Experiments in Formalizing Basic Category Theory in Higher Order Logic and Set Theory

Sten Agerholm

December 22, 1995

DRAFT

Contents

1 Introduction **2**

2 Higher Order Logic **2**

3 Category Theory in Higher Order Logic **3**

3.1 Definition of Category 3
3.2 The Category of Sets 4
3.3 Arrows, Morphisms and Hom Sets 4
3.4 Definition of Functor 5
3.5 The Category of Categories 5
3.6 Commuting Squares 6
3.7 The Functor Category and Other Constructions 6
3.8 The Hom Functor 7
3.9 Comments 7

4 Category Theory in Higher Order Logic and Set Theory **8**

4.1 ZF Set Theory in HOL 8
4.2 Formalizing the Category of Sets in ZF 10

5 Conclusions **11**

1 Introduction

Set theory is the standard foundation for mathematics. However, the majority of general-purpose theorem provers support versions of type theory. Examples include ALF, Coq, HOL, LEGO, Nuprl and PVS. For many applications type theory works well and provides, for specification, the benefits of type checking that are well-known in programming. However, there are areas where types get in the way or seem unmotivated. For instance, the classical construction of the natural numbers via the set $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots\}$ is impossible in type theory and Scott's classical set-theoretic construction of a non-trivial model D_∞ of the λ-calculus is impossible in simple type theory but could probably be performed in dependently typed systems, which on the other hand do not provide fully automatic type checking.

A question now is: Would it be desirable to base theorem provers on set theory rather than type theory in order to support such applications. It was already demonstrated in [2, 3] that the inverse limit construction, which is a categorical method for constructing solutions to recursive domain equations that generalizes Scott's original set-theoretic construction, can be formalized in Zermelo-Fraenkel (ZF) set theory. As an example to help shed further light on this question this research will consider the formalization of some basic concepts of category theory in a mechanical theorem prover which supports both higher order logic and ZF set theory; this is an axiomatic extension of the HOL system [5] with ZF set theory, called HOL-ST [6].

NB! None of the definitions and theorems presented here have actually been entered into HOL or HOL-ST. We encountered the problems mentioned below before coming to that part of the formalization process.

2 Higher Order Logic

This section provides a quick introduction to the higher order logic of the HOL system, which is roughly Church's simply typed λ-calculus extended with ML-style polymorphism. We shall not go into the details of the theorem proving infrastructure of HOL but mainly concentrate on logic issues such that a reader will be able to read the syntax of the rest of the paper without knowing HOL in advance.

The terms of the HOL logic can be variables, constants, λ-abstractions (written `\x.t`) and applications (written `t1 t2`). The usual logical connectives are represented as constants. Types can be atomic types (like `bool` for the boolean truth values), type variables (like `*`, `**` or `*o`), compound types (like `*#**` for the product type), and function types `*->**`. Type variables range over any type. All terms must be well-typed in the usual sense. Types can usually be inferred automatically in ML by type inference but terms may also be typed explicitly using ':', e.g. `t:*#**->bool`, where parentheses can be omitted since product types binds stronger than function types.

The HOL logic is a higher-order logic, so it is possible to quantify over any type. Universal and existential quantification are written `!x. t` and `?x. t`, where `t` may contain `x` and the type of `x` may be any valid HOL type, including a type variable. Restricted quantification is also supported: universal quantification `!x::P. t` abbreviates `!x. P x ==> t`, where `==>` is logical implication, and existential quantification `?x::P. t` abbreviates `?x. P x /\ t`, where `/\` is conjunction. Disjunction is written `\/`, negation `~`, truth `T` and falsity `F`. Unique existence is written as `?!x. t`. The usual projection functions associated with the product type are written `FST` and `SND`. Function composition is written `o`. Local declarations can be written using `let x = t in t'[x]`.

The restricted λ-abstraction `\x::P. t` yields a function which equals `t[t'/x]` when applied to a term `t'` satisfying `P` and `ARB` otherwise. The constant `ARB` is a fixed but arbitrary value of some type. It is defined using the choice operator, written `@x. t` and used to select an element of some type such that some predicate term holds (in the case of `ARB` the predicate is just truth).

