CATS 2004 Preliminary Version

Towards a Readable Formalisation of Category Theory

Greg O'Keefe [1]

Automated Reasoning Group, RSISE
Australian National University
Canberra, Australia

Abstract

We formally develop category theory up to Yoneda's lemma, using Isabelle/HOL/Isar, and survey previous formalisations. By using recently added Isabelle features, we have produced a formal text that more closely approximates informal mathematics.

Key words: formalised mathematics, theorem proving, category theory

1 Introduction

Formalised mathematics is a computer science discipline largely ignored by mathematicians. There are good reasons for this. It takes months to become a competent user of any proof assistant. Once the tool has been mastered, even the most elementary results take days of tedious labour to reproduce. Perhaps worst of all, the resulting formal text is mostly incomprehensible to anyone unfamiliar with the tool in question.

If it were possible to produce formalisations without excessive effort, then mathematicians might do so for increased confidence. If there are errors in a proof, they will almost certainly be detected in the attempt to formalise it.

In order to minimise the effort required, and maximise the value of the result, the formalisation should be as close as possible to its informal origins.

Recent advances in proof assistant technology, implemented in Isabelle [20] have made it possible to reduce the gap between formal and informal work. One of our aims is to demonstrate this. Category theory is an appropriate test, because category theory is "notoriously hard to formalize in any kind of system" [12, Page 20].

[1] Email: `greg.okeefe@anu.edu.au`

We begin with a survey of existing formalisations of category theory. The bulk of the paper is an overview of our own formalisation using Isabelle/HOL/Isar. Our concluding remarks include some plans for further work.

2 Survey

The key features of a category theory formalisation are: language, type system, representation, automation, coverage, and project status.

Proof assistants provide a language for defining concepts, stating theorems and giving proofs. There are two styles of proof language: procedural and declarative. Most proof assistants use the procedural style. A procedural proof consists of instructions, called tactics, showing how to reach the goal theorem by applying the logic's rules.

Note that these are "proofs" only in a broad sense, as they lack two important properties. First, there is generally no proof object about which we can reason meta-theoretically. Secondly, ordinary readers will not be convinced to accept the conclusion by inspecting such a proof: instead one must first believe in the soundness of the proof checker, then execute it.

Declarative proof languages [18] [5] [33] [31] [12] [35] [27] seek to overcome this second defect. They emulate the "mathematical vernacular" [18, §F.3], where we make assumptions and state claims which we justify using previous claims and assumptions.

Proof assistants also come equipped with a type system. For our purposes, the main species are mono-typed, simply typed and dependently typed. "Mono-typed" refers to systems such as first order logic, where there is really no interesting type system. Simply typed systems have atomic types and type constructors which combine types to yield new types. The higher order logic of the HOL prover [29] is simply typed. Dependent type systems have type constructors that also take terms. For example, for each pair of natural numbers (m, n) we have a type of $m \times n$ matrices. Coq [13] and NuPrl [1] have dependent type systems.

A useful survey and comparison of theorem proving assistants can be found in [34].

There are choices to be made when encoding preformal concepts in a formal system. These choices affect how easily the development proceeds and how much effort is required to interpret it. As a simple example, we may represent functors as pairs of functions, or we might leave the action on objects implicit.

Some of the developments we examine have automation of proof as a main objective, whilst others give explicit detailed proofs. Automation is a worthy goal in its own right of course, but it is also important for readable formalisation because lengthy proofs of obvious facts are distracting.

The developments which cover more of the theory deserve greater attention, as this is an indication that they are doing something right. This judgement must be made with caution however: it is possible that good ideas have been unfruitful for practical reasons, or that bad ideas have been overcome by sheer hard work. To compare the amount of theory covered, we will use the table of contents of the standard text [15]. I dub this the "MacLane metric".

Finally, we are interested in whether a formalisation is under active development, runs on the current version of its proof assistant, or is of merely historical interest.

We now examine each of the developments in turn. They are grouped according to the proof assistant used. We give more space to the better developed formalisations.

2.1 Coq and Lego

Peter Aczel led a project, Galois [3], whose aim is "to formalise some abstract algebra in a predicative style". Part of this work was a formalisation of category theory using Lego [2]. This formalisation covers very little theory, and is not being actively developed. However, its novel representation of categories in dependent type theory has been adopted by the most successful formalisation: the Coq [13] development of Saïbi and Huet [26] [25].

A *setoid* is a triple consisting of a type, a relation on the type and a proof that the relation is an equivalence. The universe of the type, factored by the equivalence looks pretty much like a set. The homsets in Aczel's and Saïbi's formalisations are setoids, and the objects are types.

The material from [15, I,II,III] is covered, including Yoneda's lemma. In addition, Freyd's adjoint functor theorem is proved and an example Cartesian Closed Category is given. The current version [24] consists of 105 `.v` files, and is compatible with the current (7.4) version of Coq.

This is a difficult piece of work to understand unless you are familiar with constructive dependent type theory. Its relationship to category theory as usually presented in maths texts is indirect at best. However, it does develop more category theory than the other existing formalisations.

