Category Theory In Isabelle/HOL

Alex Katovsky

June 7, 2010

Abstract

We describe a development of Category Theory in Isabelle. A Category is defined using records and locales in Isabelle/HOL. Functors and Natural Transformations are also defined. The main result that has been formalized is that the Yoneda functor is a full and faithful embedding. We also formalize the completeness of many sorted monadic equational logic. Extensive use is made of the HOLZF theory in both cases.

1 Introduction

The development here[1] is inspired in large measure, at least initially, by a previous effort to formalize category theory in Isabelle [O'K09]. This work can be found in the Archive of Formal Proofs[2]. In this previous work, just enough is done to formalize a version of Yoneda's Lemma for the HOL type *'a set*. In the present development, which is independent of the former, Yoneda's lemma is proved for the type *ZF*, which supports ZFC set theory in HOL. The development much more closely mimics the standard mathematical presentation, in which the Yoneda embedding is defined and it is shown to be full and faithful.

2 Categories

Category theory is a general theory of composable directed arrows. It is a useful framework because many interesting phenomena have the properties of directedness and composability; the arrows could represent mathematical functions, procedures, or steps in the spatiotemporal evolution of some system.

Category theory can be developed within a model of set theory satisfying Grothendieck's axiom (called a Grothendieck universe). This is done in [KS06] for example and some of the code in `Universe.thy` survives from an aborted attempt to imitate this. This might be a

[1] see http://www.srcf.ucam.org/∼apk32/Isabelle/Category/ for the source code (the given code has been tested with `Isabelle2009-1: December 2009`)

[2] http://afp.sourceforge.net/

sensible approach if we were using Isabelle ZF but, as we are using HOL, it makes more sense to follow [O'K09] and define a category as a polymorphically typed record containing all the attributes of "composable directed arrows". The attributes are a collection of objects, a collection of arrows (also called *morphisms*), a rule for composition, and, for each arrow, a domain and a codomain, and for each object an identity arrow. We take here the *internalist* view of category theory and define all the rules as partial maps. So, for example, composition is a partial map

$$\mathrm{Mor}\,(\mathbb{C}) \times \mathrm{Mor}\,(\mathbb{C}) \to \mathrm{Mor}\,(\mathbb{C})$$

which is defined on the collection

$$\{(f, g) \in \mathrm{Mor}\,(\mathbb{C}) \times \mathrm{Mor}\,(\mathbb{C})\,; \mathrm{cod}\,(f) = \mathrm{dom}\,(g)\}$$

By contrast, the *externalist* view avoids partial maps and instead defines composition as a family of maps indexed by objects. The latter approach might be more natural in a dependently typed system, such as Coq.

So, to be precise, here is the definition in Isabelle:

```
record ('o,'m) Category =
  Obj :: "'o set" ("objı" 70)
  Mor :: "'m set" ("morı" 70)
  Dom :: "'m ⇒ 'o" ("domı _" [80] 70)
  Cod :: "'m ⇒ 'o" ("codı _" [80] 70)
  Id  :: "'o ⇒ 'm" ("idı _" [80] 75)
  Comp :: "'m ⇒ 'm ⇒ 'm" (infixl ";;ı" 70)
```

and we make some auxiliary definitions before we state the axioms of a category:

```
definition
"MapsTo C f X Y ≡
       f ∈ Mor C ∧ Dom C f = X ∧ Cod C f = Y"
```

```
definition
  CompDefined "CompDefined C f g ≡
       f ∈ Mor C ∧ g ∈ Mor C ∧ Cod C f = Dom C g"
```

The first is abbreviated $f\ \mathit{maps}_C\ X\ \mathit{to}\ Y$ and the second as $f \approx>_C g$. As we are taking the internal view of a category, we need to include axioms for the extensionality of the partial maps. Otherwise there will be multiple versions of the same category (just by defining the partial maps outside their valid domains in different ways), which would be a problem when defining a category of categories (the "conglomerate of all categories" in [AHS04]). We define these axioms in a separate locale:

```
locale ExtCategory =
```

```
  fixes C :: "('o,'m,'a) Category_scheme" (structure)
  assumes CdomExt: "(Dom C) ∈ extensional (Mor C)"
  and     CcodExt: "(Cod C) ∈ extensional (Mor C)"
  and     CidExt:  "(Id C) ∈ extensional (Obj C)"
  and     CcompExt:
    "(split (Comp C)) ∈ extensional ({(f,g) | f g . f ≈> g})"
```

Then the axioms for a category are

```
locale Category = ExtCategory +
  assumes Cdom : "f ∈ mor ⟹ dom f ∈ obj"
  and     Ccod : "f ∈ mor ⟹ cod f ∈ obj"
  and     Cidm [dest]: "X ∈ obj ⟹ (id X) maps X to X"
  and     Cidl : "f ∈ mor ⟹ id (dom f) ;; f = f"
  and     Cidr : "f ∈ mor ⟹ f ;; id (cod f) = f"
  and     Cassoc :
  "⟦f ≈> g ; g ≈> h⟧ ⟹ (f ;; g) ;; h = f ;; (g ;; h)"
  and     Ccompt :
  "⟦f maps X to Y ; g maps Y to Z⟧ ⟹ (f ;; g) maps X to Z"
```

And we define `MakeCat` so that we can forget about the extensionality in defining a category:

```
definition
MakeCat :: "('o,'m,'a) Category_scheme ⇒ ('o,'m,'a) Category_scheme"
where
  "MakeCat C ≡ ⦇
  Obj = Obj C ,
  Mor = Mor C ,
  Dom = restrict (Dom C) (Mor C) ,
  Cod = restrict (Cod C) (Mor C) ,
  Id  = restrict (Id C) (Obj C) ,
  Comp = λ f g . (restrict (split (Comp C))
                 ({(f,g) . f ≈>_C g})) (f,g),
  ... = Category.more C
  ⦈"
```

```
lemma MakeCat: "Category_axioms C ⟹ Category (MakeCat C)"
```

So the way that categories, say `CC`, will often be defined is to define a `Category` record `CC'` (with total functions), prove that `Category_axioms C'` and then use the above lemma `MakeCat` to show that `MakeCat C'` is a category. There are many examples of this in what follows. And the same technique is used for defining functors and natural transformations.

