'**Sets**' Revisited: Working with a Large Category in Isabelle/HOL

Eugene W. Stark

February 10, 2026

Abstract

We revisit the problem of formalization of the category of sets and functions in Isabelle/HOL, regarding it as a paradigm for the formalization of other large categories. We follow a general plan in which we extend the "category" locale from our previous article [3] with a few axioms that allow us to pass back and forth between objects and arrows internal to the category and "real" sets and functions external to it. Using this setup, we prove the standard properties of the category of sets as consequences of the properties of the external notions. A key feature is the inclusion of an axiom that allows us to obtain objects internal to the category corresponding to externally given sets. To avoid inconsistency, our framework axiomatizes a notion of "smallness" and only asserts the existence of objects corresponding to small sets. We give two "top-level" interpretations of our "sets category" locale. One uses "finite" as the notion of smallness and uses only standard HOL for its construction, which results in a small category. The other uses the axiomatic extension of HOL given in [2] to construct an interpretation that incorporates infinite sets as well, resulting in a large (but locally small) category.

Contents

Chapter 1

Introduction

In a previous article [3] we formalized many basic notions and facts from category theory. The formalization was carried out in HOL, in spite of the fact that HOL is significantly weaker than set theories usually cited as foundations for category theory. The rationale for doing so was that most of the central concepts in category theory have significant content, even in contexts, such as small categories, that pose no foundational issues. At some point, however, one wants to be able to work with categories that are not small; the category of sets being the prototypical example. That is, we would like to have a category S that first of all can be considered as a "set category", in the sense that there is fully faithful functorial way of mapping its objects to sets and its arrows to functions, and which in addition has "enough objects" in the sense that if we given any "real" set then there will exist a representative object of S whose elements correspond bijectively to the elements of the given set. Such a category would enjoy the small completeness and cocompleteness properties we would expect of the "real" category of sets.

Now, in standard HOL it is not possible to define a category of sets as described above, because the normal axioms of HOL do not prove the existence of a type "large enough" to provide (even up to equipollence) sets to represent the result of iterated exponentiations starting from an infinite set. However, it is possible to get around this restriction by adding additional axioms that assert the existence of such a type. This is the approach taken in the article [2], which augments HOL with additional axioms whose essence is to assert the existence of a new type V whose elements correspond to sets that can be proved to exist in ZFC. To avoid obvious inconsistency, clearly not every set of elements at type V can correspond to an element of V; the sets that do correspond to elements of V are declared to be "small". The notion of smallness is then extended via equipollence to obtain a notion of small sets at arbitrary types.

In the article [3] the present author used the ZFC-in-HOL axiomatization to define a "set category" whose objects are in bijective correspondence with the small sets at type V. This does produce a usable category of small sets, but there are some identifiable deficiencies. First of all, the construction is very closely tied to the ZFC-in-HOL development and the particular type V introduced there. It would be more flexible if somehow the necessary assumptions could be distilled and expressed (using Isabelle's locale fea-

ture, for example) as assumptions about an unspecified type named by a type variable, or, more generally, as assumptions about a set of elements of such a type. Secondly, the construction given in [3] was somewhat *ad hoc*, which although it served its purpose as a proof-of-concept, did not pay much attention to the ultimate usability of the theory nor provide much guidance as to how the construction might be generalized to produce categories of sets with additional structure (a category of groups, for example).

The purpose of present article is to revisit the problem of formalizing the category of sets in Isabelle/HOL while trying to address the above deficiencies. The approach we have taken is as follows. We first attempt to decouple the underlying extensions needed to HOL from the particular development in ZFC-in-HOL and to re-express these extensions, independently of the particular type *V*, using Isabelle's locale feature. This leads us to identify two main aspects that need to be addressed: (1) the notion of "smallness" of a set; and (2) and notion of a "universe", comprising a collection of sets that is in some sense closed under the usual set-theoretic constructions.

The notion of smallness is addressed by the theory *Smallness*, which introduces several locales whose assumptions concern a function *sml :: 'V set => bool* which is understood as specifying a collection of sets, at some unspecified but fixed type *'V*, which are to be considered "small". A base locale, *smallness*, assumes as a regularity condition that the function *sml* respects equipollence and then uses polymorphism to extend this function by equipollence to a function *small :: 'a set => bool* at every type. (It is done this way because types mentioned in locale parameters are essentially fixed, whereas functions defined in the body of a locale can be polymorphic.) Several extensions to the *smallness* locale are then defined, corresponding to various assumptions about what sets are to be considered as small. The *small_finite* locale is satisfied by notions of smallness for which arbitrary finite sets are considered to be small. The *small_nat* locale is satisfied by notions of smallness for which the set of natural numbers is small. The *small_product* locale is satisfied by notions of smallness that are preserved under cartesian product. The *small_sum* locale is satisfied by notions of smallness that are preserved under the formation of small-indexed unions. The *small_powerset* locale is satisfied by notions of smallness for which the set of all subsets of a small set is again small. The *small_funcset* locale is satisfied by notions of smallness that are preserved by a suitable construction of function spaces (this involves some technical issues that result from the the fact that HOL requires all functions to be total).

The notion of a "universe" is addressed by the theory *Universe*. This theory introduces several locales whose assumptions concern a set *univ :: 'U set*, at some unspecified but fixed type *'U*, which admits embeddings of various other sets; typically resulting from constructions on *univ* itself. A base locale, *embedding*, defines the notion of an injective embedding of another set into *univ*. The *lifting* locale is satisfied when the set *univ* embeds the disjoint union of itself and an additional element. The *pairing* locale is satisfied when the set *univ* embeds $univ \times univ$. The *powering* locale is satisfied when the set *univ* embeds the set of all its "small" subsets. The *tupling* locale is satisfied when the set *univ* embeds the set of all "small extensional functions" on its elements (here, again, there are some technical issues to be addressed). Finally, the *universe* locale combines the *tupling* locale with the assumption that the set of natural numbers is small.

Having defined the above locales, we proceed to defining the *sets_cat* locale, which axiomatizes the notion "category of sets and functions". This definition follows a general plan that can be applied to construct locales that axiomatize categories of other kinds of algebraic structures. We first define the locale *sets_cat_base*, which is satisfied by an arbitrary category C with terminal object together with a notion of smallness. The *sets_cat_base* locale provides a convenient place to define correspondences, between objects of C and sets and between arrows of C and functions. Specifically, after making an arbitrary choice of terminal object, we define a function *Set* that takes each object to the set of its global elements, and a function *Fun* that takes each arrow to the function on global elements it induces by composition. Here we are exploiting the well-pointedness of a category of sets and functions to simplify things a bit. To apply the same plan to categories that are not well-pointed, we will have to use generalized elements instead, which is possible, but more cumbersome.

The *sets_cat_base* locale is then extended to the *sets_cat* locale by adding four axioms. The first axiom asserts that the set of global elements of every object is small. The second axiom asserts that the mapping *Fun* that takes arrows to functions on global elements is injective. The third axiom asserts that for every "real" function F from the set of global elements of object a to the set of global elements of object b there is an arrow $f : a \to b$ of C such that $Fun\ f = f$. Finally, the fourth axiom, which we call "repleteness", asserts that for every small subset A of the set of arrows of C there exists an object a of C such that the set of global elements of a is equipollent with A. Although the restrictions imposed by Isabelle/HOL on locale definitions require that this axiom be expressed with respect to a fixed type, namely the type of arrows of C, in the body of the locale we can immediately extend the repleteness property to show the existence of objects corresponding to small sets at arbitrary types, as long as a set for which we want to obtain an object "embeds" via an injective mapping into the set of arrows of C.

The gist of the *sets_cat* axioms is to assert the existence of a "meta-functor" from C to "real sets" (of global elements of C) and "real functions" (between sets of global elements), which is full, faithful, and surjective from objects to small sets (of arrows of C). Moreover, we can obtain an object corresponding to a given small set at an arbitrary type, assuming that there is an embedding of that set into the set of arrows of C. So, the image of C under this meta-functor is a "meta-category" whose objects are sets of arrows of C and whose arrows are functions between such sets. This meta-category is in general only equivalent to C, not isomorphic to it, because when we pass from a small set A to the corresponding object *mkide A* and then back to the set *Set(mkide A)* of global elements of *mkide a*, we recover a set that is only equipollent to A, rather than equal to it. We therefore obtain a pair of inverse "comparison maps" between an externally given small set A and the set of global elements of the object *mkide a* corresponding to it. The map *IN* encodes each element of A as a corresponding global element of *mkide A*; the inverse map *OUT* decodes each global element of *mkide A* to the corresponding element of A. We use the just-outlined structure to prove a "categoricity" result which states that, a category C that satisfies the *sets_cat* locale is, up to equivalence of categories, the unique such category whose set of arrows has the same cardinality as that of C. The same overall pattern can be applied to algebraic structures more general than sets, but

note that in this case the comparison maps will end up being isomorphisms for these structures, rather than just invertible functions.

We then proceed to develop the consequences of the *sets_cat* axioms; proving a set of properties roughly patterned after those in Lawvere's "Elementary Theory of the Category of Sets" [1]. In brief, we show that, if the collection of arrows of C forms a "universe", then C is well-pointed, small-complete and small co-complete, cartesian closed, has a subobject classifier and a natural numbers object, and splits all epimorphisms. The fact that the correspondences, between objects and sets and between arrows and functions, have been defined in terms of structure intrinsic to the category C means that we can carry out the proofs without having to reference concrete details of the construction of a particular underlying type, such as that of the type V from *ZFC_in_HOL*. Of particular interest is the pattern we use to show the existence of limits and colimits in C. Consider the case of binary products as an example. We know that the set of global elements of the product $a \otimes b$ of objects a and b of C should be equipollent with the cartesian product $Set\ a \times Set\ b$ of the set of global elements of a and that of b. Moreover, the sets of global elements of a and b are small (by the locale assumptions), so if we have available as an additional assumption about smallness that it is preserved by cartesian product, then we may conclude that the set $Set\ a \times Set\ b$ is also small. If we have also assumed the existence of a pairing function, which injectively maps pairs of arrows of C to arrows of C, then we may use repleteness to prove the existence of an object $a \otimes b$ whose set of global elements is equipollent with $Set\ a \times Set\ b$. Once the existence of this object has been shown, then we can prove that it is in fact a categorical product of a and b. To do this, we need to obtain the projections, but these are just the arrows of C that correspond to the "real" projection functions on $Set\ a \times Set\ b$. So to summarize, to show that C admits a particular categorical construction, we first carry out a corresponding construction on sets of global elements. This will typically result in a set at a higher type than that of the arrows of C. To obtain an object of C we must show that this set is small and in addition that it "embeds" back down into the set of arrows of C.

Finally, as everything described up to this point has been carried out axiomatically (the locale assumptions are the axioms), to keep ourselves honest we have to show that the axioms are actually consistent. We do this by constructing two "top-level" interpretations of the *sets_cat* locale. One interpretation is carried out in "vanilla HOL" without the use of *ZFC_in_HOL* and takes "finite" as the notion of smallness. It shows that the category whose objects are the natural numbers and whose arrows correspond to functions between finite sets, interprets the *sets_cat_with_tupling* locale, which satisfies all the smallness and embedding assumptions we use, except for the assumption that the set of natural numbers is small. The second interpretation, which uses *ZFC_in_HOL*, shows that the category of sets we constructed in the previous article [3] interprets the *sets_cat_with_tupling* locale as well as the *small_nat* locale, which asserts also that the set of natural numbers is small.

In the end, what we achieve is a locale, *sets_cat*, which axiomatizes the notion of a category of sets and functions, and which can be used to perform reasoning internal to such a category without having to refer to details of a particular concrete construction. When required, we can pass from inside the category to the "external world" via a fully

faithful functorial mapping. Functions that exist externally can be internalized as arrows using the fullness of this mapping. In addition, sets that exist externally, at any type, can be internalized as objects of the category, provided that we establish two facts: (1) their smallness; and (2) that they can be embedded into the set of arrows of the category. We have demonstrated this procedure by using it to prove the familiar properties of a "set category".

Chapter 2

Smallness

theory *Smallness*
imports *HOL−Library.Equipollence*
begin

The purpose of this theory is to axiomatize, using locales, a notion of "small set" that is polymorphic over types and that is preserved by certain set-theoretic constructions in the way we would usually expect. We first observe that we cannot simply define such a notion within normal HOL, because HOL does not permit us to quantify over types, nor does it permit us to show the existence of a single type "large enough" to admit sets of all cardinalities that would result, say, by iterating the application of the powerset operator starting with some infinite set. So any way of defining "smallness" is going to require extending HOL in some way. Note that this is exactly what is already done in the article [2], which axiomatizes a particular type V and then defines a polymorphic function *small* using the properties of that type. However, we would prefer to have a notion of smallness that is not tied to one particular type or construction.

Ideally, what we would like to do is to define a locale *smallness*, whose assumptions express closure properties that we would like to hold for a function $small :: 'a\ set \Rightarrow bool$. This does not quite work, though, because the types involved in locale assumptions are essentially fixed, so that the function *small* could not be applied polymorphically. A workaround is to have the locale assumption express closure properties of a function $sml :: 'b \Rightarrow bool$, where type $'b$ is essentially fixed, and then to define within the locale context the actually polymorphic function $small :: 'a \Rightarrow bool$, which extends *sml* by equipollence to an arbitrary type $'a$. This is essentially what is done in [2], except rather than basing the definition on a notion of smallness derived from a particular type V we are defining a locale that takes the type and associated basic notion of smallness as a parameter.

In the development here we have defined a basic *smallness* locale, along with several extensions that express various collections of closure properties. It is not yet clear how useful this level of generality might turn out to be in practice, however at the very least, this allows us to segregate the property "the set of natural number is small" from the others. This allows us to consider two interpretations for "category of small sets and functions"; one of which only has objects corresponding to finite sets and the other of

which also has objects corresponding to infinite sets.

2.1 Basic Notions

Here we define the base locale *smallness*, which takes as a parameter a function *sml* :: $'a$ *set* $\Rightarrow$ *bool* that defines a basic notion of smallness at some fixed type, and extends this basic notion by equipollence to arbitrary types. We assume that the basic notion of smallness *sml* given as a parameter already respects equipollence, so that *small* and *sml* coincide at type $'a$.

locale *smallness* =
fixes *sml* :: $'V$ *set* $\Rightarrow$ *bool*
assumes *lepoll-small-ax*: $[\![sml\ X;\ lepoll\ Y\ X]\!] \Longrightarrow sml\ Y$
begin

definition *small* :: $'a$ *set* $\Rightarrow$ *bool*
where $small\ X \equiv \exists X_0.\ sml\ X_0 \wedge X \approx X_0$

lemma *smallI*:
assumes $sml\ X_0$ **and** $X \approx X_0$
shows *small X*
⟨*proof*⟩

lemma *smallE*:
assumes *small X*
and $\bigwedge X_0.\ [\![sml\ X_0;\ X \approx X_0]\!] \Longrightarrow T$
shows *T*
⟨*proof*⟩

lemma *small-iff-sml*:
shows $small\ X \longleftrightarrow sml\ X$
⟨*proof*⟩

lemma *lepoll-small*:
assumes *small X* **and** *lepoll Y X*
shows *small Y*
⟨*proof*⟩

lemma *smaller-than-small*:
assumes *small X* **and** $Y \subseteq X$
shows *small Y*
⟨*proof*⟩

lemma *small-image* [*intro*, *simp*]:
assumes *small X*
shows *small* (*f* ‘ *X*)
⟨*proof*⟩

lemma *small-image-iff* [*simp*]: *inj-on f A* $\Longrightarrow$ *small* (*f* ‘ *A*) $\longleftrightarrow$ *small A*
⟨*proof*⟩

lemma *small-Collect* [*simp*]: *small X* $\Longrightarrow$ *small* {*x* ∈ *X*. *P x*}
⟨*proof*⟩

end

2.2 Smallness of Finite Sets

The locale *small-finite* is satisfied by notions of smallness that admit small sets of arbitrary finite cardinality.

locale *small-finite* =
smallness +
assumes *small-finite-ax*: ∃ *Y*. *sml Y* ∧ *eqpoll* {*1..n* :: *nat*} *Y*
begin

lemma *small-finite*:
shows *finite X* $\Longrightarrow$ *small X*
⟨*proof*⟩

lemma *small-insert*:
assumes *small X*
shows *small* (*insert a X*)
⟨*proof*⟩

lemma *small-insert-iff* [*iff*]: *small* (*insert a X*) $\longleftrightarrow$ *small X*
⟨*proof*⟩

end

2.3 Smallness of Binary Products

The locale *small-product* is satisfied by notions of smallness that are preserved under cartesian product.

locale *small-product* =
smallness +
assumes *small-product-ax*: ⟦*sml X*; *sml Y*⟧ $\Longrightarrow$ ∃ *Z*. *sml Z* ∧ *eqpoll* (*X* × *Y*) *Z*
begin

lemma *small-product* [*simp*]:
assumes *small X small Y* **shows** *small* (*X* × *Y*)
⟨*proof*⟩

end

2.4 Smallness of Sums

The locale *small-sum* is satisfied by notions of smallness that are preserved under the formation of small-indexed unions.

locale *small-sum* =
  small-finite +
assumes *small-sum-ax*: $[\![sml\ X;\ \bigwedge x.\ x \in X \Longrightarrow sml\ (F\ x)]\!]$
  $\Longrightarrow \exists\, U.\ sml\ U \wedge eqpoll\ (Sigma\ X\ F)\ U$
begin

  lemma *small-binary-sum*:
  assumes *small X* **and** *small Y*
  shows *small* $((\{False\} \times X) \cup (\{True\} \times Y))$
  ⟨*proof*⟩

  lemma *small-union*:
  assumes *X*: *small X* **and** *Y*: *small Y*
  shows *small* $(X \cup Y)$
  ⟨*proof*⟩

  lemma *small-Union-spc*:
  assumes A_0: *sml* A_0 **and** *B*: $\bigwedge x.\ x \in A_0 \Longrightarrow small\ (B\ x)$
  shows *small* $(\bigcup x{\in}A_0.\ B\ x)$
  ⟨*proof*⟩

  lemma *small-Union* [*simp*, *intro*]:
  assumes *A*: *small A* **and** *B*: $\bigwedge x.\ x \in A \Longrightarrow small\ (B\ x)$
  shows *small* $(\bigcup x{\in}A.\ B\ x)$
  ⟨*proof*⟩

The *small-sum* locale subsumes the *small-product* locale, in the sense that any notion of smallness that satisfies *small-sum* also satisfies *small-product*.

  sublocale *small-product*
  ⟨*proof*⟩

end

2.5 Smallness of Powersets

The locale *small-powerset* is satisfied by notions of smallness for which the set of all subsets of a small set is again small.

locale *small-powerset* =
  smallness +
assumes *small-powerset-ax*: $sml\ X \Longrightarrow \exists\, PX.\ sml\ PX \wedge eqpoll\ (Pow\ X)\ PX$
begin

  lemma *small-powerset*:

assumes *small X*
shows *small* (*Pow X*)
⟨*proof*⟩

lemma *large-UNIV*:
shows ¬ *small* (*UNIV* :: *'a set*)
⟨*proof*⟩

end

2.6 Smallness of the Set of Natural Numbers

The locale *small-nat* is satisfied by notions of smallness for which the set of natural numbers is small.

locale *small-nat* =
smallness +
assumes *small-nat-ax*: ∃ *X*. *sml X* ∧ *eqpoll X* (*UNIV* :: *nat set*)
begin

lemma *small-nat*:
shows *small* (*UNIV* :: *nat set*)
⟨*proof*⟩

end

2.7 Smallness of Function Spaces

The objective of this section is to define a locale that is satisfied by notions of smallness for which "the set of functions between two small sets is small." This is complicated in HOL by the requirement that all functions be total, which forces us to define the value of a function at points outside of what we would consider to be its domain. If we don't impose some restriction on the values taken on by a function outside of its domain, then the set of functions between a domain and codomain set could be large, even if the domain and codomain sets themselves are small. We could limit the possible variation by restricting our consideration to "extensional" functions; *i.e.* those that take on a particular default value outside of their domain, but it becomes awkward if we have to make an *a priori* choice of what this value should be.

