaf `H` is that `T` is a presheaf over the category of elements of `H`. We chose to unpack this abstract definition and spell out what it means.

Such a `T` is a functor so it has two components. The first component, a family of Nuprl types indexed by the objects of `H`, has type `I:ob(C)` → `rho:H(I)` → `Type`. The second component is a family of maps between these types indexed by the arrows of `C`. When `f` ∈ `J` → `I` is an arrow in `C` then for `rho` ∈ `H(I)` we have `f(rho)` ∈ `H(J)` and this must induce a map from `T(I,rho)` to `T(J,f(rho))`. So the second component of `T` has type `I:ob(C)`→`J:ob(C)`→`f:(J`→`I)`→`rho:H(I)`→`T(I,rho)`→`T(J, f(rho))`. Applied to `I,J,f,rho` and `u` ∈`T(I,rho)` the second component of `T` gives a member of `T(J, f(rho))` and we write this as `T(I,J,f,rho,u)`.

To preserve the identity and composition we require

```
T(I,I,id(C),rho,u) = u   and
T(I,K, f o g, rho, u) = T(J,K,g,f(rho), T(I,J,f,rho,u))
```

Putting all of this together we get the definition of the Nuprl type {`H` ⊢ _}, the type of types in context `H`:

```
{H ⊢ _}  ==
{TF:T:I:fset(ℕ) → H(I) → 𝕌{i} ×
   I:ob(C)→ J:ob(C)→ f:(J→ I)→ a:H(I)→ T(I,a) → T(J,f(a))
   | let  T,F = TF  in
   ∀I:ob(C). ∀a:H(I). ∀u:T(I,a). F(I,I,id,a.u) = u ∧
   ∀I,J,K:ob(C). ∀f:J→ I.  ∀g:K → J. ∀a:H(I). ∀u:T(I,a).
    F(I,K, f · g, a, u) = F(J, K, g, f(a), F(I,J,f,a ,u))}
```

The display form {H ⊢ _} is meant to indicate that this is the (Nuprl) type of things that can follow `H ⊢`. Then we display `T ∈ {H ⊢ _}` as `H ⊢ T`.

Universe levels for Contexts and Types In Nuprl, almost every definition is provided with a typing lemma that we call its *wellformedness lemma*. When for an expression `t` and a type `T` it is true the `t ∈ T`, the `Auto` tactic can usually use the wellformedness lemmas to prove this (even though typing is undecidable in general). Note that because of subtyping and type extensionality the type of an expression `t` is not unique, so there may be many types `T` for which `t ∈ T` is provable and `Auto` may only prove some of them, while proving others may take more steps. The wellformedness lemma for {H ⊢ _} is

```
∀[H:⊢ ]. {H ⊢ _} ∈ 𝕌{i+1}
```

Although we display only ⊢ for the Nuprl type of all contexts, its definition as the presheaves over `C` means that it has a universe level parameter `i`. The typing lemma says that for any context `H` (with level parameter `i`), the type of types in context `H` is a Nuprl type in universe `i+1`.

Since definition of {H ⊢ _} mentions 𝕌{i} the smallest universe it can be a member of is 𝕌{i+1}, and as long as all the other types in the definition are in 𝕌{i+1}, the whole type will be in 𝕌{i+1}. So we can allow the "sets" `H(I)` to be in 𝕌{i+1}. This means that we can define ⊢ to be `Presheaf(C)`{i+1} and still have {H ⊢ _} ∈ 𝕌{i+1}. This subtlety turns out to be important when we come to modeling the rules for universes in cubical type theory.

Definitions of `(H.T)`, `p`, `(T)sigma`, and empty context: If `H` is a context and `H ⊢ T` then `H.T` is also a context. It is a presheaf that assigns to `I` the set of pairs `<rho,u>` where `rho ∈ H(I)` and `u ∈ T(I,rho)`. The `H`-restriction map `f(rho)` and the "morphism map" `T(I,J,f,rho,u)` give the `H.T`-restriction map.

```
H.T ==
```

```
Presheaf(Set(I) = rho:H(I) × T(I,rho)
        Morphism(I,J,f,pr) = let rho,u = pr in
                             <f(rho), T(I,J,f,rho,u)>          )
```

A natural transformation from `H.T` to `H` must assign to each `I` a map from `rho:H(I) × T(I,rho)` to `H(I)`. One obvious choice is the polymorphic map:

```
p ==  I |→ λ pr.fst(pr)
```

If `H` and `G` are contexts, `G ⊢ T`, and `sigma:H → G` then to get a type `(T)sigma` in context `H` we need to define the type `(T)sigma(I,rho)` and the morphism map `(T)sigma(I,J,f,rho,u)`. Since `sigma` is a natural transformation, `sigma(I,rho) ∈ G(I)` when `rho ∈ H(I)`, so the definition is:

```
(T)sigma ==
<λI,rho. T(I, sigma(I,rho)),
 λI,J,f,rho,u. T(I,J,f,sigma(I,rho),u)>
```