HOL has a large collection of built-in types, theorems and proof tools to support all kinds of reasoning. For instance, the predicate sets library, which provides typed set theory (not ZF untyped set theory) is used in the development below. Sets are represented as subsets of HOL types, via predicates of type `*->bool`. It provides set notation, like set membership `x::s` and set abstraction `{x::s | P x}`, and the usual operations on sets.

3 Category Theory in Higher Order Logic

This section introduces some basic notions of category theory and presents a formalization in higher order logic. In Section 3.9 a number of problems and limitations of the formalization are discussed.

3.1 Definition of Category

The following definition is copied almost directly from Jaap van Oosten's "Basic Category Theory" [9]:

> A category $\mathcal{C}$ is given by a class $\mathcal{C}_0$ of objects and a class $\mathcal{C}_1$ of arrows which have the following structure:
>
> - Each arrow has a domain and a codomain which are objects; one writes $f : X \to Y$ if X is the domain and Y the codomain of the arrow f. One also writes $X = \mathrm{dom}(f)$ and $Y = \mathrm{cod}(f)$.
> - Given two arrows f and g such that $\mathrm{cod}(f) = \mathrm{dom}(g)$, the composition of f and g, written as $g\,f$, is defined, i.e. is an arrow, and has domain $\mathrm{dom}(f)$ and codomain $\mathrm{cod}(g)$.
> - Composition is associative: given $f : X \to Y$, $g : Y \to Z$ and $h : Z \to W$, $h\,(g\,f) = (h\,g)\,f$.
> - For every object X there is an identity arrow $\mathrm{id}_X : X \to X$, satisfying $\mathrm{id}_X\,g = g$ for every $g : Y \to X$ and $f\,\mathrm{id}_X = f$ for every $f : X \to Y$.

This can be translated easily to higher order logic:

```
DEF (Category):
  category(C_0:*o->bool,C_1:*a->bool,dom,cod,c) =
   (!f::C_1. (dom f)::C_0 /\ (cod f)::C_0) /\
   (!f g::C_1.
     (cod f = dom g) ==>
     let h = c g f in h::C_1 /\ (dom h = dom f) /\ (cod h = cod g)) /\
   (!f g h::C_1.
     (cod f = dom g) /\ (cod g = dom h) ==> (c h(c g f) = c(c h g)f)) /\
   (!X::C_0.
     (?id::C_1.
       (dom id = X) /\ (cod id = X) /\
       (!Y::C_0.
         (!f::C_1. (dom f = X) /\ (cod f = Y) ==> (c f id = f)) /\
         (!f::C_1. (dom f = Y) /\ (cod f = x) ==> (c id f = f)))))
```

Here we represent a category as a 5-tuple, consisting of predicate sets of objects and arrows, functions for obtaining the domain and codomain of arrows, and finally a composition operation. The "informal" classes have been represented by predicates over some HOL type. Though this representation is perhaps too weak in general, we hope that it is strong enough for some category theory. A stronger representation is presented in Section 4.2.

Note that we use the type variables `*o` and `*a` for objects and arrows, respectively. This means that any HOL type can be used to represent objects and arrows. Also note that we use HOL equality to compare arrows. In ALF [4], Coq [7] and LEGO [1] the equality is part of the category structure; so each category has its own equality on arrows.

It is convenient to treat a category as one entity `CC` (corresponding to $\mathcal{C}$) instead of as a 5-tuple. Therefore we shall make use of projections to extract the components of a category. These are called `Obj` (objects), `Arr` (arrows), `Dom` (domain of an arrow), `Cod` (codomain of an arrow), and `Comp` (composition). We can also define a function to obtain the identity arrow for a given object of a category `Id CC X` (using the choice operator).

3.2 The Category of Sets

A first challenge when formalizing category theory is to formalize the category of all sets, called **Set**. The objects of **Set** are the class of all sets and the arrows are the class of all functions between sets. **Set** can be defined in HOL as follows:

```
DEF (Category of all sets):
  Set =
   ({s:*->bool | T},
    {(f,s,t) | map f (s,t)}, FST o SND, SND o SND,
    (\(g,s',t')(f,s,t). (g o f,s,t'))
```

where the notion of map is defined by:

```
DEF (Map between sets):
  map f (s,t) = (image f s) SUBSET t /\ (!x. ~(x::s) ==> (f x = ARB))
```

We have used a type variable `*` to represent the type of elements of sets; sets are represented as predicates of type `*->bool`. Note that arrows are triples where the domain and codomain are specified explicitly.