The approach we take here is to define the notion of a "popular value" of a function. This will be a value, in the function's range, whose preimage is a large set. The idea here is that the default values of extensional functions will typically have their default values as popular values (though this is not necessarily the case, as a function whose domain type is small will not have any popular values according to this definition). We then define a "small function" to be a function whose range is a small set and which has at most one popular value. The "essential domain" of small function is the set of arguments on which the value of the function is not a popular value. Then we can

consistently require of a smallness notion that, if A and B are small sets, that the set of functions whose essential domains are contained in A and whose ranges are contained in B, is again small.

2.7.1 Small Functions

context *smallness*
begin

abbreviation *popular-value* :: $('b \Rightarrow 'c) \Rightarrow 'c \Rightarrow bool$
where *popular-value* $F\ y \equiv \neg\ small\ \{x.\ F\ x = y\}$

definition *some-popular-value* :: $('b \Rightarrow 'c) \Rightarrow 'c$
where *some-popular-value* $F \equiv SOME\ y.$ *popular-value* $F\ y$

lemma *popular-value-some-popular-value*:
assumes $\exists\, y.$ *popular-value* $F\ y$
shows *popular-value* F (*some-popular-value* F)
⟨*proof*⟩

abbreviation *at-most-one-popular-value*
where *at-most-one-popular-value* $F \equiv \exists_{\leq 1}\ y.$ *popular-value* $F\ y$

definition *small-function*
where *small-function* $F \equiv$ *small* (*range* F) $\wedge$ *at-most-one-popular-value* F

lemma *small-functionI* [*intro*]:
assumes *small* (*range* f) **and** *at-most-one-popular-value* f
shows *small-function* f
⟨*proof*⟩

lemma *small-functionD* [*dest*]:
assumes *small-function* f
shows *small* (*range* f) **and** *at-most-one-popular-value* f
⟨*proof*⟩

end

If there are small sets of arbitrarily large finite cardinality, then the preimage of a popular value of a function must be an infinite set (in particular, it must be nonempty, since the empty set must be small). We can derive various useful consequences of this fairly lax assumption.

context *small-finite*
begin

lemma *popular-value-in-range*:
assumes *popular-value* $F\ v$
shows $v \in$ *range* F

⟨*proof*⟩

lemma *small-function-const*:
shows *small-function* (λ*x*. *y*)
⟨*proof*⟩

definition *inv-into*$_E$
where *inv-into*$_E$ *X f* ≡ λ*y*. *if y* ∈ *f* ‘ *X then inv-into X f y*
else SOME x. *popular-value f* (*f x*)

lemma *small-function-inv-into*$_E$:
assumes *small-function f* **and** *inj-on f X*
shows *small-function* (*inv-into*$_E$ *X f*)
⟨*proof*⟩

end

context *small-sum*
begin

lemma *small-function-comp*:
assumes *small-function f* **and** *small-function g*
shows *small-function* (*g* ∘ *f*)
⟨*proof*⟩

In the present context, a small function has a popular value if and only if its domain type is large. This simplifies special cases that concern whether or not a function happens to have any popular value at all.

lemma *ex-popular-value-iff*:
assumes *small-function* (*F* :: *'b* ⇒ *'c*)
shows (∃ *v*. *popular-value F v*) ⟷ ¬ *small* (*UNIV* :: *'b set*)
⟨*proof*⟩

A consequence is that the preimage of the set of all unpopular values of a function is small.

lemma *small-preimage-unpopular*:
fixes *F* :: *'b* ⇒ *'c*
assumes *small-function F*
shows *small* {*x*. *F x* ≠ *some-popular-value F*}
⟨*proof*⟩

Here we are working toward showing that a small function has a “small encoding”, which consists of its graph for arguments that map to non-popular values, paired with the single popular value it has on all other arguments.

abbreviation *SF-Dom*
where *SF-Dom f* ≡ {*x*. ¬ *popular-value f* (*f x*)}

abbreviation *SF-Rng*
where *SF-Rng f* ≡ *f* ‘ *SF-Dom f*

abbreviation *SF-Grph*
where *SF-Grph f* ≡ (λ*x*. (*x*, *f x*)) ‘ *SF-Dom f*

abbreviation *the-PV*
where *the-PV f* ≡ *THE y*. *popular-value f y*

lemma *small-SF-Dom*:
assumes *small-function f*
shows *small* (*SF-Dom f*)
⟨*proof*⟩

lemma *small-SF-Rng*:
assumes *small-function f*
shows *small* (*SF-Rng f*)
⟨*proof*⟩

lemma *small-SF-Grph*:
assumes *small-function f*
shows *small* (*SF-Grph f*)
⟨*proof*⟩

lemma *small-function-expansion*:
assumes *small-function f*
shows *f* = (λ*x*. *if x* ∈ *fst* ‘ *SF-Grph f then* (*THE y*. (*x*, *y*) ∈ *SF-Grph f*) *else the-PV f*)
⟨*proof*⟩

end

2.7.2 Small Funcsets

locale *small-funcset* =
small-sum +
small-powerset
begin

For a suitable definition of “between”, the set of small functions between small sets is small.

lemma *small-funcset*:
assumes *small X* **and** *small Y*
shows *small* {*f*. *small-function f* ∧ *SF-Dom f* ⊆ *X* ∧ *range f* ⊆ *Y*}
⟨*proof*⟩

end

2.8 Smallness of Sets of Lists

A notion of smallness that is preserved under sum and powerset, and in addition declares the set of natural numbers to be small, is sufficiently inclusive as to include any set whose

existence is provable in ZFC. So it is not a surprise that we can show, for example, that the set of lists with elements in a given small set is again small. We do not use this particular fact in the present development, but we will have a use for it in a subsequent article.

locale *small-funcset-and-nat* =
small-funcset +
small-nat
begin

definition *list-as-fn* :: *'b list* ⇒ *nat* ⇒ *'b option*
where *list-as-fn l n* = (*if n* ≥ *length l then None else Some* (*l* ! *n*))

lemma *inj-list-as-fn*:
shows *inj list-as-fn*
⟨*proof*⟩

lemma *small-function-list-as-fn*:
shows *small-function* (*list-as-fn l*)
⟨*proof*⟩

lemma *small-listset*:
assumes *small Y*
shows *small* {*l*. *List.set l* ⊆ *Y*}
⟨*proof*⟩

end

end

Chapter 3

Universe

theory *Universe*
imports *Smallness*
begin

This section defines a "universe" to be a set *univ* that admits embeddings of various other sets, typically the result of constructions on *univ* itself. These embeddings allow us to perform constructions on *univ* that result in sets at higher types, and then to encode the results of these constructions back down into *univ*. An example application is showing that a category admits products: given objects a and b in a category whose arrows form a universe *univ*, for each object x we may form the cartesian product $hom\ x\ a \times hom\ x\ b \subseteq univ \times univ$ and then use an embedding of $univ \times univ$ in *univ* (*i.e.* a pairing function) to map the result back into *univ*. Assuming we can show that the resulting set has the proper structure to be the set of arrows of an object of the category, we obtain an object $a \times b$ with $hom\ x\ (a \times b) \cong hom\ x\ a \times hom\ x\ b$, as required for a product object in a category.

3.1 Embeddings

Here we define some basic notions pertaining to injections into a set *univ*.

locale *embedding* =
fixes *univ* :: *'U set*
begin

abbreviation *is-embedding-of*
where *is-embedding-of* $\iota\ X \equiv$ *inj-on* $\iota\ X \wedge \iota\ `\ X \subseteq univ$

definition *some-embedding-of*
where *some-embedding-of* $X \equiv$ *SOME* ι. *is-embedding-of* $\iota\ X$

abbreviation *embeds*
where *embeds* $X \equiv \exists \iota$. *is-embedding-of* $\iota\ X$

lemma *is-embedding-of-some-embedding-of*:
assumes *embeds X*
shows *is-embedding-of* (*some-embedding-of X*) *X*
⟨*proof*⟩

lemma *embeds-subset*:
assumes *embeds X* **and** $Y \subseteq X$
shows *embeds Y*
⟨*proof*⟩

end

3.2 Lifting

The locale *lifting* axiomatizes a set *univ* that embeds itself, together with an additional element. This is equivalent to *univ* being infinite.

locale *lifting* =
embedding univ
for *univ* :: *'U set* +
assumes *embeds-lift*: *embeds* ({*None*} ∪ *Some* ‘ *univ*)
begin

definition *some-lifting* :: *'U option* ⇒ *'U*
where *some-lifting* ≡ *some-embedding-of* ({*None*} ∪ *Some* ‘ *univ*)

lemma *some-lifting-is-embedding*:
shows *is-embedding-of some-lifting* ({*None*} ∪ *Some* ‘ *univ*)
⟨*proof*⟩

lemma *some-lifting-in-univ* [*intro*, *simp*]:
shows *some-lifting None* ∈ *univ*
and $x \in univ \Longrightarrow$ *some-lifting* (*Some x*) ∈ *univ*
⟨*proof*⟩

lemma *some-lifting-cancel*:
shows ⟦$x \in univ$; *some-lifting* (*Some x*) = *some-lifting None*⟧ ⟹ *False*
and ⟦$x \in univ$; $x' \in univ$; *some-lifting* (*Some x*) = *some-lifting* (*Some x'*)⟧ ⟹ $x = x'$
⟨*proof*⟩

lemma *infinite-univ*:
shows *infinite univ*
⟨*proof*⟩

lemma *embeds-bool*:
shows *embeds* (*UNIV* :: *bool set*)
⟨*proof*⟩

lemma *embeds-nat*:

shows *embeds* (*UNIV* :: *nat set*)
⟨*proof*⟩

end

3.3 Pairing

The locale *pairing* axiomatizes a set *univ* that embeds *univ* × *univ*.

locale *pairing* =
embedding univ
for *univ* :: *'U set* +
assumes *embeds-pairs*: *embeds* (*univ* × *univ*)
begin

definition *some-pairing* :: *'U* ∗ *'U* ⇒ *'U*
where *some-pairing* ≡ *some-embedding-of* (*univ* × *univ*)

lemma *some-pairing-is-embedding*:
shows *is-embedding-of some-pairing* (*univ* × *univ*)
⟨*proof*⟩

abbreviation *pair*
where *pair x y* ≡ *some-pairing* (*x*, *y*)

abbreviation *is-pair* :: *'U* ⇒ *bool*
where *is-pair x* ≡ *x* ∈ *some-pairing* ‘ (*univ* × *univ*)

definition *first* :: *'U* ⇒ *'U*
where *first x* ≡ *fst* (*inv-into* (*univ* × *univ*) *some-pairing x*)

definition *second* :: *'U* ⇒ *'U*
where *second x* = *snd* (*inv-into* (*univ* × *univ*) *some-pairing x*)

lemma *first-conv*:
assumes *x* ∈ *univ* **and** *y* ∈ *univ*
shows *first* (*pair x y*) = *x*
⟨*proof*⟩

lemma *second-conv*:
assumes *x* ∈ *univ* **and** *y* ∈ *univ*
shows *second* (*pair x y*) = *y*
⟨*proof*⟩

lemma *pair-conv*:
assumes *is-pair x*
shows *pair* (*first x*) (*second x*) = *x*
⟨*proof*⟩

lemma *some-pairing-in-univ* [*intro*, *simp*]:
shows $[\![x \in univ;\ y \in univ]\!] \Longrightarrow pair\ x\ y \in univ$
⟨*proof*⟩

lemma *some-pairing-cancel*:
shows $[\![x \in univ;\ x' \in univ;\ y \in univ;\ y' \in univ;\ pair\ x\ y = pair\ x'\ y']\!]$
$\Longrightarrow x = x' \wedge y = y'$
⟨*proof*⟩

end

3.4 Powering

The *powering* locale axiomatizes a universe that embeds the set of all its "small" subsets. Obviously, some condition on the subsets is required because (by Cantor's Theorem) it is not possible for a set to embed the set of *all* its subsets. The concept of "smallness" used here is not fixed, but rather is taken as a parameter.

locale *powering* =
embedding univ +
smallness sml
for *sml* :: $'V\ set \Rightarrow bool$
and *univ* :: $'U\ set$ +
assumes *embeds-small-sets*: *embeds* $\{X.\ X \subseteq univ \wedge small\ X\}$
begin

abbreviation *some-embedding-of-small-sets* :: $('U\ set) \Rightarrow 'U$
where *some-embedding-of-small-sets* ≡ *some-embedding-of* $\{X.\ X \subseteq univ \wedge small\ X\}$

definition *emb-set* :: $('U\ set) \Rightarrow 'U$
where *emb-set* ≡ *some-embedding-of-small-sets*

lemma *emb-set-is-embedding*:
shows *is-embedding-of emb-set* $\{X.\ X \subseteq univ \wedge small\ X\}$
⟨*proof*⟩

lemma *emb-set-in-univ* [*intro*, *simp*]:
shows $[\![X \subseteq univ;\ small\ X]\!] \Longrightarrow emb\text{-}set\ X \in univ$
⟨*proof*⟩

lemma *emb-set-cancel*:
shows $[\![X \subseteq univ;\ small\ X;\ X' \subseteq univ;\ small\ X';\ emb\text{-}set\ X = emb\text{-}set\ X']\!] \Longrightarrow X = X'$
⟨*proof*⟩

If *univ* embeds the collection of all its small subsets, then *univ* itself must be large.

lemma *large-univ*:
shows ¬ *small univ*
⟨*proof*⟩

end

3.5 Tupling

The *tupling* locale axiomatizes a set *univ* that embeds the set of all "small extensional functions" on its elements. Here, the notion of "extensional function" is parametrized by the default value *null* produced by such a function when it is applied to an argument outside of *univ*. The default value *null* is neither assumed to be in *univ* nor outside of it.

locale *tupling* =
  lifting univ +
  pairing univ +
  powering sml univ +
  small-funcset sml
for *sml* :: $'V$ *set* $\Rightarrow$ *bool*
and *univ* :: $'U$ *set*
and *null* :: $'U$
begin

EF is the set of extensional functions on *univ*. These map *univ* to *univ* $\cup$ $\{$*null*$\}$ and map values outside of *univ* to *null*. The default value *null* might or might not be an element of *univ*. The set *SEF* is the subset of *EF* consisting of those functions that are "small functions".

definition *EF*
where $EF \equiv \{f.\ f\ `\ univ \subseteq univ \cup \{null\} \wedge (\forall x.\ x \notin univ \longrightarrow f\ x = null)\}$

abbreviation *SEF*
where $SEF \equiv Collect\ small\text{-}function \cap EF$

lemma *EF-apply*:
assumes $F \in EF$
shows $x \in univ \Longrightarrow F\ x \in univ \cup \{null\}$
and $x \notin univ \Longrightarrow F\ x = null$
  ⟨*proof*⟩

Since *univ* is large, the set of all values at type $'U$ must also be large. This implies that every small extensional function having type $'U$ as its domain type must have a popular value.

lemma *SEFs-have-popular-value*:
assumes $F \in SEF$
shows $\exists v.\ popular\text{-}value\ F\ v$
  ⟨*proof*⟩

The following technical lemma uses powering to obtain an encoding of small extensional functions as elements of *univ*. The idea is that a small extensional function F mapping *univ* to *univ* $\cup$ $\{$*null*$\}$ can be canonically described by a small subset of *univ* $\times$ (*univ* $\cup$ $\{$*null*$\}$) consisting of all pairs $(x,\ F\ x) \subseteq univ \times (univ \cup \{null\})$ for which