The diploma thesis [8] extends an older version of Saïbi's development, giving an alternative definition of natural transformation, and proving a couple of extra results on adjunctions and developing the theory of cocartesian liftings [6].

2.2 Mizar

There are two distinct developments of category theory in Mizar [5] consisting of about 30 "articles". Most of [15, I,II,III] is covered, but little beyond that. There are no adjunctions or general limits. The developments are part of a large integrated library of formalised mathematics. As a result there are many more examples than in other developments, including categories of groups, rings, and modules, and posets considered as categories.

The main difference between the two Mizar developments is the treatment of arrows. One uses the usual definition where a category has a collection of arrows, and has source and target functions from arrows to objects. Homsets are then defined in terms of these. The other development follows the alternative definition in [15, §1.8], where the function *hom* from pairs of objects to sets of arrows is part of the category definition.

The Mizar library is under active development. The most recent Mizar articles on category theory are from 2001. The Mizar library is maintained to keep it compatible with the current version of the proof checker.

Most computer systems for formalised mathematics use a typed λ-calculus as their foundation. Mizar is the exception. Its logic is classical and first order. Its axioms are those of Tarski-Grothendieck set theory, which is "basically the Zermelo-Frankel set theory with the axiom of choice replaced by Tarski's stronger axiom [28] of the existence of arbitrarily large, strongly inaccessible cardinals." [22].

Another difference is the style of the proofs. Mizar along with Automath [18] originated the recently popular notion of "declarative" proof. It is possible to follow the reasoning in a Mizar proof.

Mizar does very little automation: proofs must be given to it in full detail. As a result, Mizar formalisations are very long. The article which proves Yoneda's lemma is 1730 lines long, and depends on at least 8 category theory related articles of similar size, as well as more basic ones. Our own formalisation below consists of 1208 lines, and is in our view, too long for its content. Long proofs of obvious facts do not help the reader to understand the reasoning.

Another disadvantage of Mizar is the lack of introductory documentation. Also, the source code for the system is not openly available. This seems inappropriate for such foundational work, where everything must be doubted and examined, nothing taken on trust.

2.3 NuPrl

In 1990, Altucher and Panangaden published "A Mechanically Assisted Constructive Proof in Category Theory" [4]. I seems that no further work has been done on this NuPrl [1] development. The focus was automation of proof, and extraction of programs from those proofs. This ability is a key feature of NuPrl. Category theory was seen a promising subject matter because of its connections with type theory and functional programming, and because the routine "diagram chasing" proofs seem amenable to automation.

The source files for this development seem to be unavailable. It would be unlikely to work with the current version of the system in any case. The exposition in [4] is very limited. It is claimed that definitions were given for subobjects, limits, adjunctions, Cartesian closed categories and triples. The development is focused on proving the adjoint functor theorem [15, V.6, Theorem 2].

A much more recent development in NuPrl is that of Tjark Weber presented in [30]. This masters thesis is concerned with program transformations "anamorphisms" and "catamorphisms", which are defined in terms of category theory. After defining category and functor, algebras and coalgebras and homomorphisms are introduced. These notions are then used to define the program transformation concepts.

2.4 Isabelle and HOL

Lockwood Morris has used HOL [29] to formalise most of [15, I,II,III], including Yoneda's lemma. There are no publications yet, nor has there been a public distribution of the HOL code. My sources are an unpublished manuscript [17] and email correspondence with the primary author.

Like the older NuPrl effort [4], the focus here is on automation. Tactics have been written to automate the most common proof tasks, such as showing that a given term denotes an arrow or object of the category under consideration, and breaking complex predicates into simple subgoals.

Whilst Coq and NuPrl have dependent types, HOL and Isabelle have only simple types. This makes category theory subject to annoying koans like "what is the composite of uncomposable arrows?" and "what is the component of a natural-transformation at a non-object?" These do not arise in dependent type systems, because the relevant terms and formulae are ill typed.

Johan Glimming's 2001 masters thesis "Logic and Automation for Algebra of Programming" [10] contains a development of some elementary Category Theory in Isabelle/HOL/Isar [20] [31]. Most of chapter I and parts of Chapter II of [15] are covered. The proofs are in procedural style, using Isar's "tactic emulation".

2.5 Others

The work described in "Computational Category Theory" [23] is not a formalisation of category theory, since there is no means to prove anything. It consists of ML structures and functions that represent the main ideas of category theory. For example, like many formalisations, a category over a pair of types is a tuple of four functions: source target identity and composition. The difference is that there are no axioms. Instances of the ML structure `Cat` can have $Dom(1_A) \neq A$ for example.

Even so, the structures defined could be used in a formal development. Furthermore, the code could perhaps form the basis of a categorical "logical framework" [21].

Definitions for most ideas from [15, I,II,III] are given, plus toposes.

The final chapter, "Formal Systems for Category Theory" of [23] describes two formalisations.