A HOL function f is called a map from s to t if it sends elements of s to elements of t, and furthermore, if it is determined by its action on s. Determinedness ensures that all partially specified functions always return a fixed arbitrary value outside their domain. In this way, we become able to distinguish partial functions by how they behave on their domain; this notion is needed because HOL extensional equality works on the whole underlying type of the elements of sets.

Informal category theory is based on untyped set theory, so the present formalization in typed set theory does not fully capture the textbook meaning of **Set**; as mentioned above, higher order logic, and in particular its typed set theory, is not expressive enough. A formalization of **Set** in HOL and ST as described in Section 4.2 may be more faithful to informal category theory, but more clever people than myself have told me that even in ZF set theory one needs some inaccesible cardinals to have a strong enough theory. Apparently, the Mizar system provides such a very strong set theory [8].

3.3 Arrows, Morphisms and Hom Sets

Arrows are sometimes called homomorphisms, or simply morphisms. Often it is useful to consider the "set" of arrows between two objects of a category, called the Hom set:

```
DEF (Hom set):
  Homset CC (X,Y) = {f :: (Arr CC) | (Dom CC f = X) /\ (Cod CC f = Y)}
```

In category theory textbooks, the collection of arrows between any two objects is not necessarily a set. If it always is then one says that the category is locally small. We do not have to make this assumption here since it is always satisfied; in fact, the formalization only supports small categories where both the collection of objects and arrows are sets. Note that the Hom set yields an object of the category of all sets.

3.4 Definition of Functor

Jaap von Oosten defines the notion of functor between two categories as follows:

> Given two categories $\mathcal{C}$ and $\mathcal{D}$ a functor $F : \mathcal{C} \to \mathcal{D}$ consists of operations $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$ and $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$, such that for each $f : X \to Y$, $F_1(f) : F_0(X) \to F_0(Y)$ and:
>
> - for $f : X \to Y$ and $g : Y \to Z$, $F_1(g\, f) = F_1(g)\, F_1(f)$;
> - $F_1(\mathrm{id}_X) = \mathrm{id}_{F_0(X)}$ for each $X \in \mathcal{C}_0$.

Again, this can be translated easily to higher order logic:

```
DEF (Functor):
  functor (F_0:*o->**o,F_1:*a->**a) (CC,DD) =
   (!X Y::(Obj CC).
     (!f::(Homset CC(X,Y)).
       (F_1 f) :: Homset DD(F_0 X,F_0 Y))) /\
   (!X Y Z::(Obj CC).
     (!(f::Homset CC(X,Y))(g::Homset CC(Y,Z)).
       F_1(Comp CC g f) = Comp DD(F_1 g)(F_1 f)) /\
   (!X::(Obj CC). F_1(Id CC X) = Id DD(F_0 X))
```

So, a functor is a pair of functions which works on objects and on arrows respectively. Usually, a functor is viewed as one entity `FF` and we shall apply `op0` and `op1` to a functor to obtain each of its two operations.

3.5 The Category of Categories

A second challenge when formalizing category theory is formalizing the category of categories, called **Cat**, whose objects are all categories and whose arrows are functors between categories. Size problems and paradoxes are usually (more or less) ignored in informal category theory, but of course we cannot do that in HOL. The definition of **Cat** is:

```
DEF (Category of all categories):
  Cat =
   ({CC:cat | category CC},
    {(FF,CC,DD) | functor FF(CC,DD)}, FST o SND, SND o SND,
    (\((G_0,G_1),DD,EE)((F_0,F_1),CC,DD'). ((G_0 o F_0,G_1 o F_1),CC,EE)))
```

where the type `cat` abbreviates `(*o->bool)#(*a->bool)#(*a->*o)#(*a->*o)#(*a->*a->*a)`. Note that as part of an arrow we also take the domain and codomain of a functor in order to be able to define the corresponding operations on arrows more easily.