We prove some simple theorems, for example

```
lemma (in Category) CompDefComp:
  assumes "f ≈> g" and "g ≈> h"
```

```
  shows "f ≈> (g ;; h)" and "(f ;; g) ≈> h"

lemma (in Category) IdInj:
  assumes "X ∈ obj" and "Y ∈ obj" and "id X = id Y"
  shows   "X = Y"
```

Then we show that an inverse is unique:

```
lemma (in Category) LeftRightInvUniq:
  assumes 0: "h ≈> f" and  z: "f ≈> g"
  assumes 1: "f ;; g = id (dom f)"
  and     2: "h ;; f = id (cod f)"
  shows   "h = g"
proof-
  have mor: "h ∈ mor ∧ g ∈ mor"
  and  dc : "dom f = cod h ∧ cod f = dom g" using 0 z by auto
  then have "h = h ;; id (dom f)" by (auto simp add: Simps)
  also have "... = h ;; (f ;; g)" using 1 by simp+
  also have "... = (h ;; f) ;; g" using 0 z by (simp add: Cassoc)
  also have "... = (id (cod f)) ;; g" using 2 by simp+
  also have "... = g" using mor dc by (auto simp add: Simps)
  finally show ?thesis .
qed
```

and we develop a small theory of isomorphisms, which culminates in this theorem (see `Cat.thy` for the details):

```
lemma (in Category) IsoCompose:
  assumes 1: "f ≈> g" and 2: "ciso f" and 3: "ciso g"
  shows "ciso (f ;; g)"
    and "Cinv (f ;; g) = (Cinv g) ;; (Cinv f)"
```

We then define our first category; the unit category and prove that it is a category:

```
definition
  UnitCategory :: "(unit, unit) Category" where
  "UnitCategory = MakeCat (|
      Obj = {()} ,
      Mor = {()} ,
      Dom = (λf.()) ,
      Cod = (λf.()) ,
      Id = (λf.()) ,
      Comp = (λf g. ())
  |)"

lemma [simp]: "Category(UnitCategory)"
apply (simp add: UnitCategory_def, rule MakeCat)
by (unfold_locales, auto simp add: UnitCategory_def)
```

The next is the *Opposite Category* of a category:

```
definition
  OppositeCategory :: "('o,'m,'a) Category_scheme ⇒
     ('o,'m,'a) Category_scheme" ("Op _" [65] 65) where
  "OppositeCategory C ≡ (|
      Obj = Obj C ,
      Mor = Mor C ,
      Dom = Cod C ,
      Cod = Dom C ,
      Id  = Id C ,
      Comp = (λf g. g ;;C f),
      ... = Category.more C
  |)"
```

which takes a given category and reverses the direction of the arrows, so domain and codomain swap places and the order of composition is switched. At the end of `Cat.thy` we show that if *Category C* then *Category (Op C)*. The opposite category is used in the definition of the contravariant hom functor in `SetCat.thy` and in the definition of the Yoneda embedding in `Yoneda.thy`.

2.1 Functors

A *functor* is a directed arrow from one category to another, which is structure preserving. It consists of a domain category, a codomain category, and a map taking morphisms in one category to morphisms in another.

```
record ('o1, 'o2, 'm1, 'm2, 'a, 'b) Functor =
  CatDom :: "('o1,'m1,'a)Category_scheme"
  CatCod :: "('o2,'m2,'b)Category_scheme"
  MapM :: "'m1 ⇒ 'm2"
```

We abbreviate the functor *F* applied to the morphism *f* by *F ## f*. In many expositions of category theory, the functor is a pair of maps; the morphism part and the object part. But the object part (the one mapping objects in the domain category to objects in the codomain category) can be deduced from the morphism part by the action on identities in the following way:

```
definition
  "MapO F X ≡ THE Y . Y ∈ Obj(CatCod F) ∧
            F ## (Id (CatDom F) X) = Id (CatCod F) Y"
```

and we abbreviate the functor *F* applied to the object *X* by *F @@ X*. The axioms are split into a number of locales

```
locale PreFunctor =
  fixes F :: "('o1, 'o2, 'm1, 'm2, 'a1, 'a2, 'a) Functor_scheme"
(structure)
  assumes FunctorComp: "f ≈>CatDom F g ⟹
    F ## (f ;;CatDom F g) = (F ## f) ;;CatCod F (F ## g)"
  and     FunctorId:   "X ∈ objCatDom F ⟹
    ∃ Y ∈ objCatCod F . F ## (idCatDom F X) = idCatCod F Y"
  and     CatDom[simp]:      "Category(CatDom F)"
  and     CatCod[simp]:      "Category(CatCod F)"

locale FunctorM = PreFunctor +
  assumes FunctorCompM: "f mapsCatDom F X to Y ⟹
        (F ## f) mapsCatCod F (F @@ X) to (F @@ Y)"

locale FunctorExt =
  fixes F :: "('o1, 'o2, 'm1, 'm2, 'a1, 'a2, 'a) Functor_scheme"
(structure)
  assumes FunctorMapExt:
    "(MapM F) ∈ extensional (Mor (CatDom F))"

locale Functor = FunctorM + FunctorExt
```

The strategy to defining a functor *F* is to prove first that *PreFunctor F*. We can then deduce the action on objects using

```
lemma (in PreFunctor) FmToFo:
    "⟦X ∈ objCatDom F ; Y ∈ objCatCod F ;
      F ## (idCatDom F X) = idCatCod F Y⟧ ⟹ F @@ X = Y"
```

then we prove *FunctorM F*. As with categories, we have a function *MakeFtor* so that we can forget about extensionality. We abbreviate that *F* is a functor with domain *A* and codomain *B* by

$$\mathit{Ftor}\ F\ :\ A\ \longrightarrow\ B$$

The first functor we define is the identity functor, which takes a category to itself, and the action on morphisms is the identity:

```
definition
  IdentityFunctor' ("FId' _" [70]) where
  "IdentityFunctor' C ≡ (|CatDom = C , CatCod = C ,
              MapM = (λ f . f) |)"

definition
  IdentityFunctor ("FId _" [70]) where
  "IdentityFunctor C ≡ MakeFtor(IdentityFunctor' C)"
```

We show that it is a *PreFunctor*:

```
lemma IdFtor'PreFunctor:
```

```
    "Category C ⟹ PreFunctor (FId' C)"
by(auto simp add: PreFunctor_def IdentityFunctor'_def)
```

then we show that the action on objects is identity using *FmToFo*:

```
lemma IdFtor'Obj:
  assumes "Category C" and "X ∈ obj_CatDom (FId' C)"
  shows "(FId' C) @@ X = X"
proof-
  have "(FId' C) ## id_CatDom (FId' C) X = id_CatCod (FId' C) X"
    by(simp add: IdentityFunctor'_def)
  moreover have "X ∈ obj_CatCod (FId' C)" using assms
    by (simp add: IdentityFunctor'_def)
  ultimately show ?thesis using assms
    by (simp add: PreFunctor.FmToFo IdFtor'PreFunctor)
qed
```

another short lemma (*IdFtor'FtorM*) shows that *FunctorM(FId' C)* and then we use *MakeFtor* to complete the proof:

```
lemma IdFtorFtor:  "Category C ⟹ Functor (FId C)"
by (auto simp add: IdentityFunctor_def IdFtor'FtorM
         intro: MakeFtor)
```

And we go through the same procedure with the *Constant Functor*, which maps every morphism in the domain category to the identity of some object in the codomain category:

```
definition
  "ConstFunctor' A B b ≡ (|
         CatDom = A ,
         CatCod = B ,
         MapM = (λ f . (Id B) b)
  |)"

definition "ConstFunctor A B b ≡
                MakeFtor(ConstFunctor' A B b)"
```

As a special case, we get the unit functor, which maps everything in the domain category to the unit morphism in the unit category:

```
definition
  "UnitFunctor C ≡ ConstFunctor C UnitCategory ()"
```

We said that functors are directed arrows between categories and that category theory is the theory of composable directed arrows, so now we define the composition of two functors:

```
definition
```

```
  FunctorComp' (infixl ";;:" 71) where
  "FunctorComp' F G ≡ (|
        CatDom = CatDom F ,
        CatCod = CatCod G ,
        MapM   = λ f . (MapM G)((MapM F) f)
  |)"
```

```
definition FunctorComp (infixl ";;;" 71) where
    "FunctorComp F G ≡ MakeFtor (FunctorComp' F G)"
```

It takes a little more work to prove that this is a functor, but the procedure is the same as before.