The first, due to Joseph Goguen uses his pioneering algebraic specification language OBJ [11]. Only the most elementary notions are defined. As OBJ is "algebraic", the only formulae are equations, and the only inference rule is equational rewriting. There are no explicit proofs, only automated rewriting. This would surely limit how far the theory could go.

The chapter also summarises Roy Dyckhoff's [9] encoding of category theory using the Göteborg Type Theory System. Category theory is directly represented as a type system, with atomic types such as Cat and Func, and rules such as

$$\frac{\mathcal{A} : \mathsf{Cat} \quad \mathcal{B} : \mathsf{Cat}}{\mathsf{Func}(\mathcal{A}, \mathcal{B}) : \mathsf{type}}$$

This is a novel and attractive approach to category theory and foundations, worthy of further investigation.

We also note the survey on formalised category theory of Takahisi Mohri [16].

3 Development

The full text of the formal development, produced automatically by Isabelle, is 27 pages long. It is available on the internet [19], along with the Isabelle/HOL source files. In what follows we give most of the definitions, some of the results and a couple of proofs.

We have tried to keep the representation of categories in this formal setting as close as possible to the usual descriptions in mathematical texts.

A category is defined as a sextuple. The category record type takes two type parameters, one each for the arrow and object types. Isabelle's record facility allows us to name the components, and to use these names as projections. We also define abbreviated forms (beginning with upper case letters) for use in locales, where the category can be omitted or specified by a numeric subscript.

```
record ('o, 'a) category =
  ob :: 'o set (Obı  70)
  ar :: 'a set (Arı  70)
  dom :: 'a ⇒ 'o (Domı - [81] 70)
  cod :: 'a ⇒ 'o (Codı - [81] 70)
  id :: 'o ⇒ 'a (Idı - [81] 80)
  comp :: 'a ⇒ 'a ⇒ 'a  (infixl ·ı 60)
```

Between parentheses at the end of each line are syntax annotations. For example the first one (Ob_1 70) is like saying "We write Ob_1 or simply *Ob* for *ob* $\mathcal{C}$ when the context makes it clear that it is $\mathcal{C}$ we are referring to." It might be better if this syntax could be given separately.

Homsets are defined in the obvious way.

constdefs
  $hom :: [('o,'a,'m)\ category\text{-}scheme,\ 'o,\ 'o] \Rightarrow\ 'a\ set\ (Hom\imath\ \text{-}\ \text{-})$
  $hom\ \mathcal{C}\ A\ B \equiv \{\ f.\ f{\in}ar\ \mathcal{C}\ \&\ dom\ \mathcal{C}\ f\ =\ A\ \&\ cod\ \mathcal{C}\ f\ =\ B\ \}$

The corresponding definition in MacLane [15, page 27] is

$$hom_C(a,b) = \{f \mid f \text{ is an arrow } f : a \longrightarrow b \text{ in } C\}$$

A *locale* [14] is a named collection of fixed arbitrary objects, assumptions and definitions. It saves a lot of repetition, and emulates informal mathematical practice. Here, a fixed object $\mathcal{C}$ is assumed to satisfy the given rules. We have written the assumptions as rules (using meta-connectives $\Longrightarrow$) rather than axioms (using object-connectives $\longrightarrow$).

The locale also yields a predicate, so that we can write *category* X to assert that X satisfies the assumptions in the locale.

The arrows in the *comp-types* assumption denote function sets. When applied to a member of $Hom\ B\ C$, the composition operation returns a function that, when applied to a member of $Hom\ A\ B$, yields a member of $Hom\ A\ C$.

locale *category* = *struct* $\mathcal{C}$ +
  assumes *dom-object* [*intro*]:
  $f\ \in\ Ar \Longrightarrow Dom\ f\ \in\ Ob$
  and *cod-object* [*intro*]:
  $f\ \in\ Ar \Longrightarrow Cod\ f\ \in\ Ob$
  and *id-left* [*simp*]:
  $f\ \in\ Ar \Longrightarrow Id\ (Cod\ f) \cdot f = f$
  and *id-right* [*simp*]:
  $f\ \in\ Ar \Longrightarrow f\ \cdot\ Id\ (Dom\ f) = f$
  and *id-hom* [*intro*]:
  $A\ \in\ Ob \Longrightarrow Id\ A\ \in\ Hom\ A\ A$
  and *comp-types* [*intro*]:
  $\bigwedge A\ B\ C.\ (comp\ \mathcal{C}) : (Hom\ B\ C) \rightarrow (Hom\ A\ B) \rightarrow (Hom\ A\ C)$
  and *comp-associative* [*simp*]:
  $f\ \in\ Ar \Longrightarrow g\ \in\ Ar \Longrightarrow h\ \in\ Ar$
  $\Longrightarrow Cod\ h\ =\ Dom\ g \Longrightarrow Cod\ g\ =\ Dom\ f$
  $\Longrightarrow f\ \cdot\ (g\ \cdot\ h)\ =\ (f\ \cdot\ g)\ \cdot\ h$

Although this associativity rule is fairly clear, we hope in future work to get closer to statements such as MacLane's [15, page 7]

Associativity. For given objects and arrows in the configuration

$$a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{k} d$$

one always has the equality

$$k \circ (g \circ f) = (k \circ g) \circ f$$

It may even be possible to admit arbitrary diagrams as formulae. Then the diagram above could be the antecedent and the equation the consequent of a conditional formula.