$F\ x$ is not a popular value, together with the single popular value of F taken at other arguments x not represented by such pairs.

lemma *embeds-SEF*:
shows *embeds SEF*
⟨*proof*⟩

definition *some-embedding-of-small-functions* :: $('U \Rightarrow 'U) \Rightarrow 'U$
where *some-embedding-of-small-functions* ≡ *some-embedding-of SEF*

lemma *some-embedding-of-small-functions-is-embedding*:
shows *is-embedding-of some-embedding-of-small-functions SEF*
⟨*proof*⟩

lemma *some-embedding-of-small-functions-in-univ* [*intro*, *simp*]:
assumes $F \in SEF$
shows *some-embedding-of-small-functions* $F \in univ$
⟨*proof*⟩

lemma *some-embedding-of-small-functions-cancel*:
assumes $F \in SEF$ **and** $F' \in SEF$
and *some-embedding-of-small-functions* F = *some-embedding-of-small-functions* F'
shows $F = F'$
⟨*proof*⟩

end

3.6 Universe

The *universe* locale axiomatizes a set that is equipped with an embedding of its own small extensional function space, and in addition the set of natural numbers is required to be small (*i.e.* there is a small infinite set).

locale *universe* =
tupling sml univ null +
small-nat sml
for *sml* :: *'V set* ⇒ *bool*
and *univ* :: *'U set*
and *null* :: *'U*
begin

For a fixed notion of smallness, the property of being a universe is respected by equipollence; thus it is a property of the set itself, rather than something that depends on the ambient type.

lemma *is-respected-by-equipollence*:
assumes *eqpoll univ univ'*
shows *universe sml univ'*
⟨*proof*⟩

A universe admits an embedding of all lists formed from its elements.

sublocale *small-funcset-and-nat* ⟨*proof*⟩

fun *some-embedding-of-lists* :: *'U list* ⇒ *'U*
where *some-embedding-of-lists* [] = *some-lifting None*
| *some-embedding-of-lists* (*x* # *l*) =
some-lifting (*Some* (*some-pairing* (*x*, *some-embedding-of-lists l*)))

lemma *embeds-lists*:
shows *embeds* {*l*. *List.set l* ⊆ *univ*}
and *is-embedding-of some-embedding-of-lists* {*l*. *List.set l* ⊆ *univ*}
⟨*proof*⟩

A universe also admits an embedding of all small sets of lists formed from its elements.

lemma *embeds-small-sets-of-lists*:
shows *is-embedding-of* (λ*X*. *some-embedding-of-small-sets* (*some-embedding-of-lists* ‘ *X*))
{*X*. *X* ⊆ {*l*. *list.set l* ⊆ *univ*} ∧ *small X*}
and *embeds* {*X*. *X* ⊆ {*l*. *list.set l* ⊆ *univ*} ∧ *small X*}
⟨*proof*⟩

end

end

Chapter 4

The Category of Small Sets

theory *SetsCat*
imports *Category3.SetCat Category3.CategoryWithPullbacks Category3.CartesianClosedCategory Category3.EquivalenceOfCategories Category3.Colimit Universe*
begin

In this section we consider the category of small sets and functions between them as an exemplifying instance of the pattern we propose for working with large categories in HOL. We define a locale *sets-cat*, which axiomatizes a category with terminal object, such that each object determines a "small" set (the set of its global elements), there is an object corresponding to any externally given small set, and such that the hom-sets between objects are in bijection with the small extensional functions between sets of global elements. We show that this locale characterizes the category of small sets and functions, in the sense that, for a fixed notion of smallness, any two interpretations of the *sets-cat* locale are equivalent as categories. We then proceed to derive various familiar properties of a category of sets; assuming in each case that the notion of "smallness" satisfies suitable conditions as defined in the theory *Smallness*, and that the collection of all arrows of the category satisfies suitable closure conditions as defined in the theory *Universe*. In particular, we show if the collection of arrows forms a "universe", then the category is well-pointed, small-complete and small co-complete, cartesian closed, has a subobject classifier and a natural numbers object, and splits all epimorphisms.

4.1 Basic Definitions and Properties

We will describe the category of small sets and functions as a certain kind of category with terminal object, which has been equipped with a notion of "smallness" that specifies what sets will correspond to objects in the category.

locale *sets-cat-base* =
  smallness sml +
  category-with-terminal-object C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹⋅› *55*)

begin

sublocale *embedding* ‹*Collect arr*› ⟨*proof*⟩

Every object in the category determines a set: its set of global elements (we make an arbitrary choice of terminal object).

abbreviation *Set*
where *Set* ≡ *hom* $\mathbf{1}^?$

Every arrow in the category determines an extensional function between sets of global elements.

definition *Fun*
where *Fun f x* ≡ *if x* ∈ *Set* (*dom f*) *then f* · *x else null*

abbreviation *Hom*
where *Hom a b* ≡ (*Set a* → *Set b*) ∩ {*F*. ∀ *x*. *x* ∉ *Set a* ⟶ *F x* = *null*}

lemma *Fun-in-Hom*:
assumes «*f* : *a* → *b*»
shows *Fun f* ∈ *Hom a b*
⟨*proof*⟩

lemma *Set-some-terminal*:
shows *Set some-terminal* = {*some-terminal*}
⟨*proof*⟩

lemma *Fun-some-terminator*:
assumes *ide a*
shows *Fun* $\mathrm{t}^?$[*a*] = (λ*x*. *if x* ∈ *Set a then* $\mathbf{1}^?$ *else null*)
⟨*proof*⟩

The following function will allow us to obtain an object corresponding to an externally given set. The set of global elements of the object is to be equipollent with the given set. We give the definition here, but of course it will be necessary to prove that this function actually does produce such an object under suitable conditions.

definition *mkide* :: *'a set* ⇒ *'U*
where *mkide A* ≡ *SOME a*. *ide a* ∧ *Set a* ≈ *A*

end

The following locale states our axioms for the category of small sets and functions. The axioms assert: (1) that the set of global elements of every object is small; (2) that the mapping from hom-sets to extensional functions between small sets of global elements is injective and surjective; and (3) that the category is "replete" in the sense that for every small set of arrows of the category there exists an object whose set of elements is equipollent with it.

locale *sets-cat* =
sets-cat-base sml C

for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*) +
assumes *small-Set*: *ide a* ⟹ *small* (*Set a*)
and *inj-Fun*: ⟦*ide a*; *ide b*⟧ ⟹ *inj-on Fun* (*hom a b*)
and *surj-Fun*: ⟦*ide a*; *ide b*⟧ ⟹ *Hom a b* ⊆ *Fun* ‘ (*hom a b*)
and *repleteness-ax*: ⟦*small A*; *A* ⊆ *Collect arr*⟧ ⟹ ∃ *a*. *ide a* ∧ *Set a* ≈ *A*
begin

It is convenient to extend the repleteness property to apply to any small set, at any type, which happens to have an embedding into the collection of arrows of the category.

lemma *repleteness*:
assumes *small A* **and** *embeds A*
shows ∃ *a*. *ide a* ∧ *Set a* ≈ *A*
⟨*proof*⟩

We obtain a pair of inverse comparison maps between an externally given small set A and the set of global elements of the object *mkide a* corresponding to it. The map *IN* encodes each element of A as a global element of *mkide A*. The inverse map *OUT* decodes global elements of *mkide A* to the corresponding elements of A. We will need to pay attention to these comparison maps when relating notions internal to the category to notions external to it. However, when working completely internally to the category these maps do not appear at all.

definition *OUT* :: *'a set* ⇒ *'U* ⇒ *'a*
where *OUT A* ≡ *SOME F*. *bij-betw F* (*Set* (*mkide A*)) *A*

abbreviation *IN* :: *'a set* ⇒ *'a* ⇒ *'U*
where *IN A* ≡ *inv-into* (*Set* (*mkide A*)) (*OUT A*)

The following is the main fact that allows us to produce objects of the category. It states that, given any small set A for which there is some embedding into the collection of arrows of the category, there exists a corresponding object *mkide A* whose set of global elements is equipollent to A.

lemma *ide-mkide*:
assumes *small A* **and** *embeds A*
shows [*intro*]: *ide* (*mkide A*)
and *Set* (*mkide A*) ≈ *A*
⟨*proof*⟩

lemma *bij-OUT*:
assumes *small A* **and** *embeds A*
shows *bij-betw* (*OUT A*) (*Set* (*mkide A*)) *A*
⟨*proof*⟩

lemma *bij-IN*:
assumes *small A* **and** *embeds A*
shows *bij-betw* (*IN A*) *A* (*Set* (*mkide A*))
⟨*proof*⟩

lemma *OUT-elem-of*:
assumes *small A* **and** *embeds A* **and** «$x : \mathbf{1}^? \to$ *mkide A*»
shows *OUT A x* $\in$ *A*
⟨*proof*⟩

lemma *IN-in-hom*:
assumes *small A* **and** *embeds A* **and** $x \in A$ **and** $a =$ *mkide A*
shows «*IN A x* : $\mathbf{1}^? \to a$»
⟨*proof*⟩

lemma *IN-OUT*:
assumes *small A* **and** *embeds A*
shows $x \in$ *Set* (*mkide A*) $\Longrightarrow$ *IN A* (*OUT A x*) $= x$
⟨*proof*⟩

lemma *OUT-IN*:
assumes *small A* **and** *embeds A*
shows $x \in A \Longrightarrow$ *OUT A* (*IN A x*) $= x$
⟨*proof*⟩

lemma *Fun-IN*:
assumes *small A* **and** *embeds A* **and** $y \in A$
shows *Fun* (*IN A y*) = (λx. *if* $x = \mathbf{1}^?$ *then IN A y else null*)
⟨*proof*⟩

The following function enables us to obtain an arrow of the category by specifying an extensional function between sets of global objects.

definition *mkarr* :: $'U \Rightarrow 'U \Rightarrow ('U \Rightarrow 'U) \Rightarrow 'U$
where *mkarr a b F* $\equiv$ *if ide a* $\wedge$ *ide b* $\wedge$ *F* $\in$ *Hom a b*
then SOME f. «$f : a \to b$» $\wedge$ *Fun f* $= F$
else null

lemma *mkarr-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *F* $\in$ *Hom a b*
shows «*mkarr a b F* : $a \to b$»
⟨*proof*⟩

lemma *arr-mkarr* [*intro*, *simp*]:
assumes *ide a* **and** *ide b* **and** *F* $\in$ *Hom a b*
shows *arr* (*mkarr a b F*)
⟨*proof*⟩

lemma *arr-mkarrD* [*dest*]:
assumes *arr* (*mkarr a b F*)
shows *ide a* **and** *ide b* **and** *F* $\in$ *Hom a b*
⟨*proof*⟩

lemma *arr-mkarrE* [*elim*]:
assumes *arr* (*mkarr a b F*)

and $[\![ide\ a;\ ide\ b;\ F \in Hom\ a\ b]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *dom-mkarr* [*simp*]:
assumes *arr* (*mkarr a b F*)
shows *dom* (*mkarr a b F*) = *a*
⟨*proof*⟩

lemma *cod-mkarr* [*simp*]:
assumes *arr* (*mkarr a b F*)
shows *cod* (*mkarr a b F*) = *b*
⟨*proof*⟩

lemma *Fun-mkarr* [*simp*]:
assumes *arr* (*mkarr a b F*)
shows *Fun* (*mkarr a b F*) = *F*
⟨*proof*⟩

lemma *mkarr-Fun*:
assumes «*f* : *a* → *b*»
shows *mkarr a b* (*Fun f*) = *f*
⟨*proof*⟩

The locale assumptions ensure that, for any two objects a and b, there is a bijection between the hom-set *hom a b* and the set *Hom a b* of extensional functions from *Set a* to *Set b*.

lemma *bij-Fun*:
assumes *ide a* **and** *ide b*
shows *bij-betw Fun* (*hom a b*) (*Hom a b*)
and *bij-betw* (*mkarr a b*) (*Hom a b*) (*hom a b*)
⟨*proof*⟩

lemma *arr-eqI*:
assumes *par t u* **and** *Fun t* = *Fun u*
shows *t* = *u*
⟨*proof*⟩

lemma *arr-eqI′*:
assumes *in-hom f a b* **and** *in-hom g a b*
and ⋀*x*. *in-hom x* $\mathbf{1}^?$ *a* ⟹ *f* · *x* = *g* · *x*
shows *f* = *g*
⟨*proof*⟩

lemma *Fun-arr*:
assumes «*f* : *a* → *b*»
shows *Fun f* = (λ*x*. *if x* ∈ *Set a then f* · *x else null*)
⟨*proof*⟩

lemma *Fun-ide*:
assumes *ide a*
shows *Fun a* = (λ*x*. *if x* ∈ *Set a then x else null*)
⟨*proof*⟩

lemma *Fun-comp*:
assumes *seq t u*
shows *Fun* (*t* · *u*) = *Fun t* ∘ *Fun u*
⟨*proof*⟩

lemma *mkarr-comp*:
assumes *seq g f*
shows *mkarr* (*dom f*) (*cod g*) (*Fun g* ∘ *Fun f*) = *g* · *f*
⟨*proof*⟩

lemma *comp-mkarr*:
assumes *arr* (*mkarr a b F*) **and** *arr* (*mkarr b c G*)
shows *mkarr b c G* · *mkarr a b F* = *mkarr a c* (*G* ∘ *F*)
⟨*proof*⟩

lemma *app-mkarr*:
assumes *in-hom* (*mkarr a b F*) *a b* **and** *in-hom x* $\mathbf{1}^?$ *a*
shows *mkarr a b F* · *x* = *F x*
⟨*proof*⟩

lemma *ide-as-mkarr*:
assumes *ide a*
shows *mkarr a a* (λ*x*. *if x* ∈ *Set a then x else null*) = *a*
⟨*proof*⟩

An object a is terminal if and only if its set of global elements *Set a* is a singleton set.

lemma *terminal-char*:
shows *terminal a* ⟷ *ide a* ∧ (∃!*x*. *x* ∈ *Set a*)
⟨*proof*⟩

An object a is initial if and only if its set of global elements *Set a* is the empty set, except in the degenerate situation in which every object is both an initial and a terminal object.

lemma *initial-char*:
shows *initial a* ⟷ *ide a* ∧ (*Set a* = {} ∨ (∀ *b*. *ide b* ⟶ *terminal b*))
⟨*proof*⟩

An arrow is a monomorphism if and only if the corresponding function is injective.

lemma *mono-char*:
shows *mono f* ⟷ *arr f* ∧ *inj-on* (*Fun f*) (*Set* (*dom f*))
⟨*proof*⟩

An arrow is a retraction if and only if the corresponding function is surjective.

lemma *retraction-char*:
shows *retraction f* $\longleftrightarrow$ *arr f* $\wedge$ *Fun f* ‘ *Set* (*dom f*) = *Set* (*cod f*)
⟨*proof*⟩

An arrow is a isomorphism if and only if the corresponding function is a bijection.

lemma *iso-char*:
shows *iso f* $\longleftrightarrow$ *arr f* $\wedge$ *bij-betw* (*Fun f*) (*Set* (*dom f*)) (*Set* (*cod f*))
⟨*proof*⟩

lemma *isomorphic-char*:
shows *isomorphic a b* $\longleftrightarrow$ *ide a* $\wedge$ *ide b* $\wedge$ *Set a* $\approx$ *Set b*
⟨*proof*⟩

end

4.2 Categoricity

The following is a kind of “categoricity in power” result which states that, for a fixed notion of smallness, if C and D are “sets categories” whose collections of arrows are equipollent, then in fact C and D are equivalent categories.

lemma *categoricity*:
assumes *sets-cat sml C* **and** *sets-cat sml D*
and *Collect* (*partial-composition.arr C*) $\approx$ *Collect* (*partial-composition.arr D*)
shows *equivalent-categories C D*
⟨*proof*⟩

4.3 Well-Pointedness

context *sets-cat*
begin

lemma *is-well-pointed*:
assumes *par f g* **and** $\bigwedge x.\ x \in$ *Set* (*dom f*) $\Longrightarrow f \cdot x = g \cdot x$
shows $f = g$
⟨*proof*⟩

end

4.4 Epis Split

In this section we assume that smallness encompasses sets of arbitrary finite cardinality, and that the category has at least two arrows, so that we can show the existence of an object with two global elements. If this fails to be the case, then the situation is somewhat pathological and not very interesting.

locale *sets-cat-with-bool* =
sets-cat sml C +

small-finite sml
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*) +
assumes *embeds-bool-ax*: *embeds* (*UNIV* :: *bool set*)
begin

definition *two* (**2**)
where *two* ≡ *mkide* {*True*, *False*}

lemma *ide-two* [*intro*, *simp*]:
shows *ide two*
and *bij-betw* (*IN* {*True*, *False*}) *UNIV* (*Set two*)
and *bij-betw* (*OUT* {*True*, *False*}) (*Set two*) *UNIV*
⟨*proof*⟩

definition *tt*
where *tt* ≡ *IN* {*True*, *False*} *True*

definition *ff*
where *ff* ≡ *IN* {*True*, *False*} *False*

lemma *tt-in-hom* [*intro*]:
shows «*tt* : **1**[?] → **2**»
⟨*proof*⟩

lemma *ff-in-hom* [*intro*]:
shows «*ff* : **1**[?] → **2**»
⟨*proof*⟩

lemma *tt-simps* [*simp*]:
shows *arr tt* **and** *dom tt* = **1**[?] **and** *cod tt* = **2**
⟨*proof*⟩

lemma *ff-simps* [*simp*]:
shows *arr ff* **and** *dom ff* = **1**[?] **and** *cod ff* = **2**
⟨*proof*⟩

lemma *Fun-tt*:
shows *Fun tt* = (λ*x*. *if x* ∈ *Set* **1**[?] *then tt else null*)
⟨*proof*⟩

lemma *Fun-ff*:
shows *Fun ff* = (λ*x*. *if x* ∈ *Set* **1**[?] *then ff else null*)
⟨*proof*⟩

lemma *mono-tt*:
shows *mono tt*
⟨*proof*⟩

lemma *mono-ff*:
shows *mono ff*
⟨*proof*⟩

lemma *tt-ne-ff*:
shows $tt \neq ff$
⟨*proof*⟩

lemma *Set-two*:
shows *Set* **2** = {*tt*, *ff*}
⟨*proof*⟩

In the present context, an arrow is epi if and only if the corresponding function is surjective. It follows that every epimorphism splits.

lemma *epi-char*$_{SCB}$:
shows *epi f* ⟷ *arr f* ∧ *Fun f ' Set* (*dom f*) = *Set* (*cod f*)
⟨*proof*⟩

corollary *epis-split*:
assumes *epi e*
shows ∃ *m*. *e* · *m* = *cod e*
⟨*proof*⟩

end

4.5 Equalizers

In this section we show that the category of small sets and functions has equalizers of parallel pairs of arrows. This is our first example of a general pattern that we will apply repeatedly in the sequel to other categorical constructions. Given a parallel pair *f*, *g* of arrows in a category of sets, we know that the global elements of the domain of the equalizer will be in bijection with the set *E* of global elements *x* of *dom f* such that *f* · *x* = *g* · *x*. So, we obtain this set, which in this case happens already to be a small subset of the set of arrows of the category, and we obtain the corresponding object *mkide E*, which will be the domain of the equalizer. This part of the proof uses the smallness of *E* and the fact that it embeds in (actually, is a subset of) the set of arrows of the category. Once we have shown the existence of the object *mkide E*, we can apply *mkarr* to the inclusion of *Set* (*mkide e*) in *Set* (*dom f*) to obtain the equalizing arrow itself. Showing that this arrow has the necessary universal property requires reasoning about the comparison maps between *E* and *Set* (*mkide e*), but once that has been accomplished we are left simply with a universal property that does not mention these maps.