For the empty context we take the presheaf

```
() == Presheaf(Set(I) = Unit; Morphism(I,J,f,u) = u)
```

Terms of type `T` in context `H`: When `H ⊢ T`, we have "sets" (i.e. types) `T(I,rho)` for each `rho ∈ H(I)`. A *term* `t:T` will assign to each `I` and `rho` a member of `T(I,rho)` in a way that respects the morphism maps. So the Nuprl type for the terms `{H ⊢ _:T}`, i.e. the `t` for which `{H ⊢ t:T}`, is:

```
{H ⊢ _:T}  ==
  {t:I:ob(C) → rho:H(I) → T(I,rho) |
    ∀I,J:ob(C). ∀f:J → I. ∀rho:H(I).
      T(I,J,f,rho,t(I,rho)) = t(J,f(rho))}
```

So a term in the presheaf model is a family of Nuprl terms.

Given a natural transformation `sigma:H → G` and a term `t` such that `G ⊢ t:T`, we get a term `(t)sigma` for which `H ⊢ (t)sigma:(T)sigma`

```
(t)sigma == λI,rho. t(I,sigma(I,rho))
```

Since for the context `H.T` the set `(H.T)(I)` is `rho:H(I) × T(I,rho)`, we see that the function

```
q == λI,pr. snd(pr)
```

is a term of type `H.T ⊢ q:(T)p`.

Substitutions: We said that the *context maps* `sigma:H⟶G` in the name-free syntax play the role that substitutions do in a theory with bound variables. We need one more generic definition that makes this analogy more precise. Consider an elimination rule that says in `H, x:A ⊢ C` we can eliminate `x` by substituting a term `u` where `H ⊢ u:A` for `x` to get `H ⊢ C[x/u]`. How do we express this without the bound variable `x`?

We need a context map `[u]:H⟶ H.A`, for then if `H.A ⊢ C` we get `H ⊢ C[u]`. We first make a more general definition:

```
(sigma;u) ==   I |⟶ λ rho.<sigma(I,rho), u(I,rho)>
```

If `sigma:H⟶G`, and `G ⊢ A`, and `H ⊢ u:(A)sigma` then for `rho ∈H(I)`, we have `sigma(I,rho) ∈ G(I)` and `u(I,rho) ∈ ((A)sigma)(I,rho)`. Since `((A)sigma)(I,rho) = A(I,sigma(I,rho))`, `(sigma;u) ∈ H⟶ G.A`.

With the identity map `1` for `sigma` we get

```
[u] == (1,u)
```

and `[u]` has type `H⟶ H.A`.

Basic structural rules for MLTT: We have given the formal Nuprl definitions for $(H \vdash)$, $\sigma : H \to G$, $(H \vdash T)$, $(H \vdash t : T)$, $(H.T)$, $(T)\sigma$, $(t)\sigma$, p, q, and $[u]$. They satisfy the following twenty basic rules.

1. `G ⊢ ⇒ 1 ∈ G⟶G`
2. `sigma ∈ H⟶G ∧ delta ∈ K⟶H ⇒ sigma delta ∈ K⟶G`
3. `G ⊢ A ∧ sigma ∈ H⟶G ⇒ H ⊢ (A)sigma`
4. `G ⊢ t:A ∧ sigma ∈ H⟶G ⇒ H ⊢ (t)sigma:(A)sigma`
5. `() ⊢`
6. `G ⊢ ∧ G ⊢ A ⇒ G.A ⊢`
7. `G ⊢ A ⇒ p:  G.A ⟶ G`
8. `G ⊢ A ⇒ G.A ⊢ q:(A)p`
9. `sigma ∈ H⟶G ∧ G ⊢ A ∧ H ⊢ u:(A)sigma ⇒ (sigma,u):  H⟶G.A`
10. `1 sigma = sigma 1 = sigma`
11. `(sigma delta) nu = sigma (delta nu)`

12. `[u] = (1,u)`
13. `(A)1 = A`
14. `((A)sigma)delta = (A)(sigma delta)`
15. `(u)1 = 1`
16. `((u)sigma)delta = (u)(sigma delta)`
17. `(sigma,u) delta = (sigma delta, (u)delta)`
18. `p (sigma,u) = sigma`
19. `(q)(sigma,u) = u`
20. `(p,q) = 1`

Notice that all of these rules are either typing rules or equations. Thus, at least in Nuprl, there is no computational content to be extracted from the proofs of these rules. The only computational content, so far, is in the given definitions. Because of this, there is not much reason to discuss the formal proofs of these rules, except perhaps to note which of them depend on Nuprl's extensional features, and just note that they have all been proved formally and are in the Nuprl library.