We skip the few trivial lemmas about categories, and move on to the definition of the category of sets. Rather than assuming that there is one universe that contains every set, we define the category of subsets of a given universe set U. MacLane [15, page 11] calls these categories Ens_U or just Ens. The type of this set is a parameter determining the type of the category.

We define the arrows of set categories to be triples, to make the domain and codomain explicit.

```
record 'c set-arrow =
  set-dom :: 'c set
  set-func :: 'c ⇒ 'c
  set-cod :: 'c set
```

Not every instance of the record type *set-arrow* is a legitimate arrow. We define a predicate, which states that the domain and codomain are in the universe, that the function takes the domain into the codomain and that the function is "extensional" on the domain. This means that outside that domain, the function takes the value "*arbitrary*". Without this, the arrows $\mathbb{N} \xrightarrow{id} \mathbb{N}$ and $\mathbb{N} \xrightarrow{abs} \mathbb{N}$ would be distinct, which is not what we want. The theory of functions with restricted domains is developed in the Isabelle/HOL theory `src/HOL/Library/FuncSet.thy` in the Isabelle 2003 distribution [20].

Note that we introduce a new symbol $\odot$ for set arrow composition. Once we have established that our set categories are indeed categories, we will be able reason about them within category locales using $\cdot$ for arrow composition.

```
constdefs
  set-arrow :: ['c set, 'c set-arrow] ⇒ bool
  set-arrow U f ≡ set-dom f ⊆ U & set-cod f ⊆ U
  & (set-func f): (set-dom f) → (set-cod f)
  & set-func f ∈ extensional (set-dom f)
  set-id :: ['c set, 'c set] ⇒ 'c set-arrow
  set-id U ≡ λs∈Pow U. (|set-dom=s, set-func=λx∈s. x, set-cod=s|)
  set-comp :: ['c set-arrow, 'c set-arrow] ⇒ 'c set-arrow (infix ⊙ 70)
  set-comp g f ≡
  (|
    set-dom = set-dom f,
    set-func = compose (set-dom f) (set-func g) (set-func f),
    set-cod = set-cod g
  |)
  set-cat :: 'c set ⇒ ('c set, 'c set-arrow) category
  set-cat U ≡
  (|
    ob = Pow U,
    ar = {f. set-arrow U f},
    dom = set-dom,
    cod = set-cod,
    id = set-id U,
    comp = set-comp
  ⦈
```

We must now show that this structure satisfies the category axioms. For example:

lemma *set-id-left*:
  assumes $f \in ar\ (set\text{-}cat\ U)$
  shows $set\text{-}id\ U\ (set\text{-}cod\, f) \odot f = f$

and

lemma *set-id-hom*:
  assumes $A \in ob\ (set\text{-}cat\ U)$
  shows $id\ (set\text{-}cat\ U)\ A \in hom\ (set\text{-}cat\ U)\ A\ A$

and eventually

theorem *set-cat-cat*:
  category (*set-cat U*)

Functors are pairs of functions. The present development does not include functor categories, but to do so it would be necessary to combine these pairs with their domain and codomain categories.

record $('o1,'a1,'o2,'a2)$ *functor* =
  $om :: 'o1 \Rightarrow 'o2$
  $am :: 'a1 \Rightarrow 'a2$

We would like to write FA and Ff, as is the usual informal notation. Although we could define a particular F to be overloaded in this way, we can not define functors as functions that take objects to objects and arrows to arrows. Attempting to do so yields an angry message from the type inference module. Instead we define an alternative explicit notation.

Also note that for any record type *whatever*, there is a type called *whatever-scheme* for the extensions of that record type.

syntax
  $\text{-}om :: ('o1,'a1,'o2,'a2,'m)functor\text{-}scheme \Rightarrow 'o1 \Rightarrow 'o2\ (\text{-}\ _{\mathrm{o}}\ [81])$
  $\text{-}am :: ('o1,'a1,'o2,'a2,'m)functor\text{-}scheme \Rightarrow 'o1 \Rightarrow 'o2\ (\text{-}\ _{\mathrm{a}}\ [81])$
translations
  $F_{\mathrm{o}} \rightleftharpoons om\ F$
  $F_{\mathrm{a}} \rightleftharpoons am\ F$

We do things a little differently with functors. The properties of functors are defined as predicates in a locale containing two categories. The locale saves us two arguments on each predicate. Defining and naming the predicates allows us to use them in our reasoning.