The construction and proofs here are simpler than for the other constructions we will consider, because the set *E* to which we apply *mkide* is already a subset of the collection of arrows of the category – in particular it is at the same type. This means that the smallness and embedding property required for the application of *mkide* holds automatically, without any further assumptions. In general, though, a set to which we

wish to apply *mkide* will not be a subset of the set of arrows, nor will it even be at the same type, so it will be necessary to reason about an encoding that embeds the elements of this set into the set of arrows of the category.

locale *equalizers-in-sets-cat* =
sets-cat
begin

abbreviation *Dom-equ*
where $Dom\text{-}equ\ f\ g \equiv \{x.\ x \in Set\ (dom\ f) \wedge f \cdot x = g \cdot x\}$

definition *dom-equ*
where $dom\text{-}equ\ f\ g \equiv mkide\ (Dom\text{-}equ\ f\ g)$

abbreviation *Equ*
where $Equ\ f\ g \equiv \lambda x.\ if\ x \in Set\ (dom\text{-}equ\ f\ g)\ then\ OUT\ (Dom\text{-}equ\ f\ g)\ x\ else\ null$

definition *equ*
where $equ\ f\ g \equiv mkarr\ (dom\text{-}equ\ f\ g)\ (dom\ f)\ (Equ\ f\ g)$

It is useful to include convenience facts about OUT and IN in the following, so that we can avoid having to deal with the smallness and embedding conditions elsewhere.

lemma *ide-dom-equ*:
assumes *par f g*
shows *ide* (*dom-equ f g*)
and *bij-betw* (*OUT* (*Dom-equ f g*)) (*Set* (*dom-equ f g*)) (*Dom-equ f g*)
and *bij-betw* (*IN* (*Dom-equ f g*)) (*Dom-equ f g*) (*Set* (*dom-equ f g*))
and $\bigwedge x.\ x \in Set\ (dom\text{-}equ\ f\ g) \Longrightarrow OUT\ (Dom\text{-}equ\ f\ g)\ x \in Set\ (dom\ f)$
and $\bigwedge y.\ y \in Dom\text{-}equ\ f\ g \Longrightarrow IN\ (Dom\text{-}equ\ f\ g)\ y \in Set\ (dom\text{-}equ\ f\ g)$
and $\bigwedge x.\ x \in Set\ (dom\text{-}equ\ f\ g) \Longrightarrow IN\ (Dom\text{-}equ\ f\ g)\ (OUT\ (Dom\text{-}equ\ f\ g)\ x) = x$
and $\bigwedge y.\ y \in Dom\text{-}equ\ f\ g \Longrightarrow OUT\ (Dom\text{-}equ\ f\ g)\ (IN\ (Dom\text{-}equ\ f\ g)\ y) = y$
⟨*proof*⟩

lemma *Equ-in-Hom* [*intro*]:
assumes *par f g*
shows *Equ f g* ∈ *Hom* (*dom-equ f g*) (*dom f*)
⟨*proof*⟩

lemma *equ-in-hom* [*intro*, *simp*]:
assumes *par f g*
shows «*equ f g* : *dom-equ f g* → *dom f*»
⟨*proof*⟩

lemma *equ-simps* [*simp*]:
assumes *par f g*
shows *arr* (*equ f g*) **and** *dom* (*equ f g*) = *dom-equ f g* **and** *cod* (*equ f g*) = *dom f*
⟨*proof*⟩

lemma *Fun-equ*:
assumes *par f g*

shows *Fun* (*equ f g*) = *Equ f g*
⟨*proof*⟩

lemma *equ-equalizes*:
assumes *par f g*
shows *f* · *equ f g* = *g* · *equ f g*
⟨*proof*⟩

lemma *equ-is-equalizer*:
assumes *par f g*
shows *has-as-equalizer f g* (*equ f g*)
⟨*proof*⟩

lemma *has-equalizers*:
assumes *par f g*
shows ∃ *e*. *has-as-equalizer f g e*
⟨*proof*⟩

end

4.5.1 Exported Notions

As we don't want to clutter the *sets-cat* locale with auxiliary definitions and facts that no longer need to be used once we have completed the equalizer construction, we have carried out the construction in a separate locale and we now transfer to the *sets-cat* locale only those definitions and facts that we would like to export. In general, we will need to export the objects and arrows mentioned by the universal property together with the associated infrastructure for establishing the types of expressions that use them. We will also need to export facts that allow us to externalize these arrows as functions between sets of global elements, and we will need facts that give the types and inverse relationship between the comparison maps.

context *sets-cat*
begin

interpretation *Equ*: *equalizers-in-sets-cat sml C* ⟨*proof*⟩

abbreviation *equ*
where *equ* ≡ *Equ.equ*

abbreviation *Equ*
where *Equ f g* ≡ {*x*. *x* ∈ *Set* (*dom f*) ∧ *f* · *x* = *g* · *x*}

lemma *equalizer-comparison-map-props*:
assumes *par f g*
shows *bij-betw* (*OUT* (*Equ f g*)) (*Set* (*dom* (*equ f g*))) (*Equ f g*)
and *bij-betw* (*IN* (*Equ f g*)) (*Equ f g*) (*Set* (*dom* (*equ f g*)))
and ⋀*x*. *x* ∈ *Set* (*dom* (*equ f g*)) ⟹ *OUT* (*Equ f g*) *x* ∈ *Set* (*dom f*)
and ⋀*y*. *y* ∈ *Equ f g* ⟹ *IN* (*Equ f g*) *y* ∈ *Set* (*dom* (*equ f g*))

and $\bigwedge x.\ x \in Set\ (dom\ (equ\ f\ g)) \Longrightarrow IN\ (Equ\ f\ g)\ (OUT\ (Equ\ f\ g)\ x) = x$
and $\bigwedge y.\ y \in Equ\ f\ g \Longrightarrow OUT\ (Equ\ f\ g)\ (IN\ (Equ\ f\ g)\ y) = y$
⟨*proof*⟩

lemma *equ-is-equalizer*:
assumes *par f g*
shows *has-as-equalizer f g* (*equ f g*)
⟨*proof*⟩

lemma *Fun-equ*:
assumes *par f g*
shows *Fun* (*equ f g*) = ($\lambda x.$ *if* $x \in Set\ (dom\ (equ\ f\ g))$
then OUT $\{x.\ x \in Set\ (dom\ f) \wedge f \cdot x = g \cdot x\}\ x$
else null)
⟨*proof*⟩

lemma *has-equalizers*:
assumes *par f g*
shows $\exists\, e.$ *has-as-equalizer f g e*
⟨*proof*⟩

end

4.6 Binary Products

In this section we show that the category of small sets and functions has binary products. We follow the same pattern as for equalizers, except that now the set to which we would like to apply *mkide* to obtain a product object will consist of pairs of arrows, rather than individual arrows. This means that we will need to assume the existence of a pairing function that embeds the set of pairs of arrows of the category back into the original set of arrows. Once again, in showing that the construction makes sense we will need to reason about comparison maps, but once this is done we will be left simply with a universal property which does not mention these maps. After that, we only have to work with the comparison maps when relating notions internal to the category to notions external to it.

The following locale specializes *sets-cat* by adding the assumption that there exists a suitable pairing function. In addition, we need to assume that the smallness notion being used is respected by pairing.

locale *sets-cat-with-pairing* =
sets-cat sml C +
small-product sml +
pairing ‹*Collect arr*›
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)

As previously, we carry out the details of the construction in an auxiliary locale and later transfer to the *sets-cat* locale only those things that we want to export.

locale *products-in-sets-cat* =
sets-cat-with-pairing sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

lemma *small-product-set*:
assumes *ide a* **and** *ide b*
shows *small* (*Set a* × *Set b*)
⟨*proof*⟩

lemma *embeds-product-sets*:
assumes *ide a* **and** *ide b*
shows *embeds* (*Set a* × *Set b*)
⟨*proof*⟩

We define the product of two objects as the object determined by the cartesian product of their sets of elements.

definition $prod_o$
where $prod_o\ a\ b \equiv mkide\ (Set\ a \times Set\ b)$

lemma *ide-prod*$_o$:
assumes *ide a* **and** *ide b*
shows *ide* ($prod_o\ a\ b$)
and *bij-betw* (*OUT* (*Set a* × *Set b*)) (*Set* ($prod_o\ a\ b$)) (*Set a* × *Set b*)
and *bij-betw* (*IN* (*Set a* × *Set b*)) (*Set a* × *Set b*) (*Set* ($prod_o\ a\ b$))
and $\bigwedge x.\ x \in Set\ (prod_o\ a\ b) \Longrightarrow OUT\ (Set\ a \times Set\ b)\ x \in Set\ a \times Set\ b$
and $\bigwedge y.\ y \in Set\ a \times Set\ b \Longrightarrow IN\ (Set\ a \times Set\ b)\ y \in Set\ (prod_o\ a\ b)$
and $\bigwedge x.\ x \in Set\ (prod_o\ a\ b) \Longrightarrow IN\ (Set\ a \times Set\ b)\ (OUT\ (Set\ a \times Set\ b)\ x) = x$
and $\bigwedge y.\ y \in Set\ a \times Set\ b \Longrightarrow OUT\ (Set\ a \times Set\ b)\ (IN\ (Set\ a \times Set\ b)\ y) = y$
⟨*proof*⟩

We next define the projection arrows from a product object in terms of the projection functions on the underlying cartesian product of sets.

abbreviation P_0 :: *'U* ⇒ *'U* ⇒ *'U* ⇒ *'U*
where $P_0\ a\ b \equiv \lambda x.\ if\ x \in Set\ (prod_o\ a\ b)\ then\ snd\ (OUT\ (Set\ a \times Set\ b)\ x)\ else\ null$

abbreviation P_1 :: *'U* ⇒ *'U* ⇒ *'U* ⇒ *'U*
where $P_1\ a\ b \equiv \lambda x.\ if\ x \in Set\ (prod_o\ a\ b)\ then\ fst\ (OUT\ (Set\ a \times Set\ b)\ x)\ else\ null$

lemma P_0*-in-Hom*:
assumes *ide a* **and** *ide b*
shows $P_0\ a\ b \in Hom\ (prod_o\ a\ b)\ b$
⟨*proof*⟩

lemma P_1*-in-Hom*:
assumes *ide a* **and** *ide b*
shows $P_1\ a\ b \in Hom\ (prod_o\ a\ b)\ a$
⟨*proof*⟩

definition $pr_0 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $pr_0\ a\ b \equiv mkarr\ (prod_o\ a\ b)\ b\ (P_0\ a\ b)$

definition $pr_1 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $pr_1\ a\ b \equiv mkarr\ (prod_o\ a\ b)\ a\ (P_1\ a\ b)$

lemma *pr-in-hom* [*intro*]:
assumes *ide a* **and** *ide b*
shows *in-hom* $(pr_1\ a\ b)\ (prod_o\ a\ b)\ a$
and *in-hom* $(pr_0\ a\ b)\ (prod_o\ a\ b)\ b$
⟨*proof*⟩

lemma *pr-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* $(pr_0\ a\ b)$ **and** *dom* $(pr_0\ a\ b) = prod_o\ a\ b$ **and** *cod* $(pr_0\ a\ b) = b$
and *arr* $(pr_1\ a\ b)$ **and** *dom* $(pr_1\ a\ b) = prod_o\ a\ b$ **and** *cod* $(pr_1\ a\ b) = a$
⟨*proof*⟩

lemma *Fun-pr*:
assumes *ide a* **and** *ide b*
shows *Fun* $(pr_1\ a\ b) = P_1\ a\ b$
and *Fun* $(pr_0\ a\ b) = P_0\ a\ b$
⟨*proof*⟩

Tupling of arrows is also defined in terms of the underlying cartesian product.

definition *Tuple* :: $'U \Rightarrow 'U \Rightarrow 'U \Rightarrow 'U$
where *Tuple f g* $\equiv$ $(\lambda x.$ *if* $x \in Set\ (dom\ f)$
then IN $(Set\ (cod\ f) \times Set\ (cod\ g))$ $(Fun\ f\ x,\ Fun\ g\ x)$
else null$)$

definition *tuple* :: $'U \Rightarrow 'U \Rightarrow 'U$
where *tuple f g* $\equiv mkarr\ (dom\ f)\ (prod_o\ (cod\ f)\ (cod\ g))\ (Tuple\ f\ g)$

lemma *tuple-in-hom* [*intro*]:
assumes «$f : c \to a$» **and** «$g : c \to b$»
shows «*tuple f g* : $c \to prod_o\ a\ b$»
⟨*proof*⟩

lemma *tuple-simps* [*simp*]:
assumes *span f g*
shows *arr* $(tuple\ f\ g)$
and *dom* $(tuple\ f\ g) = dom\ f$
and *cod* $(tuple\ f\ g) = prod_o\ (cod\ f)\ (cod\ g)$
⟨*proof*⟩

In verifying the equations required for a categorical product, we unfortunately do have to fuss with the comparison maps.

lemma *comp-pr-tuple*:

assumes *span f g*
shows $pr_1\ (cod\ f)\ (cod\ g) \cdot tuple\ f\ g = f$
and $pr_0\ (cod\ f)\ (cod\ g) \cdot tuple\ f\ g = g$
⟨*proof*⟩

lemma *Fun-tuple*:
assumes *span f g*
shows *Fun* (*tuple f g*) =
(λ*x*. *if* $x \in Set\ (dom\ f)$
then IN (*Set* (*cod f*) × *Set* (*cod g*)) (*Fun f x*, *Fun g x*)
else null)
⟨*proof*⟩

lemma *binary-product-pr*:
assumes *ide a* **and** *ide b*
shows *binary-product C a b* (pr_1 *a b*) (pr_0 *a b*)
⟨*proof*⟩

lemma *has-binary-products*:
shows *has-binary-products*
⟨*proof*⟩

end

4.6.1 Exported Notions

We now transfer to the *sets-cat-with-pairing* locale just the things we want to export. The projections are the main thing; most of the rest is inherited from the *elementary-category-with-binary-products* locale. We also need to include some infrastucture for moving in and out of the category and working with the comparison maps.