8 Σ and Π types

Next we define the types $\Pi(A, B)$ and $\Sigma(A, B)$ and associated terms. For a context `H`, if `H` $\vdash$ `A` and `H.A` $\vdash$ `B`, then we should have both `H` $\vdash$ **Σ**`(A,B)` and `H` $\vdash$ **Π**`(A,B)`. We also need a pairing term `(u,v)` to form a term `w` $\in$ **Σ**`(A,B)`, and terms `w.1` and `w.2` to decompose it. We need a **λ** to make `f` $\in$ **Π**`(A,B)` and `app(f,u)` to apply it.

Recall that to define a type `T` in context `H` we need a family of types `T(I,rho)` where `rho` $\in$ `H(I)` and for `u` $\in$ `T(I,rho)` and `f:J`$\rightarrow$`I` we need the morphism maps `T(I,J,f,rho,u)` $\in$ `T(J,f(rho))`.

The Σ Type: The sigma type is relatively straightforward. Given `rho` $\in$ `H(I)` we have the type `A(I,rho)` and for `u` $\in$ `A(I,rho)` the pair `<rho,u>`is a member of `(H.A)(I)` so `B(I,<rho,u>)` is a type. Thus the family **Σ**`(A,B)` is defined by

Σ`(A,B)(I,rho) ==` `u:A(I,rho)` $\times$ `B(I,<rho;u>)`

Given a pair `p` $\in$ $\mathbf{\Sigma}$`(A,B)(I,rho)` and arrow `f:J`$\rightarrow$`I`, the first component of `p` is a member of `A(I,rho)` so `A(I,J,f,rho,fst(p))` $\in$`A(J,f(rho))`. Then `B(I,J,f,<rho,fst(p)>,snd(p))` is a member of `B(J,f(rho,fst(p)))`. So we can define the morphism map:

```
Σ(A,B)(I,J,f,rho,p) ==
  <A(I,J,f,rho,fst(p)),B(I,J,f,(rho,fst(p)),snd(p))>
```

So the formal definition of the type $\mathbf{\Sigma}$`(A,B)` is:

```
Σ(A,B) ==
 <λ I,rho. u:A(I,rho) × B(I,<rho;u>),
  λ I,J,f,rho,p. <A(I,J,f,rho,fst(p)),
                  B(I,J,f,<rho,fst(p)>,snd(p))>
 >
```

Now if `H` $\vdash$ `A` and `H.A` $\vdash$ `B` we can prove `H` $\vdash$ $\mathbf{\Sigma}$`(A,B)`. Note that the definition of $\mathbf{\Sigma}$`(A,B)` does not mention the context `H`, so it is polymorphic. That will not be the case for the $\mathbf{\Pi}$-type.

Recall that a term `H` $\vdash$ `t:T` is a family of terms `t(I,rho)` $\in$ `T(I,rho)` that respects the morphisms.

If `H` $\vdash$ `u:A` and `H` $\vdash$ `v:B[u]` then the *pair term* `(u,v)` is simply

```
(u,v) == λI,rho. <u(I,rho),v(I,rho)>
```

and for `H` $\vdash$ `pr:`$\mathbf{\Sigma}$`(A,B)` the terms `pr.1` and `pr.2` are defined simply by:

```
pr.1 == λI,rho. fst(pr(I,rho))
pr.2 == λI,rho. snd(pr(I,rho))
```

The $\mathbf{\Pi}$ Type: The definition of the $\mathbf{\Pi}$-type is trickier. In order to get a family of function types that will transform properly given an arrow `f:J`$\rightarrow$`I`, we build that requirement into the definition. The family of types $\mathbf{\Pi}$`(H,A,B,I,rho)`—now depending on the context `H`— is a subtype of functions of type `J:ob(C)`$\rightarrow$`f:(J`$\rightarrow$`I)`$\rightarrow$`u:A(J,f(rho))`$\rightarrow$`B(J,<f(rho),u>)` . In order to transform properly, if `w` is such a function, then it must satisfy the condition that for all `f:J`$\rightarrow$`I` and for all `g:K`$\rightarrow$`J`, if `u`$\in$`A(J,f(rho))` then

```
B(J,K,g,<f(rho),u>,w(J,f,u)) = w(K,(f · g), A(J,K,g,f(rho),u))
```