Note that we assert the categories to be equal to themselves. This is a kludge to align the types with those of the predicates.

locale *two-cats* = *category* $\mathcal{A}$ + *category* $\mathcal{B}$ +
  assumes $\mathcal{A} = (\mathcal{A} :: (\mathit{'o1}, \mathit{'a1}, \mathit{'m1})\mathit{category\text{-}scheme})$
  assumes $\mathcal{B} = (\mathcal{B} :: (\mathit{'o2}, \mathit{'a2}, \mathit{'m2})\mathit{category\text{-}scheme})$
  fixes *preserves-dom* :: $(\mathit{'o1}, \mathit{'a1}, \mathit{'o2}, \mathit{'a2})\mathit{functor} \Rightarrow \mathit{bool}$
  and *preserves-cod* :: $(\mathit{'o1}, \mathit{'a1}, \mathit{'o2}, \mathit{'a2})\mathit{functor} \Rightarrow \mathit{bool}$
  and *preserves-id* :: $(\mathit{'o1}, \mathit{'a1}, \mathit{'o2}, \mathit{'a2})\mathit{functor} \Rightarrow \mathit{bool}$
  and *preserves-comp* :: $(\mathit{'o1}, \mathit{'a1}, \mathit{'o2}, \mathit{'a2})\mathit{functor} \Rightarrow \mathit{bool}$
  defines *preserves-dom* $G \equiv$
  $\forall f \in Ar_1.\ G_{\mathrm{o}}\ (Dom_1\ f) = Dom_2\ (G_{\mathrm{a}}\ f)$
  and *preserves-cod* $G \equiv$
  $\forall f \in Ar_1.\ G_{\mathrm{o}}\ (Cod_1\ f) = Cod_2\ (G_{\mathrm{a}}\ f)$
  and *preserves-id* $G \equiv$
  $\forall A \in Ob_1.\ G_{\mathrm{a}}\ (Id_1\ A) = Id_2\ (G_{\mathrm{o}}\ A)$
  and *preserves-comp* $G \equiv$
  $\forall f \in Ar_1.\ \forall g \in Ar_1.\ Cod_1\ f = Dom_1\ g \longrightarrow G_{\mathrm{a}}\ (g \cdot_1 f) = (G_{\mathrm{a}}\ g) \cdot_2 (G_{\mathrm{a}}\ f)$

The locale *functor*, used in the next lemma, adds a structure F, and asserts these properties of it.

Here is an example of a simple result with its proof. Some assertions are given numbers as names, so that they can be referred to later. To show that we have a homset element, we must show that it is an arrow, and has the correct domain and codomain.

lemma (**in** *functor*) *functors-preserve-homsets*:
  assumes *1*: $A \in Ob_1$
  and *2*: $B \in Ob_1$
  and *3*: $f \in Hom_1\ A\ B$
  shows $F_{\mathrm{a}}\ f \in Hom_2\ (F_{\mathrm{o}}\ A)\ (F_{\mathrm{o}}\ B)$
proof−
  from *3*
  have *4*: $f \in Ar$
    by (*simp add*: *hom-def*)
  with *F-preserves-arrows*
  have *5*: $F_{\mathrm{a}}\ f \in Ar_2$
    by (*rule funcset-mem*)
  from *4* **and** *F-preserves-dom*
  have $Dom_2\ (F_{\mathrm{a}}\ f) = F_{\mathrm{o}}\ (Dom_1\ f)$
    by (*simp add*: *preserves-dom-def*)
  also from *3* **have** $\ldots = F_{\mathrm{o}}\ A$
    by (*simp add*: *hom-def*)
  finally have *6*: $Dom_2\ (F_{\mathrm{a}}\ f) = F_{\mathrm{o}}\ A$ **.**
  from *4* **and** *F-preserves-cod*
  have $Cod_2\ (F_{\mathrm{a}}\ f) = F_{\mathrm{o}}\ (Cod_1\ f)$
    by (*simp add*: *preserves-cod-def*)
  also from *3* **have** $\ldots = F_{\mathrm{o}}\ B$

```
    by (simp add: hom-def)
  finally have 7: Cod₂ (F_a f) = F_o B .
  from 5 and 6 and 7
  show ?thesis
    by (simp add: hom-def)
qed
```

To check that we have defined what we think we have defined, it is prudent to construct an obvious instance for each concept, and to prove that is is an instance. We show that the pair containing the object and arrow identity functions is a functor. We define a function which gives us this pair for a given category, and a locale in which to prove our result.

```
constdefs
  id-func :: ('o,'a,'m) category-scheme ⇒ ('o,'a,'o,'a) functor
  id-func 𝒞 ≡ ⦇om=(λA∈ob 𝒞. A), am=(λf∈ar 𝒞. f)⦈

locale one-cat = two-cats +
  assumes endo: ℬ = 𝒜
```

After a few lemmas we are able to show

```
theorem (in one-cat) id-func-functor:
  Functor (id-func 𝒜) : 𝒜 ⟶ 𝒜
```