context *sets-cat-with-pairing*
begin

interpretation *Products*: *products-in-sets-cat* ⟨*proof*⟩

abbreviation $pr_0 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $pr_0 \equiv Products.pr_0$

abbreviation $pr_1 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $pr_1 \equiv Products.pr_1$

sublocale *elementary-category-with-binary-products C* pr_0 pr_1
⟨*proof*⟩

lemma *bin-prod-comparison-map-props*:
assumes *ide a* **and** *ide b*
shows $OUT\ (Set\ a \times Set\ b) \in Set\ (prod\ a\ b) \to Set\ a \times Set\ b$
and $IN\ (Set\ a \times Set\ b) \in Set\ a \times Set\ b \to Set\ (prod\ a\ b)$
and $\bigwedge x.\ x \in Set\ (prod\ a\ b) \Longrightarrow IN\ (Set\ a \times Set\ b)\ (OUT\ (Set\ a \times Set\ b)\ x) = x$

and $\bigwedge y.\ y \in Set\ a \times Set\ b \Longrightarrow OUT\ (Set\ a \times Set\ b)\ (IN\ (Set\ a \times Set\ b)\ y) = y$
and *bij-betw* $(OUT\ (Set\ a \times Set\ b))\ (Set\ (prod\ a\ b))\ (Set\ a \times Set\ b)$
and *bij-betw* $(IN\ (Set\ a \times Set\ b))\ (Set\ a \times Set\ b)\ (Set\ (prod\ a\ b))$
⟨*proof*⟩

lemma *Fun-pr*$_0$:
assumes *ide a* **and** *ide b*
shows $Fun\ (pr_0\ a\ b) = Products.P_0\ a\ b$
⟨*proof*⟩

lemma *Fun-pr*$_1$:
assumes *ide a* **and** *ide b*
shows $Fun\ (pr_1\ a\ b) = Products.P_1\ a\ b$
⟨*proof*⟩

lemma *Fun-prod*:
assumes «$f : a \to b$» **and** «$g : c \to d$»
shows $Fun\ (prod\ f\ g) = (\lambda x.\ if\ x \in Set\ (prod\ a\ c)$
$then\ tuple\ (Fun\ f\ (C\ (pr_1\ a\ c)\ x))\ (Fun\ g\ (C\ (pr_0\ a\ c)\ x))$
$else\ null)$
⟨*proof*⟩

lemma *prod-ide-eq*:
assumes *ide a* **and** *ide b*
shows $prod\ a\ b = mkide\ (Set\ a \times Set\ b)$
⟨*proof*⟩

lemma *tuple-eq*:
assumes «$f : x \to a$» **and** «$g : x \to b$»
shows $tuple\ f\ g = mkarr\ x\ (prod\ a\ b)$
$(\lambda z.\ if\ z \in Set\ x$
$then\ IN\ (Set\ a \times Set\ b)\ (Fun\ f\ z,\ Fun\ g\ z)$
$else\ null)$
⟨*proof*⟩

lemma *tuple-point-eq*:
assumes «$x : \mathbf{1}^? \to a$» **and** «$y : \mathbf{1}^? \to b$»
shows $tuple\ x\ y = IN\ (Set\ a \times Set\ b)\ (x,\ y)$
⟨*proof*⟩

lemma *Fun-tuple*:
assumes *span f g*
shows $Fun\ (tuple\ f\ g) =$
$(\lambda x.\ if\ x \in Set\ (dom\ f)$
$then\ IN\ (Set\ (cod\ f) \times Set\ (cod\ g))\ (Fun\ f\ x,\ Fun\ g\ x)$
$else\ null)$
⟨*proof*⟩

end

4.7 Binary Coproducts

In this section we prove the existence of binary coproducts, following the same approach as for binary products. The required assumptions are slightly different, because here we need smallness to be preserved by union.

locale *sets-cat-with-cotupling* =
  sets-cat-with-bool sml C +
  small-sum sml +
  pairing ‹*Collect arr*›
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)

locale *coproducts-in-sets-cat* =
  sets-cat-with-cotupling sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

  abbreviation *Coprod*
  where *Coprod a b* ≡ ({*tt*} × *Set a*) ∪ ({*ff*} × *Set b*)

  lemma *small-Coprod*:
  assumes *ide a* **and** *ide b*
  shows *small* (*Coprod a b*)
    ⟨*proof*⟩

  lemma *embeds-Coprod*:
  assumes *ide a* **and** *ide b*
  shows *embeds* (*Coprod a b*)
  ⟨*proof*⟩

  definition $coprod_o$
  where $coprod_o$ *a b* ≡ *mkide* (*Coprod a b*)

  lemma $ide\text{-}coprod_o$:
  assumes *ide a* **and** *ide b*
  shows *ide* ($coprod_o$ *a b*)
  and *bij-betw* (*OUT* (*Coprod a b*)) (*Set* ($coprod_o$ *a b*)) (*Coprod a b*)
  and *bij-betw* (*IN* (*Coprod a b*)) (*Coprod a b*) (*Set* ($coprod_o$ *a b*))
  and ⋀*x*. *x* ∈ *Set* ($coprod_o$ *a b*) ⟹ *OUT* (*Coprod a b*) *x* ∈ *Coprod a b*
  and ⋀*y*. *y* ∈ *Coprod a b* ⟹ *IN* (*Coprod a b*) *y* ∈ *Set* ($coprod_o$ *a b*)
  and ⋀*x*. *x* ∈ *Set* ($coprod_o$ *a b*) ⟹ *IN* (*Coprod a b*) (*OUT* (*Coprod a b*) *x*) = *x*
  and ⋀*y*. *y* ∈ *Coprod a b* ⟹ *OUT* (*Coprod a b*) (*IN* (*Coprod a b*) *y*) = *y*
  ⟨*proof*⟩

  abbreviation In_0 :: *'U* ⇒ *'U* ⇒ *'U* ⇒ *'U*
  where In_0 *a b* ≡ λ*x*. *if x* ∈ *Set b then IN* (*Coprod a b*) (*ff*, *x*) *else null*

  abbreviation In_1 :: *'U* ⇒ *'U* ⇒ *'U* ⇒ *'U*

where $In_1\ a\ b \equiv \lambda x.\ \mathit{if}\ x \in \mathit{Set}\ a\ \mathit{then}\ \mathit{IN}\ (\mathit{Coprod}\ a\ b)\ (\mathit{tt},\ x)\ \mathit{else}\ \mathit{null}$

lemma *In₀-in-Hom*:
assumes *ide a* **and** *ide b*
shows $In_0\ a\ b \in \mathit{Hom}\ b\ (\mathit{coprod}_o\ a\ b)$
⟨*proof*⟩

lemma *In₁-in-Hom*:
assumes *ide a* **and** *ide b*
shows $In_1\ a\ b \in \mathit{Hom}\ a\ (\mathit{coprod}_o\ a\ b)$
⟨*proof*⟩

definition $in_0 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $in_0\ a\ b \equiv \mathit{mkarr}\ b\ (\mathit{coprod}_o\ a\ b)\ (In_0\ a\ b)$

definition $in_1 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $in_1\ a\ b \equiv \mathit{mkarr}\ a\ (\mathit{coprod}_o\ a\ b)\ (In_1\ a\ b)$

lemma *in-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *in-hom* $(in_1\ a\ b)\ a\ (\mathit{coprod}_o\ a\ b)$
and *in-hom* $(in_0\ a\ b)\ b\ (\mathit{coprod}_o\ a\ b)$
⟨*proof*⟩

lemma *in-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* $(in_0\ a\ b)$ **and** *dom* $(in_0\ a\ b) = b$ **and** *cod* $(in_0\ a\ b) = \mathit{coprod}_o\ a\ b$
and *arr* $(in_1\ a\ b)$ **and** *dom* $(in_1\ a\ b) = a$ **and** *cod* $(in_1\ a\ b) = \mathit{coprod}_o\ a\ b$
⟨*proof*⟩

lemma *Fun-in*:
assumes *ide a* **and** *ide b*
shows *Fun* $(in_1\ a\ b) = In_1\ a\ b$
and *Fun* $(in_0\ a\ b) = In_0\ a\ b$
⟨*proof*⟩

definition *Cotuple* $:: 'U \Rightarrow 'U \Rightarrow 'U \Rightarrow 'U$
where $\mathit{Cotuple}\ f\ g \equiv (\lambda x.\ \mathit{if}\ x \in \mathit{Set}\ (\mathit{coprod}_o\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))$
$\quad \mathit{then}\ \mathit{if}\ \mathit{fst}\ (\mathit{OUT}\ (\mathit{Coprod}\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))\ x) = \mathit{tt}$
$\quad\quad \mathit{then}\ \mathit{Fun}\ f\ (\mathit{snd}\ (\mathit{OUT}\ (\mathit{Coprod}\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))\ x))$
$\quad\quad \mathit{else}\ \mathit{if}\ \mathit{fst}\ (\mathit{OUT}\ (\mathit{Coprod}\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))\ x) = \mathit{ff}$
$\quad\quad\quad \mathit{then}\ \mathit{Fun}\ g\ (\mathit{snd}\ (\mathit{OUT}\ (\mathit{Coprod}\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))\ x))$
$\quad\quad\quad \mathit{else}\ \mathit{null}$
$\quad \mathit{else}\ \mathit{null})$

definition *cotuple* $:: 'U \Rightarrow 'U \Rightarrow 'U$
where $\mathit{cotuple}\ f\ g \equiv \mathit{mkarr}\ (\mathit{coprod}_o\ (\mathit{dom}\ f)\ (\mathit{dom}\ g))\ (\mathit{cod}\ f)\ (\mathit{Cotuple}\ f\ g)$

lemma *cotuple-in-hom* [*intro*, *simp*]:

assumes «$f : a \to c$» **and** «$g : b \to c$»
shows «$cotuple\ f\ g : coprod_o\ a\ b \to c$»
⟨*proof*⟩

lemma *cotuple-simps* [*simp*]:
assumes *cospan f g*
shows *arr* (*cotuple f g*)
and *dom* (*cotuple f g*) = $coprod_o$ (*dom f*) (*dom g*)
and *cod* (*cotuple f g*) = *cod f*
⟨*proof*⟩

lemma *comp-cotuple-in*:
assumes *cospan f g*
shows *cotuple f g* · in_1 (*dom f*) (*dom g*) = *f*
and *cotuple f g* · in_0 (*dom f*) (*dom g*) = *g*
⟨*proof*⟩

lemma *Fun-cotuple*:
assumes *cospan f g*
shows *Fun* (*cotuple f g*) =
(λ*x*. *if x* ∈ *Set* ($coprod_o$ (*dom f*) (*dom g*))
then if fst (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*) = *tt*
then Fun f (*snd* (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*))
else if fst (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*) = *ff*
then Fun g (*snd* (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*))
else null
else null)
⟨*proof*⟩

lemma *binary-coproduct-in*:
assumes *ide a* **and** *ide b*
shows *binary-product* (*dual-category.comp C*) *a b* (in_1 *a b*) (in_0 *a b*)
⟨*proof*⟩

lemma *has-binary-coproducts*:
shows *category.has-binary-products* (*dual-category.comp C*)
⟨*proof*⟩

end

4.7.1 Exported Notions

context *sets-cat-with-cotupling*
begin

interpretation *Coproducts*: *coproducts-in-sets-cat* ⟨*proof*⟩

abbreviation in_0 :: *'U* ⇒ *'U* ⇒ *'U*
where in_0 ≡ *Coproducts.*in_0

abbreviation $in_1 :: 'U \Rightarrow 'U \Rightarrow 'U$
where $in_1 \equiv Coproducts.in_1$

abbreviation $Coprod :: 'U \Rightarrow 'U \Rightarrow ('U \times 'U)\ set$
where $Coprod \equiv Coproducts.Coprod$

abbreviation $coprod_o :: 'U \Rightarrow 'U \Rightarrow 'U$
where $coprod_o \equiv Coproducts.coprod_o$

lemma *ide-coprod$_o$*:
assumes *ide a* **and** *ide b*
shows *ide* (*coprod$_o$ a b*)
⟨*proof*⟩

lemma *in$_1$-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *in-hom* (*in$_1$ a b*) *a* (*coprod$_o$ a b*)
⟨*proof*⟩

lemma *in$_0$-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *in-hom* (*in$_0$ a b*) *b* (*coprod$_o$ a b*)
⟨*proof*⟩

lemma *in$_1$-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* (*in$_1$ a b*) **and** *dom* (*in$_1$ a b*) = *a* **and** *cod* (*in$_1$ a b*) = *coprod$_o$ a b*
⟨*proof*⟩

lemma *in$_0$-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* (*in$_0$ a b*) **and** *dom* (*in$_0$ a b*) = *b* **and** *cod* (*in$_0$ a b*) = *coprod$_o$ a b*
⟨*proof*⟩

lemma *bin-coprod-comparison-map-props*:
assumes *ide a* **and** *ide b*
shows *bij-betw* (*OUT* (*Coprod a b*)) (*Set* (*coprod$_o$ a b*)) (*Coprod a b*)
and *bij-betw* (*IN* (*Coprod a b*)) (*Coprod a b*) (*Set* (*coprod$_o$ a b*))
and $\bigwedge x.\ x \in Set\ (coprod_o\ a\ b) \Longrightarrow OUT\ (Coprod\ a\ b)\ x \in Coprod\ a\ b$
and $\bigwedge y.\ y \in Coprod\ a\ b \Longrightarrow IN\ (Coprod\ a\ b)\ y \in Set\ (coprod_o\ a\ b)$
and $\bigwedge x.\ x \in Set\ (coprod_o\ a\ b) \Longrightarrow IN\ (Coprod\ a\ b)\ (OUT\ (Coprod\ a\ b)\ x) = x$
and $\bigwedge y.\ y \in Coprod\ a\ b \Longrightarrow OUT\ (Coprod\ a\ b)\ (IN\ (Coprod\ a\ b)\ y) = y$
⟨*proof*⟩

lemma *Fun-in$_1$*:
assumes *ide a* **and** *ide b*
shows *Fun* (*in$_1$ a b*) = *Coproducts.In$_1$ a b*
⟨*proof*⟩

lemma *Fun-in*$_0$:
assumes *ide a* **and** *ide b*
shows *Fun* (*in*$_0$ *a b*) = *Coproducts.In*$_0$ *a b*
⟨*proof*⟩

abbreviation *cotuple*
where *cotuple* ≡ *Coproducts.cotuple*

lemma *cotuple-in-hom* [*intro*, *simp*]:
assumes «*f* : *a* → *c*» **and** «*g* : *b* → *c*»
shows «*cotuple f g* : *coprod*$_o$ *a b* → *c*»
⟨*proof*⟩

lemma *cotuple-simps* [*simp*]:
assumes *cospan f g*
shows *arr* (*cotuple f g*)
and *dom* (*cotuple f g*) = *coprod*$_o$ (*dom f*) (*dom g*)
and *cod* (*cotuple f g*) = *cod f*
⟨*proof*⟩

abbreviation *Cotuple*
where *Cotuple f g* ≡ (λ*x*. *if x* ∈ *Set* (*coprod*$_o$ (*dom f*) (*dom g*))
then if fst (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*) = *tt*
then Fun f (*snd* (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*))
else if fst (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*) = *ff*
then Fun g (*snd* (*OUT* (*Coprod* (*dom f*) (*dom g*)) *x*))
else null
else null)

lemma *cotuple-eq*:
assumes «*f* : *a* → *c*» **and** «*g* : *b* → *c*»
shows *cotuple f g* = *mkarr* (*coprod*$_o$ *a b*) *c* (*Cotuple f g*)
⟨*proof*⟩

lemma *Fun-cotuple*:
assumes *cospan f g*
shows *Fun* (*cotuple f g*) = *Cotuple f g*
⟨*proof*⟩

lemma *binary-coproduct-in*:
assumes *ide a* **and** *ide b*
shows *binary-product* (*dual-category.comp C*) *a b* (*in*$_1$ *a b*) (*in*$_0$ *a b*)
⟨*proof*⟩

lemma *has-binary-coproducts*:
shows *category.has-binary-products* (*dual-category.comp C*)
⟨*proof*⟩

end

4.8 Small Products

In this section we show that the category of small sets and functions has small products. For this we need to assume that smallness is preserved by the formation of function spaces.

locale *sets-cat-with-tupling* =
  sets-cat sml C +
  tupling sml ‹*Collect arr*› *null*
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

  sublocale *sets-cat-with-bool*
    ⟨*proof*⟩
  sublocale *sets-cat-with-pairing sml C* ⟨*proof*⟩
  sublocale *sets-cat-with-cotupling* ⟨*proof*⟩

end

locale *small-products-in-sets-cat* =
  sets-cat-with-tupling sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

A product diagram is specified by an extensional function A from small index set I to *Collect ide*, using *null* as the default value. An element of the product is given by an extensional function F from I to *Collect arr*, such that $F\ i \in Set\ (A\ i)$ for each $i \in I$.