This says applying `w` and then transforming in type `B` gives the same result as first transforming in type `A` and then applying `w`. Recall that restriction map `f(rho)` really has parameters `H,I,J,f,rho` so this family of functions does depend on the context `H`. The formal definition is:

```
pi-family(H;A;B;I;rho)  ==
{w:J:ob(C)⟶ f:(J⟶I) ⟶ u:A(J,f(rho)) ⟶ B(J,(f(rho);u)) |
 ∀J,K:ob(C).∀f:J⟶I.∀g:K ⟶J.∀u:A(J,f(rho)).
 B(J,K,g,<f(rho),u>,w(J,f,u)) = w(K,(f · g), A(J,K,g,f(rho),u))
}
```

The morphism map for this family is now easy to define. Given a `w` in `pi-family(H;A;B;I;rho)` and `f:J⟶I` then the function

```
  λ K,g,v. w(K, (f · g),v)
```

will be a member of `pi-family(H;A;B;J;f(rho))`. So the formal definition of the type **Π**`(A,B)` is:

```
Π(A,B)  ==
   <λI,rho. pi-family(H;A;B;I;rho),
    λI,J,f,rho,w,K,g. (w K (f · g))>
```

Now we need a **λ**-term to build members of **Π**`(A,B)`. If `H ⊢ A` and we have a term `b` of type `H.A ⊢ b:B` (where `H.A ⊢ B`) then we want to define the term **λ**`b` so that `H ⊢ λb:`**Π**`(A,B)`. So, given `rho ∈ H(I)` we need a member of `pi-family(H;A;B;I;rho)` and that is a function that takes as input `J`, `f:J⟶ I`, and `u ∈ A(J,f(rho))`. We get such a function by applying term `b` to `J` and the pair `<f(rho),u>`. Thus, the definition is:

```
(λb) ==  λI,a,J,f,u. b(J,<f(a),u>)
```

To prove the typing rule for `(λb)` we have to show that it is in the subtype of functions that satisfy the given constraints. This follows from the constraints on the types `A`, `B`, and the term `b`.

Finally, if we have a term `w` of type `H ⊢ `**Π**`(A,B)` and a term `u` of type `H ⊢ A`, then we want a term `app(w,u)` of type `H ⊢B[u]`. So, given `rho ∈ H(I)`, we have `u(I,rho) ∈ A(I,rho)` and `w(I,rho)` is a member of `pi-family(H;A;B;I;rho)` so we can apply `w(I,rho)` to any `J`, `f:J⟶I` and `v ∈A(J,f(rho))`. The only sensible option is to take `J = I`, `f = id(I)`, and `v = u(I,rho)`. The definition is therefore:

```
app(w,u)  ==  λI,rho. w(I,rho)(I,1,u(I,rho))
```

Rules for MLTT basic type formers: We have given the formal Nuprl definitions for **Π**`(A,B)`, **Σ**`(A,B)`, `app(w,u)`, `(λb)`, `(u,v)`, `pr.1`, and `pr.2`.

They satisfy the following nineteen basic rules.

1. G.A ⊢ B ⇒ G ⊢ **Π**(A,B)
2. G.A ⊢ B ∧ G.A ⊢ b:B ⇒ G ⊢ (**λ**b):**Π**(A,B)
3. G.A ⊢ B ⇒ G ⊢ **Σ**(A,B)
4. G.A ⊢ B ∧ G ⊢ u:A ∧ G ⊢ v:B[u] ⇒ G ⊢ (u,v):**Σ**(A,B)
5. G ⊢ pr:**Σ**(A,B) ⇒ G ⊢ (pr.1):A
6. G ⊢ pr:**Σ**(A,B) ⇒ G ⊢ (pr.2):B[pr.1]
7. G ⊢ f:**Π**(A,B) ∧ G ⊢ u:A ⇒ G ⊢ app(f,u):B[u]
8. **Π**(A,B) sigma = **Π**(A sigma, B (sigma p, q))
9. (**λ**b)sigma = **λ**(b((sigma p, q)))
10. app(f,u)sigma = app(f sigma, u sigma)
11. app(**λ**b, u) = b[u]
12. f = **λ**(app((f)p,q)
13. **Σ**(A,B) sigma = **Σ**(A sigma, B (sigma p, q))
14. (pr.1)sigma = (pr sigma).1
15. (pr.2)sigma = (pr sigma).2
16. (u,v)sigma = (u sigma, v sigma)
17. (u,v).1 = u
18. (u,v).2 = v
19. (pr.1,pr.2) = pr

Again, all of these rules are either typing rules or equations so no computational content comes from the proofs of these rules, and the only computational content is in the given definitions. Again, we need not discuss the proofs, all of which have been done and are in the Nuprl library. These proofs were first done for a particular category, the *cube category*, but we later generalized these definitions and proofs to have an arbitrary category C as a parameter, and those are the definitions given in this article.

The next article in the series will define the cube category and the interval type $\mathbb{I}$.