Again, we recycle the two-cats locale, setting the second category to be the category of sets whose type is that of the arrows of the first category. Homfunctors are defined, and we use the usual notation $Hom(A, -)$. Note that the brackets are an essential part of this notation. Homsets are written without brackets, because they are applied curried functions.

```
locale into-set = two-cats +
  assumes 𝒜 = (𝒜::('o,'a,'m)category-scheme)
  fixes U and Set
  defines U ≡ (UNIV::'a set)
  defines Set ≡ set-cat U
  assumes ℬ-Set: ℬ = Set
  fixes homf :: 'o ⇒ (⦇om::'o⇒('a set), am::'a⇒('a set-arrow)⦈ (Hom'(-,'-'))
  defines homf A ≡ ⦇
  om = (λB∈Ob. Hom A B),
  am = (λf∈Ar. ⦇set-dom=Hom A (Dom f),
  set-func=(λg∈Hom A (Dom f). f • g),
  set-cod=Hom A (Cod f)⦈)
  ⦈
```

The aim is to show that homfunctors are functors. The following is among the required lemmas.

```
lemma (in into-set) homf-preserves-dom:
  assumes f ∈ Ar
  shows Hom(A,-)_o (Dom f) = dom Set (Hom(A,-)_a f)
proof-
  have Dom f ∈ Ob ..
  hence 1: Hom(A,-)_o (Dom f) = Hom A (Dom f)
    by (simp! add: homf-def)
  have 2: dom Set (Hom(A,-)_a f) = Hom A (Dom f)
    by (simp! add: homf-def)
  from 1 and 2 show ?thesis by simp
qed
```

Eventually, we can show

```
theorem (in into-set) homf-into-set:
  Functor Hom(A,-) : 𝒜 ⟶ Set
```

We define natural transformations, give a reasonable notation for them and show that the identity arrow function of a category, when restricted to the objects of that category is a natural transformation from the identity functor to itself.

```
theory NatTrans = Functors:

locale natural-transformation = two-cats + var F + var G + var u +
  assumes Functor F : 𝒜 ⟶ ℬ
  and Functor G : 𝒜 ⟶ ℬ
  and u : ob 𝒜 → ar ℬ
  and u ∈ extensional (ob 𝒜)
  and ∀A∈Ob. u A ∈ Hom_2 (F_o A) (G_o A)
  and ∀A∈Ob. ∀B∈Ob. ∀f∈Hom A B. (G_a f) •_2 (u A) = (u B) •_2 (F_a f)
```

This last line asserts that the following diagram commutes

$$\begin{tikzcd}
F_o A \arrow[r, "u_A"] \arrow[d, "F_a f"'] & G_o A \arrow[d, "G_a f"] \\
F_o B \arrow[r, "u_B"'] & G_o B
\end{tikzcd}$$

The natural transformation locale has given us a predicate whose arguments are two categories, two parallel functors between them and the natural transformation. Rather than use the default syntax

$$\text{natural-transformation}\ \mathcal{A}\ \mathcal{B}\ F\ G\ u$$

we have defined the more familiar form

$$u : F \Rightarrow G \text{ in } Func(\mathcal{A}, \mathcal{B})$$

Note that the $Func(\mathcal{A}, \mathcal{B})$ part of the expression does not really denote a functor category. This notation is used to state a “sanity check” result

```
theorem (in endoNT) id-restrict-natural:
  (λA∈Ob. Id A) : (id-func 𝒜) ⇒ (id-func 𝒜) in Func(𝒜,𝒜)
```

We combine locales to get an environment with an arbitrary category, the corresponding category of sets, and a functor between them. Here, we define the “sandwich” function σ, which will be the witness for Yoneda's lemma, and what will be shown to be its inverse $\sigma^{\leftarrow}$, which we call “unsandwich”. Again, we help ourselves to notational sugar.

```
locale Yoneda = functor + into-set +
  assumes 𝒜 = (𝒜::('o,'a,'m)category-scheme)
  fixes sandwich :: ['o,'a,'o] ⇒ 'a set-arrow  (σ'(-,-'))
  defines sandwich A a ≡ (λB∈Ob. (|
  set-dom=Hom A B,
  set-func=(λf∈Hom A B. set-func (F_a f) a),
  set-cod=F_o B
  |))
  fixes unsandwich :: ['o,'o ⇒ 'a set-arrow] ⇒ 'a (σ←'(-,-'))
  defines unsandwich A u ≡ set-func (u A) (Id A)
```

It is necessary to show that sandwich yields natural transformations, so a few results like the following are needed.

```
lemma (in Yoneda) sandwich-funcset:
  assumes A ∈ Ob
  and a ∈ F_o A
  shows  σ(A,a) : Ob → ar Set
```

Then we get

```
lemma (in Yoneda) sandwich-natural:
  assumes A ∈ Ob
  and a ∈ F_o A
  shows  σ(A,a) : Hom(A,-) ⇒ F in Func(𝒜,Set)
```

We show that the two functions are inverses

```
lemma (in Yoneda) unsandwich-left-inverse:
  assumes 1: A ∈ Ob
  and 2: a ∈ F_o A
  shows σ←(A,σ(A,a)) = a

lemma (in Yoneda) unsandwich-right-inverse:
  assumes 1: A ∈ Ob
  and 2: u : Hom(A,-) ⇒ F in Func(𝒜,Set)
  shows σ(A,σ←(A,u)) = u
```

Now we have the results we need to prove our goal, but stating it requires a couple of minor definitions.