  abbreviation *ProdX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* ⇒ *'U*) *set*
  where *ProdX I A* ≡ {*F*. ∀ *i*. *i* ∈ *I* ⟶ *F i* ∈ *Set* (*A i*)} ∩ {*F*. ∀ *i*. *i* ∉ *I* ⟶ *F i* = *null*}

  lemma *ProdX-empty*:
  shows *ProdX* {} *A* = {λ*x*. *null*}
    ⟨*proof*⟩

  definition *prodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
  where *prodX I A* ≡ *mkide* (*ProdX I A*)

  lemma *small-function-tuple*:
  assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
  and *F* ∈ *ProdX I A*
  shows *small-function F* **and** *range F* ⊆ (⋃ *i*∈*I*. *Set* (*A i*)) ∪ {*null*}
  ⟨*proof*⟩

  lemma *small-ProdX*:
  assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*

shows *small* (*ProdX I A*)
⟨*proof*⟩

lemma *embeds-ProdX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
shows *embeds* (*ProdX I A*)
⟨*proof*⟩

lemma *ide-prodX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
shows *ide* (*prodX I A*)
and *bij-betw* (*OUT* (*ProdX I A*)) (*Set* (*prodX I A*)) (*ProdX I A*)
and *bij-betw* (*IN* (*ProdX I A*)) (*ProdX I A*) (*Set* (*prodX I A*))
and $\bigwedge x.\ x \in$ *Set* (*prodX I A*) $\Longrightarrow$ *OUT* (*ProdX I A*) $x \in$ *ProdX I A*
and $\bigwedge y.\ y \in$ *ProdX I A* $\Longrightarrow$ *IN* (*ProdX I A*) $y \in$ *Set* (*prodX I A*)
and $\bigwedge x.\ x \in$ *Set* (*prodX I A*) $\Longrightarrow$ *IN* (*ProdX I A*) (*OUT* (*ProdX I A*) x) $= x$
and $\bigwedge y.\ y \in$ *ProdX I A* $\Longrightarrow$ *OUT* (*ProdX I A*) (*IN* (*ProdX I A*) y) $= y$
⟨*proof*⟩

lemma *terminal-prodX-empty*:
shows *terminal* (*prodX* {} ($A :: 'U \Rightarrow 'U$))
⟨*proof*⟩

abbreviation $PrX :: 'a$ *set* $\Rightarrow ('a \Rightarrow 'U) \Rightarrow 'a \Rightarrow 'U \Rightarrow 'U$
where *PrX I A i* $\equiv \lambda x.$ *if* $x \in$ *Set* (*prodX I A*) *then OUT* (*ProdX I A*) *x i else null*

definition $prX :: 'a$ *set* $\Rightarrow ('a \Rightarrow 'U) \Rightarrow 'a \Rightarrow 'U$
where *prX I A i* $\equiv$ *mkarr* (*prodX I A*) (*A i*) (*PrX I A i*)

lemma *prX-in-hom* [*intro*, *simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *in-hom* (*prX I A i*) (*prodX I A*) (*A i*)
⟨*proof*⟩

lemma *prX-simps* [*simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *arr* (*prX I A i*) **and** *dom* (*prX I A i*) $=$ *prodX I A* **and** *cod* (*prX I A i*) $= A\ i$
⟨*proof*⟩

lemma *Fun-prX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *Fun* (*prX I A i*) $=$ *PrX I A i*
⟨*proof*⟩

definition $TupleX :: 'a$ *set* $\Rightarrow 'U \Rightarrow ('a \Rightarrow 'U) \Rightarrow ('a \Rightarrow 'U) \Rightarrow 'U \Rightarrow 'U$
where *TupleX I c A F* $\equiv (\lambda x.$ *if* $x \in$ *Set c then IN* (*ProdX I A*) ($\lambda i.$ *Fun* (*F i*) x) *else null*)

lemma *TupleX-in-Hom*:
assumes *small I* **and** $A \in I \to Collect\ ide$ **and** $I \subseteq Collect\ arr$
and $\bigwedge i.\ i \in I \Longrightarrow \guillemotleft F\ i : c \to A\ i\guillemotright$ **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$
shows *TupleX I c A F* $\in$ *Hom c* (*prodX I A*)
⟨*proof*⟩

definition *tupleX* :: *'a set* ⇒ *'U* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *tupleX I c A F* ≡ *mkarr c* (*prodX I A*) (*TupleX I c A F*)

lemma *tupleX-in-hom* [*intro*, *simp*]:
assumes *small I* **and** $A \in I \to Collect\ ide$ **and** $I \subseteq Collect\ arr$
and $\bigwedge i.\ i \in I \Longrightarrow \guillemotleft F\ i : c \to A\ i\guillemotright$ **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows «*tupleX I c A F* : *c* → *prodX I A*»
⟨*proof*⟩

lemma *tupleX-simps* [*simp*]:
assumes *small I* **and** $A \in I \to Collect\ ide$ **and** $I \subseteq Collect\ arr$
and $\bigwedge i.\ i \in I \Longrightarrow \guillemotleft F\ i : c \to A\ i\guillemotright$ **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows *arr* (*tupleX I c A F*)
and *dom* (*tupleX I c A F*) = *c*
and *cod* (*tupleX I c A F*) = *prodX I A*
⟨*proof*⟩

lemma *comp-prX-tupleX*:
assumes *small I* **and** $A \in I \to Collect\ ide$ **and** $I \subseteq Collect\ arr$
and $\bigwedge i.\ i \in I \Longrightarrow \guillemotleft F\ i : c \to A\ i\guillemotright$ **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$
shows $i \in I \Longrightarrow C\ (prX\ I\ A\ i)\ (tupleX\ I\ c\ A\ F) = F\ i$
⟨*proof*⟩

lemma *Fun-tupleX*:
assumes *small I* **and** $A \in I \to Collect\ ide$ **and** $I \subseteq Collect\ arr$
and $\bigwedge i.\ i \in I \Longrightarrow \guillemotleft F\ i : c \to A\ i\guillemotright$ **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows *Fun* (*tupleX I c A F*) =
(λ*x*. *if x* ∈ *Set c then IN* (*ProdX I A*) (λ*i*. *Fun* (*F i*) *x*) *else null*)
⟨*proof*⟩

lemma *product-cone-prodX*:
assumes *discrete-diagram J C D* **and** *Collect* (*partial-composition.arr J*) = *I*
and *small I* **and** *I* ⊆ *Collect arr*
shows *has-as-product J D* (*prodX I D*)
and *product-cone J C D* (*prodX I D*) (*prX I D*)
⟨*proof*⟩

lemma *has-small-products*:
assumes *small I* **and** *I* ⊆ *Collect arr*
shows *has-products I*
⟨*proof*⟩

end

4.8.1 Exported Notions

context *sets-cat-with-tupling*
begin

interpretation *Products*: *small-products-in-sets-cat* ⟨*proof*⟩

abbreviation *ProdX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* ⇒ *'U*) *set*
where *ProdX* ≡ *Products.ProdX*

abbreviation *prodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *prodX* ≡ *Products.prodX*

abbreviation *prX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'a* ⇒ *'U*
where *prX* ≡ *Products.prX*

abbreviation *tupleX* :: *'a set* ⇒ *'U* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *tupleX* ≡ *Products.tupleX*

lemma *small-prod-comparison-map-props*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
shows *OUT* (*ProdX I A*) ∈ *Set* (*prodX I A*) → *ProdX I A*
and *IN* (*ProdX I A*) ∈ *ProdX I A* → *Set* (*prodX I A*)
and ⋀*x*. *x* ∈ *Set* (*prodX I A*) ⟹ *IN* (*ProdX I A*) (*OUT* (*ProdX I A*) *x*) = *x*
and ⋀*y*. *y* ∈ *ProdX I A* ⟹ *OUT* (*ProdX I A*) (*IN* (*ProdX I A*) *y*) = *y*
and *bij-betw* (*OUT* (*ProdX I A*)) (*Set* (*prodX I A*)) (*ProdX I A*)
and *bij-betw* (*IN* (*ProdX I A*)) (*ProdX I A*) (*Set* (*prodX I A*))
⟨*proof*⟩

lemma *Fun-prX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
and *i* ∈ *I*
shows *Fun* (*prX I A i*) = *Products.PrX I A i*
⟨*proof*⟩

lemma *Fun-tupleX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
and ⋀*i*. *i* ∈ *I* ⟹ «*F i* : *c* → *A i*» **and** ⋀*i*. *i* ∉ *I* ⟹ *F i* = *null* **and** *ide c*
shows *Fun* (*tupleX I c A F*) =
(λ*x*. *if x* ∈ *Set c then IN* (*Products.ProdX I A*) (λ*i*. *Fun* (*F i*) *x*) *else null*)
⟨*proof*⟩

lemma *product-cone*:
assumes *discrete-diagram J C D* **and** *Collect* (*partial-composition.arr J*) = *I*
and *small I* **and** *I* ⊆ *Collect arr*
shows *has-as-product J D* (*prodX I D*)
and *product-cone J C D* (*prodX I D*) (*prX I D*)

⟨*proof*⟩

lemma *has-small-products*:
assumes *small I* **and** $I \subseteq$ *Collect arr*
shows *has-products I*
⟨*proof*⟩

Clearly it is not required that the index set I be actually a subset of *Collect arr* but rather only that it be embedded in it. So we are free to form products indexed by small sets at arbitrary types, as long as *Collect arr* is large enough to embed them. We do have to satisfy the technical requirement that the index set I not exhaust the elements at its type, which we introduced in the definition of *has-products* as a convenience to avoid the use of coercion maps.

lemma *has-small-products'*:
assumes *small I* **and** *embeds I* **and** $I \neq$ *UNIV*
shows *has-products I*
⟨*proof*⟩

end

4.9 Small Coproducts

In this section we show that the category of small sets and functions has small coproducts. For this we need to assume the existence of a pairing function and also that the notion of smallness is respected by small sums.

locale *small-coproducts-in-sets-cat* =
sets-cat-with-cotupling sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

The global elements of a coproduct *CoprodX I A* are in bijection with $\bigcup i{\in}I.\ \{i\} \times$ *Set* (*A i*).

abbreviation *CoprodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* × *'U*) *set*
where *CoprodX I A* ≡ $\bigcup i{\in}I.\ \{i\} \times$ *Set* (*A i*)

definition *coprodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *coprodX I A* ≡ *mkide* (*CoprodX I A*)

lemma *small-CoprodX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
shows *small* (*CoprodX I A*)
⟨*proof*⟩

lemma *embeds-CoprodX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
shows *embeds* (*CoprodX I A*)

⟨*proof*⟩

lemma *ide-coprodX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
shows *ide* (*coprodX I A*)
and *bij-betw* (*OUT* (*CoprodX I A*)) (*Set* (*coprodX I A*)) (*CoprodX I A*)
and *bij-betw* (*IN* (*CoprodX I A*)) (*CoprodX I A*) (*Set* (*coprodX I A*))
and $\bigwedge x.\ x \in$ *Set* (*coprodX I A*) $\Longrightarrow$ *OUT* (*CoprodX I A*) $x \in$ *CoprodX I A*
and $\bigwedge y.\ y \in$ *CoprodX I A* $\Longrightarrow$ *IN* (*CoprodX I A*) $y \in$ *Set* (*coprodX I A*)
and $\bigwedge x.\ x \in$ *Set* (*coprodX I A*) $\Longrightarrow$ *IN* (*CoprodX I A*) (*OUT* (*CoprodX I A*) x) $= x$
and $\bigwedge y.\ y \in$ *CoprodX I A* $\Longrightarrow$ *OUT* (*CoprodX I A*) (*IN* (*CoprodX I A*) y) $= y$
⟨*proof*⟩

abbreviation *InX* :: $'a$ *set* $\Rightarrow$ ($'a \Rightarrow 'U$) $\Rightarrow$ $'a \Rightarrow 'U \Rightarrow 'U$
where *InX I A i* $\equiv \lambda x.$ *if* $x \in$ *Set* (*A i*) *then IN* (*CoprodX I A*) (*i*, *x*) *else null*

definition *inX*
where *inX I A i* $\equiv$ *mkarr* (*A i*) (*coprodX I A*) (*InX I A i*)

lemma *InX-in-Hom*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *InX I A i* $\in$ *Hom* (*A i*) (*coprodX I A*)
⟨*proof*⟩

lemma *inX-in-hom* [*intro*, *simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *in-hom* (*inX I A i*) (*A i*) (*coprodX I A*)
⟨*proof*⟩

lemma *inX-simps* [*simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *arr* (*inX I A i*) **and** *dom* (*inX I A i*) $=$ *A i* **and** *cod* (*inX I A i*) $=$ *coprodX I A*
⟨*proof*⟩

lemma *Fun-inX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $i \in I$
shows *Fun* (*inX I A i*) $=$ *InX I A i*
⟨*proof*⟩

definition *CotupleX* :: $'a$ *set* $\Rightarrow$ ($'a \Rightarrow 'U$) $\Rightarrow$ ($'a \Rightarrow 'U$) $\Rightarrow$ $'U \Rightarrow 'U$
where *CotupleX I A F* $\equiv$
(*λx. if x* $\in$ *Set* (*coprodX I A*)
then Fun (*F* (*fst* (*OUT* (*CoprodX I A*) *x*))) (*snd* (*OUT* (*CoprodX I A*) *x*))
else null)

lemma *CotupleX-in-Hom*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $\bigwedge i.\ i \in I \Longrightarrow$ «$F\ i : A\ i \to c$» **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$
shows *CotupleX I A F* $\in$ *Hom* (*coprodX I A*) *c*
⟨*proof*⟩

definition *cotupleX*
where *cotupleX I c A F* ≡ *mkarr* (*coprodX I A*) *c* (*CotupleX I A F*)

lemma *cotupleX-in-hom* [*intro*, *simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $\bigwedge i.\ i \in I \Longrightarrow$ «$F\ i : A\ i \to c$» **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows «*cotupleX I c A F* : *coprodX I A* → *c*»
⟨*proof*⟩

lemma *cotupleX-simps* [*simp*]:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $\bigwedge i.\ i \in I \Longrightarrow$ «$F\ i : A\ i \to c$» **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows *arr* (*cotupleX I c A F*)
and *dom* (*cotupleX I c A F*) = *coprodX I A*
and *cod* (*cotupleX I c A F*) = *c*
⟨*proof*⟩

lemma *comp-cotupleX-inX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $\bigwedge i.\ i \in I \Longrightarrow$ «$F\ i : A\ i \to c$» **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows $i \in I \Longrightarrow$ *cotupleX I c A F* · *inX I A i* = *F i*
⟨*proof*⟩

lemma *Fun-cotupleX*:
assumes *small I* **and** $A \in I \to$ *Collect ide* **and** $I \subseteq$ *Collect arr*
and $\bigwedge i.\ i \in I \Longrightarrow$ «$F\ i : A\ i \to c$» **and** $\bigwedge i.\ i \notin I \Longrightarrow F\ i = null$ **and** *ide c*
shows *Fun* (*cotupleX I c A F*) =
(λ*x*. *if x* ∈ *Set* (*coprodX I A*)
then Fun (*F* (*fst* (*OUT* (*CoprodX I A*) *x*))) (*snd* (*OUT* (*CoprodX I A*) *x*))
else null)
⟨*proof*⟩

lemma *coproduct-cocone-coprodX*:
assumes *discrete-diagram J C D* **and** *Collect* (*partial-composition.arr J*) = *I*
and *small I* **and** *I* ⊆ *Collect arr*
shows *has-as-coproduct J D* (*coprodX I D*)
and *coproduct-cocone J C D* (*coprodX I D*) (*inX I D*)
⟨*proof*⟩

lemma *has-small-coproducts*:
assumes *small I* **and** *I* ⊆ *Collect arr*
shows *has-coproducts I*
⟨*proof*⟩

end

4.9.1 Exported Notions

context *sets-cat-with-cotupling*
begin

interpretation *Coproducts*: *small-coproducts-in-sets-cat* ⟨*proof*⟩

abbreviation *CoprodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* × *'U*) *set*
where *CoprodX* ≡ *Coproducts.CoprodX*

abbreviation *coprodX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *coprodX* ≡ *Coproducts.coprodX*

abbreviation *inX* :: *'a set* ⇒ (*'a* ⇒ *'U*) ⇒ *'a* ⇒ *'U*
where *inX* ≡ *Coproducts.inX*

abbreviation *cotupleX* :: *'a set* ⇒ *'U* ⇒ (*'a* ⇒ *'U*) ⇒ (*'a* ⇒ *'U*) ⇒ *'U*
where *cotupleX* ≡ *Coproducts.cotupleX*

lemma *coprod-comparison-map-props*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
shows *OUT* (*CoprodX I A*) ∈ *Set* (*coprodX I A*) → *CoprodX I A*
and *IN* (*CoprodX I A*) ∈ *CoprodX I A* → *Set* (*coprodX I A*)
and ⋀*x*. *x* ∈ *Set* (*coprodX I A*) ⟹ *IN* (*CoprodX I A*) (*OUT* (*CoprodX I A*) *x*) = *x*
and ⋀*y*. *y* ∈ *CoprodX I A* ⟹ *OUT* (*CoprodX I A*) (*IN* (*CoprodX I A*) *y*) = *y*
and *bij-betw* (*OUT* (*CoprodX I A*)) (*Set* (*coprodX I A*)) (*CoprodX I A*)
and *bij-betw* (*IN* (*CoprodX I A*)) (*CoprodX I A*) (*Set* (*coprodX I A*))
⟨*proof*⟩

lemma *Fun-inX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
and *i* ∈ *I*
shows *Fun* (*inX I A i*) = *Coproducts.InX I A i*
⟨*proof*⟩

lemma *Fun-cotupleX*:
assumes *small I* **and** *A* ∈ *I* → *Collect ide* **and** *I* ⊆ *Collect arr*
and ⋀*i*. *i* ∈ *I* ⟹ «*F i* : *A i* → *c*» **and** ⋀*i*. *i* ∉ *I* ⟹ *F i* = *null* **and** *ide c*
shows *Fun* (*cotupleX I c A F*) =
(λ*x*. *if x* ∈ *Set* (*coprodX I A*)
then Fun (*F* (*fst* (*OUT* (⋃*i*∈*I*. {*i*} × *Set* (*A i*)) *x*)))
(*snd* (*OUT* (⋃*i*∈*I*. {*i*} × *Set* (*A i*)) *x*))
else null)
⟨*proof*⟩

lemma *coproduct-cocone-coprodX*:

assumes *discrete-diagram J C D* **and** *Collect* (*partial-composition.arr J*) = *I*
and *small I* **and** *I* ⊆ *Collect arr*
shows *has-as-coproduct J D* (*coprodX I D*)
and *coproduct-cocone J C D* (*coprodX I D*) (*inX I D*)
⟨*proof*⟩

lemma *has-small-coproducts*:
assumes *small I* **and** *I* ⊆ *Collect arr*
shows *has-coproducts I*
⟨*proof*⟩

end

4.10 Coequalizers

In this section we show that a sets category has coequalizers of parallel pairs of arrows. For this, we need to assume that the set of arrows of the category embeds the set of all its small subsets. The reason we need this assumption is to make it possible to obtain an object corresponding to the set of equivalence classes that results from the quotient construction.

locale *sets-cat-with-powering* =
sets-cat sml C +
powering sml ‹*Collect arr*›
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)

sublocale *sets-cat-with-tupling* ⊆ *sets-cat-with-powering* ⟨*proof*⟩

locale *coequalizers-in-sets-cat* =
sets-cat-with-powering sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

The following defines the "equivalence closure" of a binary relation r on a set A, and proves the characterization of it as the least equivalence relation on A that contains r. For some reason I could not find such a thing in the Isabelle distribution, though I did find a predicate version *equivclp*.

definition *equivcl*
where *equivcl A r* ≡ *SOME r'*. *r* ⊆ *r'* ∧ *equiv A r'* ∧ (∀ *s'*. *r* ⊆ *s'* ∧ *equiv A s'* ⟶ *r'* ⊆ *s'*)

lemma *equivcl-props*:
assumes *r* ⊆ *A* × *A*
shows ∃ *r'*. *r* ⊆ *r'* ∧ *equiv A r'* ∧ (∀ *s'*. *r* ⊆ *s'* ∧ *equiv A s'* ⟶ *r'* ⊆ *s'*)
and *r* ⊆ *equivcl A r* **and** *equiv A* (*equivcl A r*)
and ⋀*s'*. *r* ⊆ *s'* ∧ *equiv A s'* ⟹ *equivcl A r* ⊆ *s'*
⟨*proof*⟩