2.2 Natural Transformations

Natural Transformations are directed arrows between functors. They lead to the important concept of *Natural Isomorphism*, which makes precise the idea of two structures in mathematics being naturally isomorphic, as opposed to merely isomorphic. The standard example is that a vector space is naturally isomorphic to its second dual. In the definition of a natural transformation, the domains and codomains of the domain and codomain functor must agree. In this case, the common category domain and codomain is the category of vector spaces over a fixed field. The domain functor is the identity functor and the codomain functor is the second dual functor, whose object component maps a vector space to its second dual. Unfortunately, I did not have the time to formalize this example in Isabelle, although it should be possible to combine a vector spaces theory with this category theory to show the result.

To be precise, a natural transformation is a family of morphisms in the common codomain category indexed by objects from the common domain category:

```
record ('o1, 'o2, 'm1, 'm2, 'a, 'b) NatTrans =
  NTDom :: "('o1, 'o2, 'm1, 'm2, 'a, 'b) Functor"
  NTCod :: "('o1, 'o2, 'm1, 'm2, 'a, 'b) Functor"
  NatTransMap :: "'o1 ⇒ 'm2"
```

The common domain and codomain categories are given definitions:

```
definition   "NTCatDom η ≡ CatDom (NTDom η)"
definition   "NTCatCod η ≡ CatCod (NTCod η)"
```

and we abbreviate the action on objects by `η $$ X` (which is a morphism in `NTCatCod η` and `X` is an object in `NTCatDom η`) then we define the axioms; the category domains and codomains must match up, the image of the object under the mapping must be correct, and satisfy a kind of commutativity rule (we also must take care of extensionality, which will be crucial when we come to define the functor category):

```
locale NatTransExt =
  fixes η :: "('o1, 'o2, 'm1, 'm2, 'a, 'b) NatTrans" (structure)
  assumes  NTExt :
    "NatTransMap η ∈ extensional (Obj (NTCatDom η))"
```

```
locale NatTransP =
  fixes η :: "('o1, 'o2, 'm1, 'm2, 'a, 'b) NatTrans" (structure)
  assumes NatTransFtor:   "Functor (NTDom η)"
  and     NatTransFtor2:  "Functor (NTCod η)"
  and     NatTransFtorDom:   "NTCatDom η = CatDom (NTCod η)"
  and     NatTransFtorCod:   "NTCatCod η = CatCod (NTDom η)"
  and    NatTransMapsTo:  "X ∈ obj_{NTCatDom η} ⟹
(η $$ X) maps_{NTCatCod η} ((NTDom η) @@ X) to ((NTCod η) @@ X)"
  and    NatTrans:  "f maps_{NTCatDom η} X to Y ⟹
((NTDom η) ## f) ;;_{NTCatCod η} (η $$ Y) =
                (η $$ X) ;;_{NTCatCod η} ((NTCod η) ## f)"
```

```
locale NatTrans = NatTransP + NatTransExt
```

And we have *MakeNT*, which is analogous to *MakeCat* and *MakeFtor*. We abbreviate that η is a natural transformation with domain functor *F* and codomain functor *G* by

$$NT\ \eta\ :\ F \Longrightarrow G$$

As with functors, we define the identity natural transformation on a functor:

```
lemma IdNatTransNatTrans:
    "Functor F ⟹ NatTrans (IdNatTrans F)"
```

and also the composition of two natural transformations:

```
lemma NatTransCompNatTrans:
    "η1 ≈>· η2 ⟹ NatTrans (η1 · η2)"
```

This is largely in preparation for the definition of the *functor category*, whose objects are functors between two given categories and whose morphisms are natural transformations between the functors. This is a very important category, partly because the *presheaf category* is an instance of it:

```
definition
  "CatExp' A B ≡ (|
      Cat.Category.Obj = {F . Ftor F : A ⟶ B} ,
      Cat.Category.Mor =
{η . NatTrans η ∧ NTCatDom η = A ∧ NTCatCod η = B} ,
      Cat.Category.Dom = NTDom ,
      Cat.Category.Cod = NTCod ,
      Cat.Category.Id  = IdNatTrans ,
```

```
    Cat.Category.Comp = λf g. (f · g)
  ⦈"
```

definition *"CatExp A B ≡ MakeCat(CatExp' A B)"*

The rest of `NatTrans.thy` is largely devoted to showing that this is a category.

2.3 The Category of Sets

In contrast to [O'K09], we define the category of sets using a model of ZFC, the theory HOLZF by Steven Obua, which is in the Isabelle distribution. We import the theory via `Universe.thy`. The HOL type *ZF* axiomatizes ZFC. The function *explode* takes an object of type *ZF* and returns an object of type *ZF set* which is the class of member elements:

explode z == { x. Elem x z }

For convenience, we abbreviate *Elem x z* by *x |∈| z*, and other abbreviations that place bars around their corresponding symbols for the HOL type *set*.

We first define a set function:

```
definition
  ZFfun :: "ZF ⇒ ZF ⇒ (ZF ⇒ ZF) ⇒ ZF" where
  "ZFfun d r f ≡ Opair (Opair d r) (Lambda d f)"
```

```
definition
  ZFfunDom :: "ZF ⇒ ZF" ("|dom|_" [72] 72) where
  "ZFfunDom f ≡ Fst (Fst f)"
```

```
definition
  ZFfunCod :: "ZF ⇒ ZF" ("|cod|_" [72] 72) where
  "ZFfunCod f ≡ Snd (Fst f)"
```

and we abbreviate function application by *f |@| x* and the composition of two functions (note the order in the definition) by *f |o| g* and we define *isZFfun*, which takes *ZF* to *bool* and returns true iff the argument is in the image of *ZFfun*. Now we can define the category of sets:

```
definition
  SET' :: "(ZF, ZF) Cat.Category" where
  "SET' ≡ ⦇
    Cat.Category.Obj = {x . True} ,
    Cat.Category.Mor = {f . isZFfun f} ,
    Cat.Category.Dom = ZFfunDom ,
    Cat.Category.Cod = ZFfunCod ,
    Cat.Category.Id = λx. ZFfun x x (λx . x) ,
    Cat.Category.Comp = ZFfunComp
```

```
|)"
```

```
definition "SET ≡ MakeCat SET'"
```