In order to state the lemma, we must rectify a curious omission from the Isabelle/HOL library. It defines the idea of injectivity on a given set, but surjectivity is only defined relative to the entire universe of the target type.

```
constdefs
  surj-on :: ['a ⇒ 'b, 'a set, 'b set] ⇒ bool
  surj-on f A B ≡ ∀ y∈B. ∃ x∈A. f(x)=y
  bij-on :: ['a ⇒ 'b, 'a set, 'b set] ⇒ bool
  bij-on f A B ≡ inj-on f A & surj-on f A B
  equinumerous :: ['a set, 'b set] ⇒ bool (infix ≅ 40)
  equinumerous A B ≡ ∃ f. bij-on f A B
```

MacLane [15, page 61] states the lemma as follows

Lemma 3.1 *(Yoneda) If $K : D \longrightarrow$ **Set** is a functor from D and r an object in D (for D a category with small hom-sets), there is a bijection*

$$y : Nat(D(r, \_), K) \cong Kr$$

Yoneda may have called his result a lemma, but after so many inconsequential propositions bearing that title, an upgrade seems to be in order.

```
theorem (in Yoneda) Yoneda:
  assumes 1: A ∈ Ob
  shows F_o A ≅ {u. u : Hom(A,-) ⇒ F in Func(𝒜,Set)}
apply (unfold equinumerous-def bij-on-def surj-on-def inj-on-def)
apply (intro exI conjI bexI ballI impI)
proof−
  — Sandwich is injective
  fix x and y
  assume 2: x ∈ F_o A and 3: y ∈ F_o A
  and 4: σ(A,x) = σ(A,y)
  hence σ←(A,σ(A,x)) = σ←(A,σ(A,y))
    by simp
  with unsandwich-left-inverse
  show x = y
    by (simp add: 1 2 3)
next
  — Sandwich covers F A
  fix u
  assume u ∈ {y. y : Hom(A,-) ⇒ F in Func (𝒜,Set)}
  hence 2: u : Hom(A,-) ⇒ F in Func (𝒜,Set)
    by simp
  with 1 show σ(A,σ←(A,u)) = u
    by (rule unsandwich-right-inverse)
```

— Sandwich is into F A
from *1* **and** *2*
have $u\ A \in hom\ Set\ (Hom\ A\ A)\ (F_{o}\ A)$
by (*simp add*: *natural-transformation-def natural-transformation-axioms-def homf-def*)
hence $u\ A \in ar\ Set$ **and** $dom\ Set\ (u\ A) = Hom\ A\ A$ **and** $cod\ Set\ (u\ A) = F_{o}\ A$
by (*simp-all add*: *hom-def*)
hence *uAfuncset*: $set\text{-}func\ (u\ A) : (Hom\ A\ A) \to (F_{o}\ A)$
by (*simp add*: *Set-def set-cat-def set-arrow-def*)
have $Id\ A \in Hom\ A\ A$ **..**
with *uAfuncset*
show $\sigma^{\leftarrow}(A,u) \in F_{o}\ A$
by (*simp add*: *unsandwich-def*, *rule funcset-mem*)
qed

end

4 Conclusions

The Isar proofs we have shown are cumbersome when compared to informal text-book proofs. Much of the reasoning is there for the benefit of the prover, not the reader. It is likely that better results can be obtained by more skillful application of the existing automation. There are probably lessons to be learned from a careful examination of the other formalisations. It is not our aim to automate everything though, just the steps that we expect the human reader to make for herself.

The formal language is still a burden, despite the improvements offered by Isar, locales and syntactic extensions. In particular, we wish to reuse or "overload" vocabulary for distinct but analogous notions. For example, in part of the formalisation not presented here, subcategories are defined. The obvious notation is $\mathcal{A} \subseteq \mathcal{B}$, but this led to statements being misinterpreted. Similarly, we used the $\cdot$ symbol, rather than the usual $\circ$ for arrow composition, because the latter is defined as function composition.

We plan to investigate ways of acheiving the required overloading, as it seems essential to the usual mathematical presentation. Denser proof through improved automation automation is another high priority. Finally, to address the problem of low productivity we plan to establish a disciplined process for producing formalisations, and to systematically improve it.

Despite the shortcomings we have noted, the Isar proof language, locales and flexible syntax of Isabelle have bought formalised mathematics a step closer to being accessible to non-specialists.

Thanks to Michael Norrish, Raj Góre, Jeremy Dawson, Amnon Neeman, Marieke Huisman, Tjark Weber, Lockwood Morris and Roy Dyckhoff for clues and helpful emails. Thanks also to the referees for their insightful remarks.

References

[1] *Cornell Prl automated reasoning project*, http://www.cs.cornell.edu/Info/Projects/NuPrl.