The elements of the codomain of the coequalizer of f and g are the equivalence classes of the least equivalence relation on $Set\ (cod\ f)$ that relates $f \cdot x$ and $g \cdot x$ whenever $x \in Set\ (dom\ f)$.

abbreviation *Cod-coeq* :: $'U \Rightarrow 'U \Rightarrow 'U\ set\ set$
where $Cod\text{-}coeq\ f\ g \equiv (\lambda y.\ (equivcl\ (Set\ (cod\ f))$
$((\lambda x.\ (f \cdot x,\ g \cdot x))\ `\ Set\ (dom\ f))\ ``\ \{y\}))\ `\ Set\ (cod\ f)$

lemma *small-Cod-coeq*:
assumes *par f g*
shows *small* (*Cod-coeq f g*)
⟨*proof*⟩

lemma *embeds-Cod-coeq*:
assumes *par f g*
shows *embeds* (*Cod-coeq f g*)
and $Cod\text{-}coeq\ f\ g \subseteq Pow\ (Set\ (cod\ f))$
⟨*proof*⟩

definition *cod-coeq*
where *cod-coeq f g* ≡ *mkide* (*Cod-coeq f g*)

lemma *ide-cod-coeq*:
assumes *par f g*
shows *ide* (*cod-coeq f g*)
and *bij-betw* (*OUT* (*Cod-coeq f g*)) (*Set* (*cod-coeq f g*)) (*Cod-coeq f g*)
and *bij-betw* (*IN* (*Cod-coeq f g*)) (*Cod-coeq f g*) (*Set* (*cod-coeq f g*))
and $\bigwedge x.\ x \in Set\ (cod\text{-}coeq\ f\ g) \Longrightarrow OUT\ (Cod\text{-}coeq\ f\ g)\ x \in Cod\text{-}coeq\ f\ g$
and $\bigwedge y.\ y \in Cod\text{-}coeq\ f\ g \Longrightarrow IN\ (Cod\text{-}coeq\ f\ g)\ y \in Set\ (cod\text{-}coeq\ f\ g)$
and $\bigwedge x.\ x \in Set\ (cod\text{-}coeq\ f\ g) \Longrightarrow IN\ (Cod\text{-}coeq\ f\ g)\ (OUT\ (Cod\text{-}coeq\ f\ g)\ x) = x$
and $\bigwedge y.\ y \in Cod\text{-}coeq\ f\ g \Longrightarrow OUT\ (Cod\text{-}coeq\ f\ g)\ (IN\ (Cod\text{-}coeq\ f\ g)\ y) = y$
⟨*proof*⟩

definition *Coeq*
where $Coeq\ f\ g \equiv \lambda y.\ if\ y \in Set\ (cod\ f)$
$then\ IN\ (Cod\text{-}coeq\ f\ g)$
$(equivcl\ (Set\ (cod\ f))$
$((\lambda x.\ (f \cdot x,\ g \cdot x))\ `\ Set\ (dom\ f))\ ``\ \{y\})$
$else\ null$

lemma *Coeq-in-Hom* [*intro*]:
assumes *par f g*
shows $Coeq\ f\ g \in Hom\ (cod\ f)\ (cod\text{-}coeq\ f\ g)$
⟨*proof*⟩

definition *coeq*
where *coeq f g* ≡ *mkarr* (*cod f*) (*cod-coeq f g*) (*Coeq f g*)

lemma *coeq-in-hom* [*intro*, *simp*]:
assumes *par f g*

shows «*coeq f g* : *cod f* → *cod-coeq f g*»
⟨*proof*⟩

lemma *coeq-simps* [*simp*]:
assumes *par f g*
shows *arr* (*coeq f g*) **and** *dom* (*coeq f g*) = *cod f* **and** *cod* (*coeq f g*) = *cod-coeq f g*
⟨*proof*⟩

lemma *Fun-coeq*:
assumes *par f g*
shows *Fun* (*coeq f g*) = *Coeq f g*
⟨*proof*⟩

lemma *coeq-coequalizes*:
assumes *par f g*
shows *coeq f g* · *f* = *coeq f g* · *g*
⟨*proof*⟩

lemma *Coeq-surj*:
assumes *par f g* **and** *Set* (*cod f*) ≠ {} **and** *y* ∈ *Set* (*cod-coeq f g*)
shows ∃ *x*. *x* ∈ *Set* (*cod f*) ∧ *Coeq f g x* = *y*
⟨*proof*⟩

lemma *coeq-is-coequalizer*:
assumes *par f g* **and** *Set* (*cod f*) ≠ {}
shows *has-as-coequalizer f g* (*coeq f g*)
⟨*proof*⟩

lemma *has-coequalizers*:
assumes *par f g*
shows ∃ *e*. *has-as-coequalizer f g e*
⟨*proof*⟩

end

4.10.1 Exported Notions

context *sets-cat-with-powering*
begin

interpretation *Coeq*: *coequalizers-in-sets-cat sml C* ⟨*proof*⟩

abbreviation *Cod-coeq*
where *Cod-coeq* ≡ *Coeq.Cod-coeq*

abbreviation *coeq*
where *coeq* ≡ *Coeq.coeq*

lemma *coequalizer-comparison-map-props*:

assumes *par f g*
shows *bij-betw* (*OUT* (*Cod-coeq f g*)) (*Set* (*cod* (*coeq f g*))) (*Cod-coeq f g*)
and *bij-betw* (*IN* (*Cod-coeq f g*)) (*Cod-coeq f g*) (*Set* (*cod* (*coeq f g*)))
and $\bigwedge x.\ x \in Set\ (cod\ (coeq\ f\ g)) \Longrightarrow OUT\ (Cod\text{-}coeq\ f\ g)\ x \in Cod\text{-}coeq\ f\ g$
and $\bigwedge y.\ y \in Cod\text{-}coeq\ f\ g \Longrightarrow IN\ (Cod\text{-}coeq\ f\ g)\ y \in Set\ (cod\ (coeq\ f\ g))$
and $\bigwedge x.\ x \in Set\ (cod\ (coeq\ f\ g)) \Longrightarrow IN\ (Cod\text{-}coeq\ f\ g)\ (OUT\ (Cod\text{-}coeq\ f\ g)\ x) = x$
and $\bigwedge y.\ y \in Cod\text{-}coeq\ f\ g \Longrightarrow OUT\ (Cod\text{-}coeq\ f\ g)\ (IN\ (Cod\text{-}coeq\ f\ g)\ y) = y$
⟨*proof*⟩

lemma *coeq-is-coequalizer*:
assumes *par f g* **and** $Set\ (cod\ f) \neq \{\}$
shows *has-as-coequalizer f g* (*coeq f g*)
⟨*proof*⟩

Since the fact *Fun-coeq* below is not very useful without the notions used in stating it, the function *equivcl* and characteristic fact *equivcl-props* are also exported here. It would be better if *Fun-coeq* could be expressed completely in terms of existing notions from the library.

definition *equivcl*
where *equivcl* ≡ *Coeq.equivcl*

lemma *equivcl-props*:
assumes $r \subseteq A \times A$
shows $\exists r'.\ r \subseteq r' \wedge equiv\ A\ r' \wedge (\forall s'.\ r \subseteq s' \wedge equiv\ A\ s' \longrightarrow r' \subseteq s')$
and $r \subseteq equivcl\ A\ r$ **and** *equiv A* (*equivcl A r*)
and $\bigwedge s'.\ r \subseteq s' \wedge equiv\ A\ s' \Longrightarrow equivcl\ A\ r \subseteq s'$
⟨*proof*⟩

lemma *Fun-coeq*:
assumes *par f g*
shows *Fun* (*coeq f g*) = (λ*y*. *if* $y \in Set\ (cod\ f)$
then IN (*Cod-coeq f g*)
(*equivcl* (*Set* (*cod f*))
((λ*x*. (*f* · *x*, *g* · *x*)) ‘ *Set* (*dom f*)) ‘‘ {*y*})
else null)
⟨*proof*⟩

lemma *has-coequalizers*:
assumes *par f g*
shows ∃ *e*. *has-as-coequalizer f g e*
⟨*proof*⟩

end

4.11 Exponentials

In this section we show that the category is cartesian closed.

locale *exponentials-in-sets-cat* =

sets-cat-with-tupling sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

abbreviation *app* :: *'U* ⇒ *'U* ⇒ *'U*
where *app f* ≡ *inv-into SEF some-embedding-of-small-functions f*

abbreviation *Exp* :: *'U* ⇒ *'U* ⇒ (*'U* ⇒ *'U*) *set*
where *Exp a b* ≡ {*F*. *F* ∈ *Set a* → *Set b* ∧ (∀ *x*. *x* ∉ *Set a* ⟶ *F x* = *null*)}

definition *exp* :: *'U* ⇒ *'U* ⇒ *'U*
where *exp a b* ≡ *mkide* (*Exp a b*)

lemma *memb-Exp-popular-value*:
assumes *ide a* **and** *ide b* **and** *F* ∈ *Exp a b*
and *popular-value F y*
shows *y* = *null*
⟨*proof*⟩

lemma *memb-Exp-imp-small-function*:
assumes *ide a* **and** *ide b* **and** *F* ∈ *Exp a b*
shows *small-function F*
⟨*proof*⟩

lemma *small-Exp*:
assumes *ide a* **and** *ide b*
shows *small* (*Exp a b*)
⟨*proof*⟩

lemma *embeds-Exp*:
assumes *ide a* **and** *ide b*
shows *embeds* (*Exp a b*)
⟨*proof*⟩

lemma *ide-exp*:
assumes *ide a* **and** *ide b*
shows *ide* (*exp a b*)
and *bij-betw* (*OUT* (*Exp a b*)) (*Set* (*exp a b*)) (*Exp a b*)
and *bij-betw* (*IN* (*Exp a b*)) (*Exp a b*) (*Set* (*exp a b*))
⟨*proof*⟩

abbreviation *Eval*
where *Eval b c* ≡ (λ*fx*. *if fx* ∈ *Set* (*prod* (*exp b c*) *b*)
then OUT (*Exp b c*)
(*Fun* (*pr*$_1$ (*exp b c*) *b*) *fx*)
(*Fun* (*pr*$_0$ (*exp b c*) *b*) *fx*)
else null)

definition *eval*
where *eval b c* ≡ *mkarr* (*prod* (*exp b c*) *b*) *c* (*Eval b c*)

lemma *eval-in-hom* [*intro*, *simp*]:
assumes *ide b* **and** *ide c*
shows «*eval b c* : *prod* (*exp b c*) *b* → *c*»
⟨*proof*⟩

lemma *eval-simps* [*simp*]:
assumes *ide b* **and** *ide c*
shows *arr* (*eval b c*) **and** *dom* (*eval b c*) = *prod* (*exp b c*) *b* **and** *cod* (*eval b c*) = *c*
⟨*proof*⟩

lemma *Fun-eval*:
assumes *ide b* **and** *ide c*
shows *Fun* (*eval b c*) = *Eval b c*
⟨*proof*⟩

definition *Curry*
where *Curry a b c* ≡ λ*f*. *if* «*f* : *prod a b* → *c*»
then mkarr a (*exp b c*)
(λ*x*. *if x* ∈ *Set a*
then IN (*Exp b c*)
(λ*y*. *if y* ∈ *Set b*
then C f (*tuple x y*)
else null)
else null)
else null

lemma *Curry-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and «*f* : *prod a b* → *c*»
shows «*Curry a b c f* : *a* → *exp b c*»
and *Fun* (*Curry a b c f*) =
(λ*x*. *if x* ∈ *Set a*
then IN (*Exp b c*) (λ*y*. *if y* ∈ *Set b then C f* (*tuple x y*) *else null*)
else null)
⟨*proof*⟩

lemma *Curry-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and «*f* : *prod a b* → *c*»
shows *arr* (*Curry a b c f*) **and** *dom* (*Curry a b c f*) = *a* **and** *cod* (*Curry a b c f*) = *exp b c*
⟨*proof*⟩

lemma *Fun-Curry*:
assumes *ide a* **and** *ide b* **and** *ide c*
and «*f* : *prod a b* → *c*»
shows *Fun* (*Curry a b c f*) =

(λx. *if* x ∈ *Set* a
then IN (*Exp* b c) (λy. *if* y ∈ *Set* b *then* C f (*tuple* x y) *else null*)
else null)
⟨*proof*⟩

interpretation *elementary-category-with-terminal-object* C ‹$\mathbf{1}^?$› *some-terminator*
⟨*proof*⟩

lemma *is-category-with-terminal-object*:
shows *elementary-category-with-terminal-object* C $\mathbf{1}^?$ *some-terminator*
and *category-with-terminal-object* C
⟨*proof*⟩

interpretation *elementary-cartesian-closed-category*
C pr_0 pr_1 ‹$\mathbf{1}^?$› *some-terminator exp eval Curry*
⟨*proof*⟩

lemma *is-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category* C pr_0 pr_1 $\mathbf{1}^?$ *some-terminator exp eval Curry*
⟨*proof*⟩

lemma *is-cartesian-closed-category*:
shows *cartesian-closed-category* C
⟨*proof*⟩

end

4.11.1 Exported Notions

context *sets-cat-with-tupling*
begin

sublocale *sets-cat-with-pairing* ⟨*proof*⟩

interpretation *Expos*: *exponentials-in-sets-cat sml* C ⟨*proof*⟩

abbreviation *Exp*
where *Exp* ≡ *Expos.Exp*

abbreviation *exp*
where *exp* ≡ *Expos.exp*

lemma *ide-exp*:
assumes *ide* a **and** *ide* b
shows *ide* (*exp* a b)
⟨*proof*⟩

lemma *exp-comparison-map-props*:
assumes *ide* a **and** *ide* b

shows $OUT\ (Exp\ a\ b) \in Set\ (exp\ a\ b) \to Exp\ a\ b$
and $IN\ (Exp\ a\ b) \in Exp\ a\ b \to Set\ (exp\ a\ b)$
and $\bigwedge x.\ x \in Set\ (exp\ a\ b) \Longrightarrow IN\ (Exp\ a\ b)\ (OUT\ (Exp\ a\ b)\ x) = x$
and $\bigwedge y.\ y \in Exp\ a\ b \Longrightarrow OUT\ (Exp\ a\ b)\ (IN\ (Exp\ a\ b)\ y) = y$
and *bij-betw* $(OUT\ (Exp\ a\ b))\ (Set\ (exp\ a\ b))\ (Exp\ a\ b)$
and *bij-betw* $(IN\ (Exp\ a\ b))\ (Exp\ a\ b)\ (Set\ (exp\ a\ b))$
⟨*proof*⟩

abbreviation *Eval*
where $Eval \equiv Expos.Eval$

abbreviation *eval*
where $eval \equiv Expos.eval$

lemma *eval-in-hom* [*intro*, *simp*]:
assumes *ide b* **and** *ide c*
shows «$eval\ b\ c : prod\ (exp\ b\ c)\ b \to c$»
⟨*proof*⟩

lemma *eval-simps* [*simp*]:
assumes *ide b* **and** *ide c*
shows *arr* $(eval\ b\ c)$ **and** $dom\ (eval\ b\ c) = prod\ (exp\ b\ c)\ b$ **and** $cod\ (eval\ b\ c) = c$
⟨*proof*⟩

lemma *Fun-eval*:
assumes *ide b* **and** *ide c*
shows $Fun\ (eval\ b\ c) = Eval\ b\ c$
⟨*proof*⟩

abbreviation *Curry*
where $Curry \equiv Expos.Curry$

lemma *Curry-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and «$f : prod\ a\ b \to c$»
shows «$Curry\ a\ b\ c\ f : a \to exp\ b\ c$»
⟨*proof*⟩

lemma *Curry-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and «$f : prod\ a\ b \to c$»
shows *arr* $(Curry\ a\ b\ c\ f)$
and $dom\ (Curry\ a\ b\ c\ f) = a$ **and** $cod\ (Curry\ a\ b\ c\ f) = exp\ b\ c$
⟨*proof*⟩

lemma *Fun-Curry*:
assumes *ide a* **and** *ide b* **and** *ide c*
and «$f : prod\ a\ b \to c$»
shows $Fun\ (Curry\ a\ b\ c\ f) =$

(λx. *if* x ∈ *Set a*
then IN (*Exp b c*) (λy. *if* y ∈ *Set b then C f* (*tuple x y*) *else null*)
else null)
⟨*proof*⟩

theorem *is-cartesian-closed*:
shows *elementary-cartesian-closed-category C* pr_0 pr_1 $\mathbf{1}^?$ *some-terminator exp eval Curry*
and *cartesian-closed-category C*
⟨*proof*⟩

end

4.12 Subobject Classifier

In this section we show that a sets category has a subobject classifier, which is a categorical formulation of set comprehension. We give here a formal definition of subobject classifier, because we have not done that elsewhere to date, but ultimately this definition would perhaps be better placed with a development of the theory of elementary topoi, which are cartesian closed categories with subobject classifier.

context *category*
begin

A subobject classifier is a monomorphism *tt* from a terminal object into an object Ω, which we may regard as an "object of truth values", such that for every monomorphism *m* there exists a unique arrow χ : *cod m* → Ω, such that *m* is given by the pullback of *tt* along χ.

definition *subobject-classifier*
where *subobject-classifier tt* ≡
mono tt ∧ *terminal* (*dom tt*) ∧
(∀ *m*. *mono m* ⟶
(∃!χ. «χ : *cod m* → *cod tt*» ∧
has-as-pullback tt χ (*THE f*. «*f* : *dom m* → *dom tt*») *m*))

lemma *subobject-classifierI* [*intro*]:
assumes «*tt* : *one* → Ω» **and** *terminal one* **and** *mono tt*
and ⋀*m*. *mono m* ⟹ ∃!χ. «χ : *cod m* → Ω» ∧
has-as-pullback tt χ (*THE f*. «*f* : *dom m* → *one*») *m*
shows *subobject-classifier tt*
⟨*proof*⟩

lemma *subobject-classifierE* [*elim*]:
assumes *subobject-classifier tt*
and ⟦*mono tt*; *terminal* (*dom tt*);
⋀*m*. *mono m* ⟹ ∃!χ. «χ : *cod m* → *cod tt*» ∧
has-as-pullback tt χ (*THE f*. «*f* : *dom m* → *dom tt*») *m*⟧
⟹ *T*
shows *T*

⟨*proof*⟩

end

locale *category-with-subobject-classifier* =
category +
assumes *has-subobject-classifier-ax*: ∃ *tt*. *subobject-classifier tt*
begin

sublocale *category-with-terminal-object*
⟨*proof*⟩

end

context *sets-cat-with-bool*
begin

For a sets category, the two-point object **2** (which exists in the current context *sets-cat-with-bool*) serves as the object of truth values. The subobject classifier will be the arrow $tt : \mathbf{1}^? \to \mathbf{2}$.