Once we have proved a few facts about set functions, then we prove that this is in fact a category. Next, we define a *locally small* category, which is one in which the hom classes are sets (i.e. in the range of the `explode` function). As we are allowing type polymorphism, we need an injective function taking morphisms to `ZF` (`implode` is the formal inverse of `explode`):

```
record ('o,'m) LSCategory = "('o,'m) Category" +
  mor2ZF :: "'m ⇒ ZF" ("m2zı_" [70] 70)

definition
  ZF2mor ("z2mı_" [70] 70) where
  "ZF2mor C f ≡ THE m . m ∈ mor_C ∧ m2z_C m = f"

definition
  "HOMCollection C X Y ≡ {m2z_C f | f . f maps_C X to Y}"

definition
  HomSet ("Homı _ _" [65, 65] 65)  where
  "HomSet C X Y ≡ implode (HOMCollection C X Y)"

locale LSCategory = Category +
  assumes mor2ZFInj: "[|x ∈ mor ; y ∈ mor ; m2z x = m2z y|]
     ⟹ x = y"
  and HOMSetIsSet: "[|X ∈ obj ; Y ∈ obj|]
     ⟹ HOMCollection C X Y ∈ range explode"
  and m2zExt: "mor2ZF C ∈ extensional (Mor C)"
```

then we define the hom functor:

```
definition HomFtorMap  ("Homı[_,_]" [65,65] 65)  where
  "HomFtorMap C X g ≡ ZFfun (Hom_C X (dom_C g))
                    (Hom_C X (cod_C g))
                    (λ f . m2z_C ((z2m_C f) ;;_C g))"

definition
  HomFtor' ("HomPı[_,-]" [65] 65) where
  "HomFtor' C X ≡ (|
        CatDom = C,
        CatCod = SET ,
        MapM   = λ g . Hom_C[X,g]
  |)"

definition HomFtor ("Homı[_,-]" [65] 65)  where
    "HomFtor C X ≡ MakeFtor (HomFtor' C X)"
```

Before we prove that this is indeed a functor, we prove some lemmas about *m2z*:

```
lemma (in LSCategory) m2zz2m:
  assumes "f maps X to Y" shows "(m2z f) |∈| (Hom X Y)"

lemma (in LSCategory) m2zz2mInv:
  assumes "f ∈ mor"
  shows "z2m (m2z f) = f"

lemma (in LSCategory) z2mm2z:
  assumes "X ∈ obj" and "Y ∈ obj" and "f |∈| (Hom X Y)"
  shows "z2m f maps X to Y ∧ m2z (z2m f) = f"
```

After some work, we get these:

```
lemma HomFtorFtor:
  assumes "LSCategory C"
  and     "X ∈ obj_C"
  shows   "Functor (Hom_C[X,-])"

lemma HomFtorObj:
  assumes "LSCategory C"
  and     "X ∈ obj_C" and "Y ∈ obj_C"
  shows   "(Hom_C[X,-]) @@ Y = Hom_C X Y"
```

Then we define the *contravariant hom functor*

```
definition
  HomFtorMapContra ("HomCı[_,_]" [65,65] 65) where
  "HomFtorMapContra C g X ≡ ZFfun (Hom_C (cod_C g) X)
                  (Hom_C (dom_C g) X)
                  (λ f . m2z_C (g ;;_C (z2m_C f)))"

definition   HomFtorContra' ("HomPı[-,_]" [65] 65) where
  "HomFtorContra' C X ≡ (|
        CatDom = (Op C),
        CatCod = SET ,
        MapM   = λ g . HomC_C[g,X]
  |)"

definition HomFtorContra ("Homı[-,_]" [65] 65) where
    "HomFtorContra C X ≡ MakeFtor(HomFtorContra' C X)"
```

We prove that this is a functor by showing that it is the covariant hom functor in the opposite category. As a side note, it is thanks to the more field of a record that we can speak of the opposite of a locally small category. Of course, we still need another lemma to show that the opposite of a locally small category is locally small, but that is easy and when we have that, we easily prove that the contravariant hom functor is a functor and find its action on objects:

```
lemma HomFtorContra: "Hom_C[-,X] = Hom_(Op C)[X,-]"
```

```
lemma HomFtorContraFtor:
  assumes "LSCategory C"
  and     "X ∈ obj_C"
  shows   "Ftor (Hom_C[-,X]) : (Op C) ⟶ SET"
```

```
lemma HomFtorOpObj:
  assumes "LSCategory C"
  and     "X ∈ obj_C" and "Y ∈ obj_C"
  shows   "(Hom_C[-,X]) @@ Y = Hom_C Y X"
```

The last lemma in `SetCat.thy` is a long lemma which basically expresses the naturality of the image of morphisms under the Yoneda embedding.

2.4 Yoneda Embedding

The Yoneda embedding is a functor from a locally small category to its presheaf category; each morphism in the domain category is mapped to a natural transformation between hom functors:

```
definition "YFtorNT' C f ≡ (|
    NTDom = Hom_C[-,dom_C f] ,
    NTCod = Hom_C[-,cod_C f] ,
    NatTransMap = λ B . Hom_C[B,f] |)"
```

```
definition "YFtorNT C f ≡ MakeNT (YFtorNT' C f)"
```

```
definition
  "YFtor' C ≡ (|
          CatDom = C ,
          CatCod = CatExp (Op C) SET ,
          MapM = λ f . YFtorNT C f
  |)"
```

```
definition "YFtor C ≡ MakeFtor(YFtor' C)"
```

And we prove that

```
lemma YFtorFtor:
    assumes "LSCategory C"
    shows "Ftor (YFtor C) : C ⟶ (CatExp (Op C) SET)"
```

```
lemma YFtorObj:
```

```
  assumes "LSCategory C" and "X ∈ Obj C"
  shows "(YFtor C) @@ X = Hom_C [-,X]"
```

The importance of *YFtor C* is that it is a full and faithful embedding, showing that every locally small category is isomorphic to a full subcategory of its presheaf category. A functor is *full* iff its restriction to the hom sets is surjective and it is *faithful* iff this restriction is injective and it is an *embedding* iff the object part of the functor is injective. We prove that *YFtor C* has these three properties at the end of `Yoneda.thy`. But first, we need to prove *Yoneda's Lemma*, which asserts that for any *X ∈ Obj C* and any

Ftor F : (Op C) ⟶ SET

that

HOMCollection (CatExp (Op C) SET) (Hom_C [-,X]) F

is in bijective correspondence with *explode(F@@X)*. In words, there is a bijective correspondence between the natural transformations from *Hom_C [-,X]* to *F* and the elements of *F@@X*. We will define such a bijection and call it *YMap C X* (the dependence on *F* is contained in its arguments (the natural transformations with codomain *F*)). In fact, it may be shown that *YMap C X* are the components of a natural isomorphism, but to write down the domain we need to know that the *HomCollection* above is in fact a set. Yoneda's lemma shows that this is true, but in general it is not since the presheaf category of a locally small category may not be locally small ([FS95]). This problem is ignored in [AB03] (but carefully handled in [Cro93, Lemma 2.7.4]) and I only became aware of it when I received a rude awakening from the Isabelle type checker when I naively tried to write down the hom functor of the presheaf category. However, we do not need to know that the bijections are components of a natural transformation for the proofs that we want to do here.