[2] *The LEGO homepage* (1999), http://www.dcs.ed.ac.uk/home/lego.

[3] Aczel, P., *Galois: A theory development project*, Technical report, University of Manchester (1993), http://www.cs.man.ac.uk/$\sim$petera/papers.html.

[4] Altucher, J. A. and P. Panangaden, *A mechanically assisted constructive proof in category theory*, in: *CADE-10*, number 449 in Lecture Notes in Artificial Intelligence (1990).

[5] Association of Mizar Users, *Mizar home page*, http://www.mizar.org.

[6] Barr, M. and C. Wells, "Category Theory for Computing Science," Prentice-Hall, 1990.

[7] Bertot, Y., G. Dowek, A. Hirschowitz, C. Paulin and L. Théry, editors, "Theorem Proving in Higher Order Logics," Lecture Notes in Computer Science **1690**, Springer, http://link.springer.de/link/service/series/0558/tocs/t1690.htm, 1999.

[8] Carvalho, A., *Category theory in Coq*, Technical report, Instituto Superior Técnico, 1049-001 Lisboa, Portugal (1998), http://www.cs.math.ist.utl.pt/ftp/pub/CarvalhoA/+README.html.

[9] Dyckhoff, R., *Category theory as an extension of Martin Löf type theory*, Technical report, Department of Computer Science, University of St. Andrews (1985).

[10] Glimming, J., "Logic and Automation for Algebra of Programming," Master's thesis, University of Oxford (2001), http://www.nada.kth.se/$\sim$glimming/publications.shtml.

[11] Goguen, J., *OBJ family*, http://www.cs.ucsd.edu/users/goguen/sys/obj.html.

[12] Harrison, J., *Formalized mathematics*, Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland (1996), http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html.

[13] INRIA, *The Coq proof assistant*, http://coq.inria.fr.

[14] Kammüller, F., M. Wenzel and L. C. Paulson, *Locales: A sectioning concept for Isabelle*, in: *Theorem Proving in Higher Order Logics*, 1999, [7].

[15] MacLane, S., "Categories for the Working Mathematician," Number 5 in Graduate Texts in Mathematics, Springer, 1971, 1998, 2nd edition.

[16] Mohri, T., "On Formalization of Category Theory," Master's thesis, University of Tokyo (1995), http://www-unix.mcs.anl.gov/qed/mail-archive/volume-3/0134.html.

[17] Morris, L. et al., *Interim partial description of a representation for categories in HOL* (1998), lockwood@top.cis.cyr.edu.

[18] Nederpelt, R., J. Geuvers and R. de Vrijer, editors, "Selected Papers on Automath," Elsevier, 1994.

[19] O'Keefe, G., *Some writings*, http://axiom.anu.edu.au/∼okeefe/work.

[20] Paulson, L. and T. Nipkow, *Isabelle*, http://isabelle.in.tum.de.

[21] Pfenning, F., *Logical frameworks*, , **II**, Elsevier Science, 2001 pp. 1063–1147.

[22] Rudnicki, P., *On equivalents of well-foundedness*, Journal of Automated Reasoning **23** (1999), pp. 197–234.

[23] Rydeheard, D. and R. Burstall, "Computational Category Theory," Prentice Hall, 1988.

[24] Saibi, A., *Constructive category theory*, http://coq.inria.fr/contribs/category.html.

[25] Saïbi, A., "Algèbre Constructive en Théorie des Types, Outils génériques pour la modélisation et la démonstration, Application à la théorie des Catégories," Ph.D. thesis, Université Paris VI (1998).

[26] Saïbi, A. and G. Huet, *Constructive category theory*, in: *CLICS-TYPES Workshop on Categories and Type Theory*, http://pauillac.inria.fr/$\sim$essaibi/publi1.html, 1995.

[27] Syme, D., *Three tactic theorem proving*, in: *Theorem Proving in Higher Order Logics*, 1999, [7].

[28] Tarski, A., *On well-ordered subsets of any set*, Fundamenta Mathematicae **32** (1939), pp. 176–183.

[29] University of Cambridge, *Automated reasoning group HOL page*, http://www.cl.cam.ac.uk/Research/HVG/HOL.

[30] Weber, T., "Program Transformations in Nuprl," Master's thesis, University of Wyoming, Laramie, WY (2002).

[31] Wenzel, M., *Isar - a generic interpretative approach to readable formal proof documents*, in: *Theorem Proving in Higher Order Logics*, [7].

[32] Wiedijk, F., http://www.cs.kun.nl/$\sim$freek.

[33] Wiedijk, F., *Mizar light for HOL light*, in: R. Boulton and P. Jackson, editors, *Theorem Proving in Higher Order Logics*, number 2152 in LNCS (2001), pp. 378–393, [32].

[34] Wiedijk, F., *The fifteen provers of the world*, Technical report, University of Nijmegen (2003), [32].

[35] Zammit, V., *On the implementation of an extensible declarative proof language*, in: *Theorem Proving in Higher Order Logics*, 1999, [7].