We can prove the quite suspiciously sounding fact that **Cat** is an object of itself:

```
THM: Cat :: (Obj Cat)
```

However, this is not inconsistent since the two occurrences of **Cat** do not have the same type. Thus, in a sense we can use the polymorphism to build a kind of hierarchy of types and categories.

3.6 Commuting Squares

A central notion in category theory is that of commuting squares (or diagrams):

$$\begin{tikzcd}
{} \arrow[r, "f"] \arrow[d, "f'"'] & {} \arrow[d, "g"] \\
{} \arrow[r, "g'"'] & {}
\end{tikzcd}$$

This says that if we start in the upper-left corner then the upper path is the same (arrow) as the lower path: $g\,f = g'\,f'$. Sometimes the domains and codomains of arrows are given explicitly. When this is formalized we have to make sure that the domains and codomains of arrows are right:

```
DEF (Commuting square):
  square CC (f,g) (f',g') =
    f::Arr CC /\ g::Arr CC /\ f'::Arr CC /\ g'::Arr CC /\
    (Dom CC f = Dom CC f') /\ (Cod CC g = Cod CC g') /\
    (Cod CC f = Dom CC g) /\ (Cod CC f' = Dom CC g') /\
    (Comp CC g f = Comp CC g' f')
```

3.7 The Functor Category and Other Constructions

A natural transformation between two functors `FF,GG: CC->DD` is a family of morphisms in `DD`, indexed by objects of `CC`:

```
DEF (Natural transformation):
  nattrans (mu:*o->**a) (FF,GG,CC:cat1,DD:cat2) =
   (!C::Obj CC. mu C :: Homset DD(op0 FF C,op0 GG C))
   (!C C'::Obj CC. !f::Homset CC(C,C').
     square DD(mu C',op1 GG f)(op1 FF f,mu C))
```

In the definition we used the following type abbreviations:

```
  cat1 abbr. (*o->bool)#(*a->bool)#(*a->*o)#(*a->*o)#(*a->*a->*a)
  cat2 abbr. (**o->bool)#(**a->bool)#(**a->**o)#(**a->**o)#(**a->**a->**a)
```

Often the notation `mu: FF=>GG` is used for a natural transformation between functors `FF,GG: CC->DD`.

In the functor category, functors between given categories are objects and natural transformations are arrows:

```
DEF (Functor category):
  FunCat(CC,DD) =
   ({FF | functor FF CC DD},
    {(mu,FF,GG) | nattrans mu(FF,GG,CC,DD)}, FST o SND, SND o SND,
    (\(nu,GG,HH)(mu,FF,GG')C. (Comp DD(nu C)(mu C),FF,HH)))
```

This defines a construction on categories: we can prove that given categories `CC` and `DD` then `FunCat(CC,DD)` is also a category.

Another important construction is the dual category construction, defined by

```
DEF (Dual category):
  DualCat (CC:cat) = (Obj CC,Arr CC,Cod CC,Dom CC,Comp CC)
```

In the dual category the direction of arrows is reversed, so in the definition the domain and codomain of arrows have just been interchanged.

The last construction we shall consider in this paper is the product category, defined by

```
DEF (Product category):
  ProdCat(CC:cat1,DD:cat2) =
   ({(C,D) | C::Obj CC /\ D::Obj DD},
    {(f,g) | f::Arr CC /\ g::Arr DD},
    (\(f,g). (Dom CC f,Dom DD g)),
    (\(f,g). (Cod CC f,Cod DD g))
    (\(f,g)(f',g'). (Comp CC f f',Comp DD g g')))
```

We can define the usual projections π_1 and π_2 for product categories.

3.8 The Hom Functor

The Hom functor is an extension of the Hom set to a functor `Hom CC: (ProdCat(DualCat CC,CC)) -> Set`, sometimes written as `CC(--,--)`:

```
DEF (Hom functor):
  Hom CC =
   ((\(X,Y). Homset CC(X,Y)),
    (\(f,g). let A = Dom CC f and C = Cod CC f and
                 B = Dom CC g and D = Cod CC g
             in ((\h::Homset CC(C,B). Comp CC g(Comp CC h f)),A,D)))
```

The restricted λ-abstraction yields a determined function as it should (for the Hom functor to yield an arrow of **Set**). Note that `f` is an arrow from `C` to `A` in the dual category of `CC` and is therefore an arrow from `A` to `C` in `CC`.