YMap C X and its inverse are defined as follows:

```
definition "YMap C X η ≡ (η $$ X) |@| (m2z_C (id_C X))"

definition "YMapInv' C X F x ≡ (|
      NTDom = ((YFtor C) @@ X),
      NTCod = F,
      NatTransMap = λ B . ZFfun (Hom_C B X) (F @@ B)
                   (λ f . (F ## (z2m_C f)) |@| x)
  |)"
definition "YMapInv C X F x ≡ MakeNT (YMapInv' C X F x)"
```

and we prove that they are in fact inverses of eachother on the correct domains in the following two lemmas[3] (which together constitute

[3] *YMap1* uses the lemma *NatTransExt*, which uses extensionality to show that two natural transformations are equal if their domain and codomain are the same and their maps are the same at objects

Yoneda's Lemma):

```
lemma YMap1:
  assumes "LSCategory C"
  and "Ftor F : (Op C) ⟶ SET"
  and "X ∈ Obj C"
  and "NT η : (YFtor C @@ X) ⟹ F"
  shows "YMapInv C X F (YMap C X η) = η"
```

```
lemma YMap2:
  assumes "LSCategory C"
  and "Ftor F : (Op C) ⟶ SET" and "X ∈ Obj C"
  and "x |∈| (F @@ X)"
  shows "YMap C X (YMapInv C X F x) = x"
```

The reason that Yoneda's Lemma allows us to prove the properties of *YFtor C* is that *YFtor C* is related to *YMapInv*:

```
lemma YMapYoneda:
  assumes "LSCategory C" and "f maps_C X to Y"
  shows "YFtor C ## f = YMapInv C X (YFtor C @@ Y) (m2z_C f)"
```

Then fullness and faithfulness follow very quickly by using this relationship and applying the bijection:

```
lemma YonedaFull:
  assumes "LSCategory C" and "X ∈ Obj C" and "Y ∈ Obj C"
  and "NT η : (YFtor C @@ X) ⟹ (YFtor C @@ Y)"
  shows "YFtor C ## (z2m_C (YMap C X η)) = η"
  and "z2m_C (YMap C X η) maps_C X to Y"
```

```
lemma YonedaFaithful:
  assumes "LSCategory C" and "f maps_C X to Y"
  and "g maps_C X to Y"
  and "YFtor C ## f = YFtor C ## g"
  shows "f = g"
```

Finally, we show that *YFtor C* is an embedding:

```
lemma YonedaEmbedding:
  assumes "LSCategory C" and "A ∈ Obj C" and "B ∈ Obj C"
  and "(YFtor C) @@ A = (YFtor C) @@ B"
  shows "A = B"
```

The proof of this lemma brings together many of the concepts that we have defined thus far, so I will outline the main steps. Due to the lemma *YFtorObj* this amounts to showing that if

$$Hom_C[-,A] = Hom_C[-,B]$$

then *A = B* for objects *A* and *B* in the category *C*. By applying each functor to the identity at A we get

$$\mathit{Id\ SET\ (Hom_C\ A\ A) = Id\ SET\ (Hom_C\ A\ B)}$$

By using that Id is an injection (and that SET is a category), we get that

$$\mathit{Hom_C\ A\ A = Hom_C\ A\ B}$$

The proof is completed by noting that

$$\mathit{(m2z_C\ (Id\ C\ A))\ |{\in}|\ (Hom_C\ A\ A)}$$

and we conclude from this and the equality of hom sets that

$$\mathit{cod_C\ (Id\ C\ A) = B}$$

but we know that $\mathit{cod_C\ (Id\ C\ A) = A}$ by the axioms for a category.

3 Monadic Equational Logic

One of the important uses of category theory is to provide a class of models for various logics. For example, the class of models for the typed lambda calculus is the cartesian closed categories. Types are interpreted as objects in the underlying category of the model and lambda expressions are interpreted as morphisms. A good presentation is found in [Cro93]. As an illustration of formalization in this area, we experiment with a very simple logic as presented in Moggi's fundamental paper on monads [Mog91]: the many sorted monadic equational logic. This logic is interpreted in general categories. Throughout the paper he progressively extends this logic, restricting the class of categories that model it while retaining soundness and completeness. After monadic equational logic, which is interpreted in any category, the next step is a logic that is interpreted in categories with a monad, then in categories with a strong monad, then finally in strong monads over a topos. But the essential ideas of interpreting a logic in a class of categories are contained in the first example and, from this narrow perspective, the rest of the paper consists of a sequence of steps to complicate the initial example. Although I have only formalized the first step, I would not be surprised if it could be used as a blueprint for the more sophisticated logics, once the concomitant category theory has been formalized.

The following subsections describe the Isabelle theory contained in `MonadicEquationalTheory.thy`

3.1 Soundness

The signature of a language for monadic equational logic contains a class of function symbols and base type symbols where each function symbol has a domain symbol and a codomain symbol in the base type symbols. We allow type polymorphism, although in proving completeness we shall specialize to type ZF, by embedding the higher order structures into a model of ZFC.

```
record ('t,'f) Signature =
  BaseTypes :: "'t set" ("Tyı")
  BaseFunctions :: "'f set" ("Fnı")
  SigDom :: "'f ⇒ 't" ("sDomı")
  SigCod :: "'f ⇒ 't" ("sCodı")

locale Signature =
  fixes S :: "('t,'f) Signature" (structure)
  assumes Domt: "f ∈ Fn ⟹ sDom f ∈ Ty"
  and     Codt: "f ∈ Fn ⟹ sCod f ∈ Ty"
```

We abbreviate that `f` is a function symbol in the signature `S` with domain `A` and codomain `B` by

$$f \in Sig\ S : A \rightarrow B$$

The language is built inductively from the signature. In BNF, an *expression* is

$$e ::= Vx \mid f\ E@\ e$$

where `f` ranges over the base functions of the signature, and `Vx` is the single variable symbol (there is only one in a monadic theory). So an expression is either the variable symbol or a finite number of function applications on it. The *language* is either a type $\vdash$ *A Type*, a *term* $Vx : A \vdash e : B$ or an *equation* $Vx : A \vdash e \equiv d : B$. Here, `A` and `B` have type `'t` and `e` and `d` are expressions. The well-defined sentences in the language are defined inductively in the inductive set `WellDefined`. If a sentence φ is well-defined relative to the signature `S` we write $Sig\ S \rhd \varphi$.