Often the following one-variable special cases of the Hom functor are considered and confused with the two-variable Hom functor. We have a functor `Hom1 CC X: CC -> Set`, sometimes written `CC(X,--)`:

```
DEF (Fixed-left Hom functor):
  Hom1 CC X =
   ((\Y. Homset CC(X,Y)),
    (\f. let Y = Dom CC f and Z = Cod CC f
         in ((\g::Homset CC(X,Y). Comp CC f g),X,Z)))
```

The other one-variable Hom functor is the functor `Hom2 CC Y: (DualCat CC) -> Set`, sometimes written `CC(--,Y)`:

```
DEF (Fixed-right Hom functor):
  Hom2 CC Y =
   ((\X. Homset CC(X,Y)),
    (\f. let X = Dom CC f and Z = Cod CC f
         in ((\g::Homset CC(Z,Y). Comp CC g f),X,Y)))
```

3.9 Comments

- Using predicates to represent collections of objects and arrows we can only formalize small categories.

- The representation of the partial functions that were used as arrows in the category of sets is complicated via a determinedness condition, due to the presence of an underlying type in typed set theory; extensional equality works on all elements of the underlying type rather than only at the predicate subsets.
- We must use triple representations for arrows of most categories in order to make the domain and codomain explicit.
- Some type checking is done automatically, some must be done by theorem proving due to presence of predicates, representing sets.
- Polymorphism supports hierarchies of categories, for instance, the category of categories **Cat** can be made an object of itself by using different instantiations of type variables.

4 Category Theory in Higher Order Logic and Set Theory

In this section, we will try to obtain a more faithful formalization of category theory by exploting an axiomatization of Zermelo-Fraenkel set theory in higher order logic, provided by Mike Gordon [6]. However, while this seems to work at a first sight, problems arise quite quickly. We first describe the axiomatization in Section 4.1, then discuss a reformalization of some category theory concepts in Section 4.2.

4.1 ZF Set Theory in HOL

HOL is extended with set theory by declaring a new type `V` and a new constant '`ZFin`' (set membership) of type `V->V->bool`,[1] and then postulating eight new axioms about `V` and `ZFin`:

```
Axiom of extensionality.
  !s t. (s = t) = (!x. x ZFin s = x ZFin t)

Axiom of empty set.
  ?s. !x. ~(x ZFin s)

Definition of the empty set:
  |- !x. ~x ZFin Empty

Axiom of union.
  !s. ?t. !x. x ZFin t = ?u. x ZFin u /\ u ZFin s

Definition of big union:
  |- !x t. t ZFin (UU x) = (?z. t ZFin z /\ z ZFin x)

Definition of set inclusion.
  |- !s t. Subset s t = !x. x ZFin s ==> x ZFin t

Axiom of power-sets.
  !s. ?t. !x. x ZFin t = x Subset s

Definition of the power set constructor.
```

[1] In the type of the set membership operator, note that elements of sets are themselves sets. Generally speaking, new sets must be constructed from existing sets some way, in principle starting from the empty set and then using axioms.

```
  |- !s x. x ZFin (Pow s) = (!y. y ZFin x ==> y ZFin s)

Axiom of separation.
  !p s. ?t. !x. x ZFin t = x ZFin s /\ p x

Axiom of replacement.
  !f s. ?t. !y. y ZFin t = ?x. x ZFin s /\ (y = f x)

Definition of set intersection.
  |- !s t x. x ZFin (s Intersect t) = x ZFin s /\ x ZFin t

Axiom of foundation.
  !s. ~(s = Empty) ==> ?x. x ZFin s /\ (x Intersect s = Empty)

Definition of image.
   |- !f t y. y ZFin (Image f t) = (?x. x ZFin t /\ (y = f x))

Definition of singleton set.
  |- !x y. y ZFin (Singleton x) = (y = x)

Definition of infixed binary union of sets.
  |- !s t x. x ZFin (s U t) = (x ZFin s) \/ (x ZFin t)

Definition of successor of a set.
  |- !x. Suc x = x U (Singleton x)

Axiom of Infinity.
  ?s. Empty ZFin s /\ !x. x ZFin s ==> Suc x ZFin s
```