This is the first theorem in the theory file and it is used a lot when doing induction over terms in the language, because the premise here is the base case of the induction. It is proved easily by cases.

```
lemma SigId: "Sig S ▷ (Vx : A ⊢ Vx : B) ⟹ A = B"
apply(rule WellDefined.cases)
by simp+
```

This is the first induction that we do in this theory, and it illustrates the main points. It is necessary to quantify over `B` because this will take the value `B'` in the induction hypothesis, corresponding to the domain of the function symbol, and it will take a different value `B` in the conclusion of the induction step; without quantification, it would take a fixed value at the outset.

```
lemma (in Signature) SigTy:
"⋀ B . Sig S ▷ (Vx : A ⊢ e : B) ⟹
                        (A ∈ BaseTypes S ∧ B ∈ BaseTypes S)"
proof(induct e)
  {
    fix B assume a: "Sig S ▷ Vx : A ⊢ Vx : B"
```

```
    have "A = B" using a SigId[of S] by simp
    thus "A ∈ Ty ∧ B ∈ Ty" using a by auto
  }
  {
    fix B f e assume ih:
      "⋀B'. Sig S ▷ (Vx : A ⊢ e : B') ⟹ A ∈ Ty ∧ B' ∈ Ty"
    and a: "Sig S ▷ (Vx : A ⊢ (f E@ e) : B)"
    from a obtain B' where f: "f ∈ Sig S : B' → B" and
      "Sig S ▷ (Vx : A ⊢ e : B')" by auto
    hence "A ∈ Ty" using ih by auto
    moreover have "B ∈ Ty" using f
       by (auto simp add: funsignature_abbrev_def Codt)
    ultimately show "A ∈ Ty ∧ B ∈ Ty" by simp
  }
qed
```

In the more sophisticated examples in Moggi's paper, where expressions can take more than just two forms, there would be more cases to consider in each induction, but the statements of the theorems and the general structure of the proof should go through largely unchanged (or the present theory should be modified if not).

Now we prepare for interpreting sentences of the language in a category. The interpretation of a sentence is either a boolean, a morphism, or an object. Here is a type for the value of an interpretation:

```
datatype ('o,'m) IType = IObj 'o | IMor 'm | IBool bool
```

An interpretation consists of a signature, a category, and instructions how to interpret the base types as objects and the base functions as morphisms. The definition of the interpretation function (*Interp*) is taken directly from Moggi's paper [Mog91, Table 1]

```
record ('t,'f,'o,'m) Interpretation =
  ISignature :: "('t,'f) Signature" ("iSı")
  ICategory  :: "('o,'m) Category"  ("iCı")
  ITypes     :: "'t ⇒ 'o" ("Ty⟦_⟧ı")
  IFunctions :: "'f ⇒ 'm" ("Fn⟦_⟧ı")

locale Interpretation =
  fixes I :: "('t,'f,'o,'m) Interpretation" (structure)
  assumes ICat: "Category  iC"
  and     ISig: "Signature iS"
  and     It  : "A ∈ BaseTypes iS ⟹ Ty⟦A⟧ ∈ Obj iC"
  and     If  : "(f ∈ Sig iS : A → B) ⟹
                 Fn⟦f⟧ maps_iC Ty⟦A⟧ to Ty⟦B⟧"

inductive Interp  ("L⟦_⟧ı → _") where
    InterpTy: "Sig iS_I ▷ ⊢ A Type ⟹
                    L⟦⊢ A Type⟧_I → (IObj Ty⟦A⟧_I)"
```

```
| InterpVar: "L⟦⊢ A Type⟧_I → (IObj c) ⟹
             L⟦Vx : A  ⊢ Vx : A⟧_I → (IMor (Id iC_I c))"
| InterpFn: "⟦Sig iS_I ▷ Vx : A  ⊢ e : B ;
             f ∈ Sig iS_I : B → C ;
             L⟦Vx : A  ⊢ e : B⟧_I → (IMor g)⟧ ⟹
   L⟦Vx : A  ⊢ (f E@ e) : C⟧_I → (IMor (g ;;_ICategory I Fn⟦f⟧_I))"
| InterpEq: "⟦L⟦Vx : A  ⊢ e1 : B⟧_I → (IMor g1) ;
             L⟦Vx : A  ⊢ e2 : B⟧_I → (IMor g2)⟧ ⟹
             L⟦Vx : A  ⊢ e1 ≡ e2 : B⟧_I → (IBool (g1 = g2))"
```

We first show that if a sentence evaluates to some value under the interpretation, then it is well-defined (we prove the corresponding lemma for terms first):

lemma *(in Interpretation)*
WellDefined: "L⟦φ⟧ → i ⟹ Sig iS ▷ φ"

The first important theorem is that the interpretation is functional (once again, we prove the lemma for terms first):

lemma *(in Interpretation)*
Functional: "⟦L⟦φ⟧ → i1 ; L⟦φ⟧ → i2⟧ ⟹ i1 = i2"

Then we show that if a term has domain type *A* and a codomain type *B* then the morphism that it evaluates to has the correct domain and codomain in the category:

lemma *(in Interpretation) Expr2Mor:*
"L⟦Vx : A ⊢ e : B⟧ → (IMor g) ⟹ (g maps$_{iC}$ Ty⟦A⟧ to Ty⟦B⟧)"

And a step back from this, we show that if a term is well-defined (relative to the signature of the interpretation) then there exists some morphism to which it evaluates under the interpretation:

lemma *(in Interpretation) Sig2Mor:*
assumes *"(Sig iS ▷ Vx : A ⊢ e : B)"*
shows *"∃ g . L⟦Vx : A ⊢ e : B⟧ → (IMor g)"*

Axioms consist of a signature and a set of (well-defined) equation sentences in the language of that signature. Substitution of expressions for the single variable is also defined in the obvious way, such that it satisfies the following lemmas:

lemma *SubstXinE: "(sub Vx in e) = e"*
by *(induct e, auto simp add: Subst_def)*

lemma *SubstAssoc:*
"sub a in (sub b in c) = sub (sub a in b) in c"

by *(induct c, (simp add: Subst_def)+)*

lemma *SubstWellDefined: "$\bigwedge$ C . $[\![$Sig S $\triangleright$ (Vx : A $\vdash$ e : B); Sig S $\triangleright$ (Vx : B $\vdash$ d : C)$]\!]$ $\Longrightarrow$ Sig S $\triangleright$ (Vx : A $\vdash$ (sub e in d) : C)"*

The inductive set *Theory* is defined from the axioms ([Mog91, Table 2]). And we prove that if a sentence is in the theory, then it is well-defined (relative to the signature of the axioms):

lemma *(***in** *Axioms) Equiv2WellDefined: "φ $\in$ Theory $\Longrightarrow$ Sig aS $\triangleright$ φ"*

We also prove that the theory is closed under substitution, when keeping the second variable fixed. The case where the first variable is fixed is true by definition of *Theory* (*Axioms.Subst*).

lemma *(***in** *Axioms) Subst':*
"$\bigwedge$ C . $[\![$Sig aS $\triangleright$ Vx : B $\vdash$ d : C ;
(Vx : A $\vdash$ e1 $\equiv$ e2 : B) $\in$ Theory$]\!]$ $\Longrightarrow$
(Vx : A $\vdash$ (sub e1 in d) $\equiv$ (sub e2 in d) : C) $\in$ Theory"

A *Model* consists of an *Interpretation* and *Axioms* where the signatures are the same and all equation sentences in the axioms evaluate to *IBool True* under the interpretation. One of the two main theorems is that the monadic equational logic is sound. It is proved by induction on the *Theory*, and uses a lemma about the connection between syntactic substitution and semantic composition, which is the first time that the axioms of a category are used.

lemma *(***in** *Interpretation) SubstComp:*
"$\bigwedge$ h C . $[\![$(L$[\![$Vx : A $\vdash$ e : B$]\!]$ $\rightarrow$ (IMor g)) ;
(L$[\![$Vx : B $\vdash$ d : C$]\!]$ $\rightarrow$ (IMor h))$]\!]$ $\Longrightarrow$
(L$[\![$Vx : A $\vdash$ (sub e in d) : C$]\!]$ $\rightarrow$ (IMor (g ;;$_{iC}$ h)))"

lemma *(***in** *Model) Sound: "φ $\in$ Theory $\Longrightarrow$ L$[\![\varphi]\!]$ $\rightarrow$ (IBool True)"*