Pairs and functions can be defined in ZF set theory. Pairs are defined by

```
DEF: <x,y> = {{x},{x,y}}
```

The cartesian product of two sets can then be defined by:

```
DEF: ZFprod X Y = {<x,y> ZFin Pow(Pow(X U Y)) | x ZFin X /\ y ZFin Y}
```

The usual projection functions `Fst` and `Snd` can be defined on pairs.

A total function is a set of pairs

```
DEF: ZFfun X Y = {f ZFin Pow(ZFprod X Y)) | !x ZFin X. ?!y. <x,y> ZFin f}
```

Universal quantification restricted over ZF sets is available through a parser and pretty-printer hack: `!x ZFin X. t` stands for `!x. x ZFin X ==> t`. Note that a set function, which has type `V`, is different that a logical function, which has type e.g. `*->**`. The domain and range operations on functions are provided:

```
DEF: domain f = Image Fst f
DEF: range f = Image Snd f
```

Total functions can be written using the ZF lambda abstraction:

```
(ZFlam (x ZFin X). t[x]) = {<x,y> ZFin (ZFprod X(Image f X)) | y = t[x]}
```

Set function application is defined by:

```
DEF: f ^^ x = (@y. <x,y> ZFin f)
```

Finally, set function composition is provided through the following definition:

```
DEF:
  comp f g =
  {<x,z> ZFin (ZFprod(domain g)(range f)) | ?y. <x,y> ZFin g /\ <y,z> ZFin f}
```

More details on ZF theory in HOL can be found in [6, 2].

4.2 Formalizing the Category of Sets in ZF

With the axiomatization of the new type `V` our formalization of categories in Section 3.1 suddenly becomes considerably more powerful, because type variables then also range over `V`. This means that ZF classes can be expressed as predicates over `V`, and further, the category of all sets can really consist of the class of all untyped sets and the class of all set functions as arrows:

```
DEF (Category of all sets):
  Set = ({x:V | T},{f:V | ?X Y. f ZFin (ZFfun X Y)},domain,range,comp)
```

In other words, our formalization of category now supports (at least some kind of) large categories.

With this definition of **Set** we must redefine the Hom set function, which must yield objects of **Set**. Let us rename the previous Hom set function to a Hom class function:

```
DEF (General Hom function):
  Homclass CC(X,Y) = {f :: (Arr CC) | (Dom CC f = X) /\ (Cod CC f = Y)}
```

Then a Hom set function into **Set** can be obtained from it by restricting the definition to work in situations where the Hom classes are ZF sets:

```
DEF (Hom set):
  Homset (CC:lcat) (X,Y) = mk_ZFset(Homclass CC(X,Y))
```

The type `lcat` abbreviates the type of large (and locally small) categories:

```
  lcat abbr. (V->bool)#(V->bool)#(V->V)#(V->V)#(V->V->V)
```

The constant `mk_ZFset` is used to convert a ZF class into a ZF set, if this is possible:

```
DEF: mk_ZFset (P:V->bool) = (@s:V. !x:V. x ZFin s = x :: P)
```

It only works if we assume (or prove) categories are locally small:

```
DEF: is_ZFset (P:V->bool) = (?s:V. !x:V. x ZFin s = x :: P)
DEF: locally_small CC = !X Y :: (Obj CC). is_ZFset(Hom CC(X,Y))
```

Hence, also in this respect this formalization of the category of sets is more faithful to textbook presentations.

However, with the above definition of `Homset`, which works when arrows and objects are represented in ZF set theory, it becomes problematic to represent functors and the functor category properly. For compatibility, the objects (and arrows) of the functor category should be sets and therefore functors should be sets. In turn this means that the Hom functor should be represented as a set. Recall from Section 3.8 that given a category `CC` the Hom functor must map pairs of objects of `CC` to sets of arrows and pairs of arrows of `CC` to set functions, which are the arrows of **Set**. These maps from pairs to sets must themselves be sets and would naturally be represented as set functions. But set functions must specify their domain explicitly and these domains must be sets:

```
DEF *BAD* (Hom functor):
  Hom CC =
   <(ZFlam (<X,Y> ZFin (ZFprod(*Obj CC*,*Obj CC*))). Homset CC(X,Y)),
    (ZFlam (<f,g> ZFin (ZFprod(*Arr(DualCat CC)*,*Arr CC*))).
      let A = Dom CC f and C = Cod CC f and
          B = Dom CC g and D = Cod CC g
      in
       <(ZFlam (h ZFin (Homset CC(C,B))). Comp CC g(Comp CC h f)),A,D>)>
```

This only works if the category `CC` is small in the sense that both its objects and its arrows constitute sets and then, to be precise, we should either insert conversion functions above, to convert classes to sets, or use sets instead of classes in the first place to represent the collections of objects and arrows. Neither approach would work generally, because there are categories which simply are not small, e.g. **Set**.

5 Conclusions

From the above and other attempts to use ZF sets for the category of sets we have come to the conclusion that this will always make the definitions of the Hom functor and the functor category problematic, because the formalization forces objects and arrows of categories to be sets rather than classes. Generally speaking this is due to the restriction in ZF that sets must be built from existing sets, starting from the empty set and then using the axioms, to avoid paradoxes. There is no easy way around this (sensible) restriction. For instance, it would not be a good idea to represent categories as the small categories whose collections of objects and arrows are ZF sets. Then it would not be possible to define the category of sets. Making a distinction between small and large categories, where the collections of objects and arrows in large categories are represented using predicates over sets, would solve this problem. However, the Hom functor would only be defined for small categories so it would be impossible to use it with for instance the functor category of functors from some dual category to **Set** (which is done in the Yoneda lemma). Even using conversion functions like `mk_ZFset` introduced above would not be possible, for the simple reason that **Set** cannot be converted to a small category.

It therefore seems that employing ZF set theory at any level of the formalization is problematic, at least when we choose to formalize the concepts of category theory as in this paper. There might be other formulations of category theory than the (standard) one presented in Jaap von Oosten's introduction that are more suitable for formalization. The conclusion must be that the higher order logic formalization presented first is the better one in this paper because it does not immediately give any problems. However, HOL and its typed set theory is a rather weak basis for the formalization of category theory which therefore might not follow textbook presentations precisely (and correctly). Furthermore, as we saw in Section 3, it is possible to use the polymorphism of higher order logic to make this hierarchy explicit. With one polymorphic definition of the notion of category we can make various instantiations, as we may wish. But in category theory the hierarchy of categories is never considered. Finally, an advantage of using set theory rather than higher order logic is that when we use set theory partial functions can be treated more naturally as ordinary set functions between sets. In higher order logic we have the problems of an underlying type and extensional equality (which meant that we had to introduce a determinedness condition on functions to make them proper arrows of **Set**).

References

[1] P. Aczel. Galois: A theory development project. A report on work in progress, for the Turin meeting on the Representation of Mathematics in Logical Frameworks, 1993.

[2] S. Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report No. 354, University of Cambridge Computer Laboratory, November 1994.

[3] S. Agerholm. A comparison of HOL-ST and Isabelle/ZF. Technical Report No. 369, University of Cambridge Computer Laboratory, 1995.

[4] P. Dybjer and V. Gaspes. Implementing a category of sets in alf. Manuscript, 1994.

[5] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[6] M.J.C. Gordon. Merging HOL with set theory: preliminary experiments. Technical Report No. 353, University of Cambridge Computer Laboratory, November 1994.

[7] G. Huet and A. Saïbi. Constructive category theory. Draft, 1995.

[8] Piotr Rudnicki. *An Overview of the MIZAR Project.* Unpublished; but available by anonymous FTP from `menaik.cs.ualberta.ca` in the directory `pub/Mizar/Mizar_Over.tar.Z`, 1992.

[9] J. van Oosten. Basic category theory. Technical Report LS-95-1, BRICS, Department of Computer Science, University of Aarhus, January 1995.