3.2 Completeness

To show that monadic equational logic is complete, we construct, for each set of axioms *T*, an interpretation, called the *Canonical Interpretation* (abbreviated *CI T*), which satisfies

lemma *CIModel: "ZFAxioms T $\Longrightarrow$ Model (CI T) T"*

lemma *CIComplete:*
assumes *"ZFAxioms T"* **and** *"L$[\![\varphi]\!]_{CI\ T}$ $\rightarrow$ (IBool True)"*
shows *"φ $\in$ Axioms.Theory T"*

where

```
locale ZFAxioms = Ax : Axioms Ax
  for Ax :: "(ZF,ZF) Axioms" (structure) +
  assumes fnzf: "BaseFunctions (aSignature Ax) ∈ range explode"
```

Then completeness is succinctly expressed and proved:

```
lemma Complete:
  assumes "ZFAxioms T"
  and "⋀ (I :: (ZF,ZF,ZF,ZF) Interpretation) .
          Model I T ⟹ (L⟦φ⟧_I → (IBool True))"
  shows "φ ∈ Axioms.Theory T"
proof-
  have "Model (CI T) T" using assms CIModel by simp
  thus ?thesis using CIComplete[of T φ] assms by auto
qed
```

The underlying category of the canonical interpretation is called the *Canonical Category*. Its objects are the base types of the signature of the axiom scheme. Morphisms from type *A* to type *B* are classes of expressions of the form:

```
{e' . (Vx : A ⊢ e' ≡ e : B) ∈ Axioms.Theory T}
```

So each morphism is an equivalence class of expressions where the equivalence relation on expressions is induced by the theory. Composition is given by substitution, once we have shown that substitution is compatible with the equivalence relation (which we do in *CanonicalCompWellDefined* using *Axioms.Subst* and *Axioms.Subst'*). In addition to the class of expressions, we also need to record the domain *A* and codomain *B* in the data representing a morphism, otherwise we will end up with the empty class belonging to every hom set. We therefore make the following definitions:

```
record ('t,'f) TermEquivClT =
  TDomain :: 't
  TExprSet :: "('t,'f) Expression set"
  TCodomain :: 't

definition "TermEquivClGen T A e B ≡
   {e' . (Vx : A ⊢ e' ≡ e : B) ∈ Axioms.Theory T}"
definition "TermEquivCl' T A e B ≡
   ⦇ TDomain = A , TExprSet = TermEquivClGen T A e B ,
     TCodomain = B⦈"
```

The next step is to define the canonical category. But if we did this now (using *TermEquivCl'* for the morphisms), we would end up with an interpretation of type

```
(ZF,ZF,ZF,(ZF,ZF)TermEquivClT) Interpretation
```

which is no good, because there would be a type clash with the completeness theorem we want to prove. What we need is an encoding of *(ZF,ZF)TermEquivClT* in *ZF*. That is, a map *m2ZF* such that

```
lemma m2ZFinj_on: "ZFAxioms T ⟹
    inj_on m2ZF {TermEquivCl' T A e B | A e B . True}"
```

from which we get

```
definition ZF2m :: "(ZF,ZF) Axioms ⇒ ZF ⇒ (ZF,ZF) TermEquivClT"
where "ZF2m T ≡
   inv_into {TermEquivCl' T A e B | A e B . True} m2ZF"

lemma ZF2m: "ZFAxioms T ⟹
ZF2m T (m2ZF (TermEquivCl' T A e B)) = (TermEquivCl' T A e B)"
```

Then we can define

```
definition TermEquivCl ("[_,_,_]ı") where
    "[A,e,B]_T ≡ m2ZF (TermEquivCl' T A e B)"

definition "CLDomain T ≡ TDomain o ZF2m T"
definition "CLCodomain T ≡ TCodomain o ZF2m T"
```

Then everything goes through as it would have done except that *TDomain* is replaced by *CLDomain T* and *TCodomain* is replaced by *CLCodomain T*. And we define the canonical category thus:

```
definition "CanonicalCat' T ≡ (|
  Obj = BaseTypes (aS_T),
  Mor = {[A,e,B]_T | A e B . Sig aS_T ▷ (Vx : A ⊢ e : B)},
  Dom = CLDomain T,
  Cod = CLCodomain T,
  Id  = (λ A . [A,Vx,A]_T),
  Comp = CanonicalComp T
  |)"

definition "CanonicalCat T ≡ MakeCat (CanonicalCat' T)"
```

where we define

```
definition "CanonicalComp T f g ≡
  THE h . ∃ e e' .
    h = [CLDomain T f,sub e in e',CLCodomain T g]_T ∧
    f = [CLDomain T f,e,CLCodomain T f]_T ∧
    g = [CLDomain T g,e',CLCodomain T g]_T"
```

and prove

lemma *CanonicalCompWellDefined:*
  assumes *zaxt: "ZFAxioms T"*
    and *"Sig aS$_T$ ▷ (Vx : A ⊢ d : B)"*
    and *"Sig aS$_T$ ▷ (Vx : B ⊢ d' : C)"*
  shows *"CanonicalComp T [A,d,B]$_T$ [B,d',C]$_T$ = [A,sub d in d',C]$_T$"*

using *Axioms.Subst* and *Axioms.Subst'* and

lemma *Equiv2Cl:*
  assumes *"Axioms T"*
    and *"(Vx : A ⊢ e ≡ d : B) ∈ Axioms.Theory T"*
  shows *"[A,e,B]$_T$ = [A,d,B]$_T$"*

lemma *Cl2Equiv:*
  assumes *axt: "ZFAxioms T"*
    and *sa: "Sig aS$_T$ ▷ (Vx : A ⊢ e : B)"*
    and *cl: "[A,e,B]$_T$ = [A,d,B]$_T$"*
  shows *"(Vx : A ⊢ e ≡ d : B) ∈ Axioms.Theory T"*

After these last two lemmas, we can forget about the encoding into *ZF*, namely *m2ZF*, and we proceed as we would have done without the encoding, working directly with the HOL datatypes, except for the extra argument to *CLDomain* and *CLCodomain*, which comes about because of their dependency on the axiom scheme, which comes from the definition of *ZF2m*.

We prove that the canonical category is a category using *MakeCat*

lemma *CanonicalCatCat':*
    "ZFAxioms T ⟹ Category_axioms (CanonicalCat' T)"

lemma *CanonicalCatCat:*
    "ZFAxioms T ⟹ Category (CanonicalCat T)"
by *(simp add: CanonicalCat_def CanonicalCatCat' MakeCat)*

Then we define the canonical interpretation:

definition *CanonicalInterpretation* **where**
```
"CanonicalInterpretation T ≡ (|
  ISignature = aSignature T,
  ICategory  = CanonicalCat T,
  ITypes     = λ A . A,
  IFunctions = λ f . [SigDom (aSignature T) f,
                      f E@ Vx,
                      SigCod (aSignature T) f]_T
|)"
```

abbreviation *"CI T ≡ CanonicalInterpretation T"*

We prove that it is an *Interpretation* and that terms evaluate to the equivalence class of their expression with the same domain and codomain (which we do by induction over *e*):

```
lemma CIInterpretation:
    "ZFAxioms T ⟹ Interpretation (CI T)"
```

```
lemma CIInterp2Mor: "ZFAxioms T ⟹
    (⋀ B . Sig iS_CI T ▷ (Vx : A ⊢ e : B) ⟹
            L⟦Vx : A ⊢ e : B⟧_CI T → (IMor [A, e, B]_T))"
```

We then show that the canonical interpretation of a theory models it, and finally the completeness theorems stated earlier.

The only remaining task is to define the encoding *m2ZF* so that it is an injection on the class

{TermEquivCl' T A e B | A e B . True}

for every axiom scheme *T* with *ZFAxioms T*. First, we define an encoding of expressions:

```
primrec Expr2ZF :: "(ZF,ZF) Expression ⇒ ZF" where
    "Expr2ZF Vx = ZFTriple (nat2Nat 0) (nat2Nat 0) Empty"
  | "Expr2ZF (f E@ e) = ZFTriple (SucNat (ZFTFst (Expr2ZF e)))
                                 (nat2Nat 1)
                                 (Opair f (Expr2ZF e))"
```

So an expression is encoded as a triple. The first entry is the depth in the construction tree (its complexity), the second is the number of the rule that was last used, and the third is a tuple that stores the encoding of the components in the constructor. So, for example, *f E@ Vx* would be encoded as

(1,1,(f,(0,0,0)))

since the depth in the construction tree is 1, the construction rule (i.e. function application) is the second one, and there are two arguments to this constructor: *f* and *Vx* while the encoding for *Vx* is *(0,0,0)*, which is a triple of empty sets. This encoding should work for any algebraic data type. *ZFTriple* is defined in `Universe.thy` and defines a triple using ordered pairs. *nat2Nat* converts *nat* into the *ZF* representation of a natural number. In the lemma *Expr2ZFinj* we prove that *Expr2ZF* is an injection by induction on the complexity of the formula (which we can do because the complexity data is in the encoding).

We define

```
definition m2ZF :: "(ZF,ZF) TermEquivClT ⇒ ZF" where
  "m2ZF t ≡ ZFTriple (TDomain t)
        (implode (Expr2ZF ‘ (TExprSet t)))
        (TCodomain t)"
```

Since *implode* is the inverse of *explode*, all we need to show is

```
lemma TermSetInSet: "ZFAxioms T ⟹
    Expr2ZF ` (TermEquivClGen T A e B) ∈ range explode"
```

We do this by observing that

```
TermEquivClGen T A e B⊆ {e' . (Sig aS_T ▷ (Vx : A ⊢ e' : B))}
```

and splitting this class into the complexity classes:

```
primrec WellFormedToSet ::
"(ZF,ZF) Signature ⇒ nat ⇒ (ZF,ZF) Expression set" where
  "WellFormedToSet S 0 = {Vx}"
| "WellFormedToSet S (Suc n) = (WellFormedToSet S n) ∪
      { f E@ e | f e . f ∈ BaseFunctions S ∧
                       e ∈ (WellFormedToSet S n)}"
```

So that *WellFormedToSet S n* is the class of expressions in signature *S* with complexity less than or equal to *n*. We will then complete the proof by showing by induction that

```
lemma WellDefinedToWellFormedSet:
"⋀ B . (Sig S ▷ (Vx : A ⊢ e : B)) ⟹
                                ∃n. e ∈ WellFormedToSet S n"
lemma WellFormedToSetInRangeExplode: "ZFAxioms T ⟹
    (Expr2ZF ` (WellFormedToSet aS_T n)) ∈ range explode"
```

and we piece them together using theorems about *explode* in `Universe.thy`; for example

```
lemma ZFUnionNatInRangeExplode:
  "(⋀ (n :: nat) . f n ∈ range explode) ⟹
                          (⋃ n . f n) ∈ range explode"
```

4 Conclusions and further work

In addition to the code that I have described in the main text, I have also included `PartialBinaryAlgebra.thy` in the source code, which is the skeleton of a theory to show the equivalence between the definition of category given in `Cat.thy` and the *object free* definition of category ([AHS04, Definition 3.53]). The equivalence is a functor between the quasi category of all categories and the quasi category of object free categories (which are based on partial binary algebras). I have included the theory in part to present the skeleton as a demonstration of the top-down philosophy of writing theories, and in part to show the expressive power of the theory as written in Isabelle/HOL. As to the latter, here is the definition of the quasi-category of all categories:

```
definition
  qCategory' ::  "(('a, 'a) Category,
        ('a, 'a, 'a, 'a, unit, unit) Functor) Category" where
  "qCategory' = (|
      Obj = Category  ,
      Mor = Functor  ,
      Dom = CatDom  ,
      Cod = CatCod  ,
      Id = IdentityFunctor  ,
      Comp = FunctorComp
  |)"
```

```
definition "qCategory = MakeCat (qCategory')"
```

As to the philosophy: we make all the definitions in full, and state the major theorems and then proceed to prove them from the top down, by recursive refinement. Part of this process will include finding 'bugs' in the stated definitions, with any subsequent changes reflecting our improved understanding of a mathematical theory as we formalize it. For example, it could turn out, in the process of filling out this skeleton theory, that the definition of *Category* needs slight modification. In this way, formalizing a theory is an instance of the scientific method, in that proving a theorem with a given set of definitions is an attempt to refute the validity of those definitions or the statement of the theorem.

We now list some ideas for immediate extensions to the theory. Yoneda's lemma is a generalization of Cayley's theorem for groups and this is shown in [Cro93, Example 2.7.5]. It would be nice to formalize this by combining this theory with an existing theory of groups. It would also be nice to combine this theory with an existing theory of vector spaces to show the naturality of the double dual isomorphism. Once a theory of monads and Kleisli triples has been made (see `Monad.thy` for a stub implementation) then more of Moggi's paper can be formalized. A major omission in this development is limits. One could start by defining products, and work towards a general formulation of limits, and a theory of adjunctions. On the horizon, once enough category theory has been formalized, maybe special proof techniques can be defined for common category theoretic proof tactics. And since category theory is a very visual theory, it would be nice to be able to convert commutative diagrams into Isabelle code somehow.

References

[AB03] Stephen Awodey and Andrej Bauer. *Lecture Notes: Introduction to Categorical Logic*. 2003.

[AHS04] Jiří Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. 2004.

[Cro93] Roy L. Crole. *Categories for Types*. 1993.

[FS95] Peter Freyd and Ross Street. *On the Size of Categories*. 1995.

[KS06] Masaki Kashiwara and Pierre Schapira. *Categories and Sheaves*. 2006.

[Mog91] Eugenio Moggi. *Notions of computation and monads*. 1991.

[O'K09] Greg O'Keefe. *Category Theory to Yoneda's Lemma*. 2009.