Here we define a mapping χ that takes a monomorphism m to a corresponding "predicate" $\chi\ m : cod\ m \to \mathbf{2}$.

abbreviation *Chi*
where *Chi m* ≡ λ*y*. *if y* ∈ *Set* (*cod m*)
then
if y ∈ *Fun m* ‘ *Set* (*dom m*) *then tt else ff*
else null

definition χ :: *'U* ⇒ *'U*
where χ *m* ≡ *mkarr* (*cod m*) **2** (*Chi m*)

lemma χ*-in-hom* [*intro*, *simp*]:
assumes «*m* : *b* → *a*» **and** *mono m*
shows «χ *m* : *a* → **2**»
⟨*proof*⟩

lemma χ*-simps* [*simp*]:
assumes «*m* : *b* → *a*» **and** *mono m*
shows *arr* (χ *m*) **and** *dom* (χ *m*) = *a* **and** *cod* (χ *m*) = **2**
⟨*proof*⟩

lemma *Fun-*χ:
assumes «*m* : *b* → *a*» **and** *mono m*
shows *Fun* (χ *m*) = *Chi m*
⟨*proof*⟩

lemma *bij-Fun-mono*:
assumes «*m* : *b* → *a*» **and** *mono m*
shows *bij-betw* (*Fun m*) (*Set b*) {*y*. *y* ∈ *Set a* ∧ χ *m* · *y* = *tt*}

⟨*proof*⟩

lemma *has-subobject-classifier*:
shows *subobject-classifier tt*
⟨*proof*⟩

sublocale *category-with-subobject-classifier*
⟨*proof*⟩

lemma *is-category-with-subobject-classifier*:
shows *category-with-subobject-classifier C*
⟨*proof*⟩

end

4.13 Natural Numbers Object

In this section we show that a sets category has a natural numbers object, assuming that the smallness notion is such that the set of natural numbers is small, and assuming that that the collection of arrows admits lifting, so that the category has infinitely many arrows.

locale *sets-cat-with-infinity* =
sets-cat sml C +
small-nat sml +
lifting ‹*Collect arr*›
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

abbreviation *nat* (**N**)
where *nat* ≡ *mkide* (*UNIV* :: *nat set*)

lemma *ide-nat*:
shows *ide* **N**
and *bij-betw* (*OUT* (*UNIV* :: *nat set*)) (*Set* **N**) (*UNIV* :: *nat set*)
and *bij-betw* (*IN* (*UNIV* :: *nat set*)) (*UNIV* :: *nat set*) (*Set* **N**)
⟨*proof*⟩

abbreviation *Zero*
where *Zero* ≡ λ*x*. *if x* ∈ *Set* **1**[?] *then IN* (*UNIV* :: *nat set*) *0 else null*

lemma *Zero-in-Hom*:
shows *Zero* ∈ *Hom* **1**[?] **N**
⟨*proof*⟩

definition *zero*
where *zero* ≡ *mkarr* **1**[?] **N** *Zero*

lemma *zero-in-hom* [*intro*, *simp*]:
shows «*zero* : $\mathbf{1}^? \to \mathbf{N}$»
⟨*proof*⟩

lemma *zero-simps* [*simp*]:
shows *arr zero* **and** *dom zero* = $\mathbf{1}^?$ **and** *cod zero* = **N**
⟨*proof*⟩

lemma *Fun-zero*:
shows *Fun zero* = *Zero*
⟨*proof*⟩

abbreviation *Succ*
where *Succ* ≡ λ*x*. *if x* ∈ *Set* **N** *then IN* (*UNIV* :: *nat set*) (*Suc* (*OUT UNIV x*)) *else null*

lemma *Succ-in-Hom*:
shows *Succ* ∈ *Hom* **N** **N**
⟨*proof*⟩

definition *succ*
where *succ* ≡ *mkarr* **N** **N** *Succ*

lemma *succ-in-hom* [*intro*]:
shows «*succ* : **N** → **N**»
⟨*proof*⟩

lemma *succ-simps* [*simp*]:
shows *arr succ* **and** *dom succ* = **N** **and** *cod succ* = **N**
⟨*proof*⟩

lemma *Fun-succ*:
shows *Fun succ* = *Succ*
⟨*proof*⟩

lemma *nat-universality*:
assumes «*Z* : $\mathbf{1}^? \to a$» **and** «*S* : *a* → *a*»
shows ∃!*f*. «*f* : **N** → *a*» ∧ *f* · *zero* = *Z* ∧ *f* · *succ* = *S* · *f*
⟨*proof*⟩

lemma *has-natural-numbers-object*:
shows ∃ *a z s*. «*z* : $\mathbf{1}^? \to a$» ∧ «*s* : *a* → *a*» ∧
(∀ *a′ z′ s′*. «*z′* : $\mathbf{1}^? \to a'$» ∧ «*s′* : *a′* → *a′*» ⟶
(∃!*f*. «*f* : *a* → *a′*» ∧ *f* · *z* = *z′* ∧ *f* · *s* = *s′* · *f*))
⟨*proof*⟩

end

4.14 Sets Category with Tupling and Infinity

Finally, if the collection of arrows of a sets category admits embeddings of all the usual set-theoretic constructions, then the category supports all of the constructions considered; in particular it is small-complete and small-cocomplete, is cartesian closed, has a subobject classifier (so that it is an elementary topos), and validates an axiom of infinity in the form of the existence of a natural numbers object.

context *sets-cat-with-tupling*
begin

lemmas *is-well-pointed epis-split has-binary-products has-binary-coproducts has-small-products has-small-coproducts has-equalizers has-coequalizers is-cartesian-closed has-subobject-classifier*

end

locale *sets-cat-with-tupling-and-infinity* =
sets-cat-with-tupling sml C +
sets-cat-with-infinity sml C
for *sml* :: *'V set* ⇒ *bool*
and *C* :: *'U comp* (**infixr** ‹·› *55*)
begin

sublocale *universe sml* ‹*Collect arr*› *null* ⟨*proof*⟩

lemmas *has-natural-numbers-object*

end

end

Chapter 5

Interpretations of *universe*

theory *Universe-Interps*
imports *Universe ZFC-in-HOL.ZFC-Cardinals*
begin

In this section we give two interpretations of locales defined in theory *Universe*. In one interpretation, "finite" is taken as the notion of smallness and the set of natural numbers is used to interpret the *tupling* locale. In the second interpretation, the notion "small" is as defined in *ZFC-in-HOL* and the set of elements of the type V defined in that theory is used as the universe. This interpretation interprets the *universe* locale, which augments *universe* with the assumption *small-nat* that the set of natural numbers is small. The purpose of constructing these interpretations is to show the consistency of the *universe* locale assumptions (relative, of course to the consistency of HOL itself, and of HOL as extended in *ZFC-in-HOL*), as well as to provide a starting point for the construction of large categories, such as the category of small sets which is treated in this article.

5.1 Interpretation using Natural Numbers

We first give an interpretation for the *tupling* locale, taking the set of natural numbers as the universe and taking "finite" as the meaning of "small".

context
begin

We first establish properties of *finite* :: *nat set* ⇒ *bool* as our notion of smallness.

interpretation *smallness* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

The notion *small* defined by the *smallness* locale agrees with the notion *finite* given as a locale parameter.

lemma *finset-small-iff-finite*:
shows *local.small X* ⟷ *finite X*
⟨*proof*⟩

interpretation *small-finite* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-finite-finset*:
shows *small-finite* (*finite* :: *nat set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-product* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-product-finset*:
shows *small-product* (*finite* :: *nat set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-sum* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-sum-finset*:
shows *small-sum* (*finite* :: *nat set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-powerset* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-powerset-finset*:
shows *small-powerset* (*finite* :: *nat set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-funcset* ‹*finite* :: *nat set* ⇒ *bool*› ⟨*proof*⟩

As expected, the assumptions of locale *small-nat* are inconsistent with the present context.

lemma *large-nat-finset*:
shows ¬ *local.small* (*UNIV* :: *nat set*)
⟨*proof*⟩

Next, we develop embedding properties of *UNIV* :: *nat set*.

interpretation *embedding* ‹*UNIV* :: *nat set*› ⟨*proof*⟩

interpretation *lifting* ‹*UNIV* :: *nat set*›
⟨*proof*⟩

lemma *nat-admits-lifting*:
shows *lifting* (*UNIV* :: *nat set*)
⟨*proof*⟩

interpretation *pairing* ‹*UNIV* :: *nat set*›
⟨*proof*⟩

lemma *nat-admits-pairing*:
shows *pairing* (*UNIV* :: *nat set*)
⟨*proof*⟩

interpretation *powering* ‹*finite* :: *nat set* ⇒ *bool*› ‹*UNIV* :: *nat set*›
⟨*proof*⟩

lemma *nat-admits-finite-powering*:
shows *powering* (*finite* :: *nat set* ⇒ *bool*) (*UNIV* :: *nat set*)
⟨*proof*⟩

interpretation *tupling* ‹*finite* :: *nat set* ⇒ *bool*› ‹*UNIV* :: *nat set*› ⟨*proof*⟩

lemma *nat-admits-finite-tupling*:
shows *tupling* (*finite* :: *nat set* ⇒ *bool*) (*UNIV* :: *nat set*)
⟨*proof*⟩

end

Finally, we give the interpretation of the *tupling* locale, stated in the top-level context in order to make it clear that it can be established directly in HOL, without depending somehow on any underlying locale assumptions.

interpretation *nat-tupling*: *tupling* ‹*finite* :: *nat set* ⇒ *bool*› ‹*UNIV* :: *nat set*› *undefined*
⟨*proof*⟩

5.2 Interpretation using *ZFC-in-HOL*

We now give an interpretation for the *universe* locale, taking as the universe the set of elements of type *V* defined in *ZFC-in-HOL* as the universe and using the notion *small* also defined in that theory.

context
begin

We first develop properties of *small*, which we take as our notion of smallness.

interpretation *smallness* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

The notion *small* defined by the *smallness* locale agrees with the notion *ZFC-in-HOL.small* given as a locale parameter.

lemma *small-iff-ZFC-small*:
shows *local.small X* ⟷ *ZFC-in-HOL.small X*
⟨*proof*⟩

interpretation *small-finite* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-finite-ZFC*:
shows *small-finite* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-product* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-product-ZFC*:
shows *small-product* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-sum* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-sum-ZFC*:
shows *small-sum* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

We need the following, which does not seem to be directly available in *ZFC-in-HOL*.

lemma *ZFC-small-implies-small-powerset*:
fixes *X*
assumes *ZFC-in-HOL.small X*
shows *ZFC-in-HOL.small* (*Pow X*)
⟨*proof*⟩

interpretation *small-powerset* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-powerset-ZFC*:
shows *small-powerset* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-funcset* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*› ⟨*proof*⟩

lemma *small-funcset-ZFC*:
shows *small-funcset* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-nat* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
⟨*proof*⟩

lemma *small-nat-ZFC*:
shows *small-nat* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)
⟨*proof*⟩

interpretation *small-funcset-and-nat* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*› ⟨*proof*⟩

lemma *small-funcset-and-nat-ZFC*:
shows *small-funcset-and-nat* (*ZFC-in-HOL.small* :: *V set* ⇒ *bool*)

⟨*proof*⟩

Next, we develop embedding properties of *UNIV* :: *V set*.

interpretation *embedding* ‹*UNIV* :: *V set*› ⟨*proof*⟩

interpretation *lifting* ‹*UNIV* :: *V set*›
⟨*proof*⟩

lemma *V-admits-lifting*:
shows *lifting* (*UNIV* :: *V set*)
⟨*proof*⟩

interpretation *pairing* ‹*UNIV* :: *V set*›
⟨*proof*⟩

lemma *V-admits-pairing*:
shows *pairing* (*UNIV* :: *V set*)
⟨*proof*⟩

interpretation *powering* ‹*ZFC-in-HOL.small* :: *V set* => *bool*› ‹*UNIV* :: *V set*›
⟨*proof*⟩

lemma *V-admits-small-powering*:
shows *powering* (*ZFC-in-HOL.small* :: *V set* => *bool*) (*UNIV* :: *V set*)
⟨*proof*⟩

interpretation *tupling* ‹*ZFC-in-HOL.small* :: *V set* => *bool*› ‹*UNIV* :: *V set*› *undefined*
⟨*proof*⟩

lemma *V-admits-small-tupling*:
shows *tupling* (*ZFC-in-HOL.small* :: *V set* => *bool*) (*UNIV* :: *V set*)
⟨*proof*⟩

interpretation *universe* ‹*ZFC-in-HOL.small* :: *V set* => *bool*› ‹*UNIV* :: *V set*› *undefined*
⟨*proof*⟩

theorem *V-is-universe*:
shows *universe* (*ZFC-in-HOL.small* :: *V set* => *bool*) (*UNIV* :: *V set*)
⟨*proof*⟩

end

Finally, we give the interpretation of the *universe* locale, stated in the top-level context. Note however, that this is proved not in "vanilla HOL", but rather in HOL as extended by the axiomatization in *ZFC-in-HOL*.

interpretation *ZFC-universe*: *universe* ‹*ZFC-in-HOL.small* :: *V set* => *bool*› ‹*UNIV* :: *V set*› *undefined*
⟨*proof*⟩

end

Chapter 6

Interpretations of *sets-cat*

theory *SetsCat-Interps*
imports *Category3.ConcreteCategory Category3.ZFC-SetCat Category3.Colimit*
SetsCat Universe-Interps
begin

In this section we construct two interpretations of the *sets-cat* locale: one using "finite" as the notion of smallness and one that uses *small* from the theory *ZFC-in-HOL*. These interpretations demonstrate the consistency of the variants of the *sets-cat* locale: the interpretation using finiteness validates the *sets-cat-with-tupling* locale in unextended HOL, and the interpretation in terms of *ZFC-in-HOL* validates the *sets-cat-with-tupling-and-infinity* locale, assuming that the axiomatization of *ZFC-in-HOL* is consistent with HOL.

6.1 Category of Finite Sets

The *finite-sets-cat* locale defines a category having as objects the natural numbers and as arrows from m to n the functions from m-element sets to n-element sets. In view of *SetsCat.categoricity*, this is the unique interpretation (up to equivalence of categories) of *sets-cat* having a countably infinite collection of arrows.

locale *finite-sets-cat*
begin

abbreviation *OBJ*
where $OBJ \equiv UNIV :: nat\ set$

abbreviation *HOM*
where $HOM \equiv \lambda m\ n.\ \{1..m :: nat\} \rightarrow_E \{1..n :: nat\}$

abbreviation *Id*
where $Id\ n \equiv \lambda x :: nat.\ if\ x \in \{1..n\}\ then\ x\ else\ undefined$

abbreviation *Comp*
where $Comp\ -\ -\ m \equiv compose\ \{1..m\}$

interpretation *Fin*: *concrete-category OBJ HOM Id Comp*
⟨*proof*⟩

abbreviation *comp*
where *comp* ≡ *Fin.COMP*

lemma *terminal-MkIde-1*:
shows *Fin.terminal* (*Fin.MkIde 1*)
⟨*proof*⟩

sublocale *category-with-terminal-object comp*
⟨*proof*⟩

notation *some-terminal* (**1**$^?$)

sublocale *sets-cat-base* ‹*finite* :: *nat set* ⇒ *bool*› *comp*
⟨*proof*⟩

sublocale *small-finite* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

sublocale *small-powerset* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

lemma *finite-HOM*:
shows *finite* (*HOM m n*)
⟨*proof*⟩

lemma *card-HOM*:
shows *card* (*HOM m n*) = *n* ^ *m*
⟨*proof*⟩

lemma *terminal-char*$_{FSC}$:
shows *Fin.terminal a* ⟷ *a* = *Fin.MkIde 1*
⟨*proof*⟩

lemma *MkIde-1-eq*:
shows *Fin.MkIde 1* = **1**$^?$
⟨*proof*⟩

lemma *finite-Set*:
assumes *Fin.ide a*
shows *finite* (*Set a*)
⟨*proof*⟩

lemma *card-Set*:
assumes *Fin.ide a*
shows *card* (*Set a*) = *Fin.Dom a*

⟨*proof*⟩

abbreviation *mkpoint*
where *mkpoint n k* ≡ *Fin.MkArr 1 n* (λ*x*. *if x* = *1 then k* :: *nat else undefined*)

abbreviation *valof*
where *valof x* ≡ *Fin.Map x* (*1* :: *nat*)

lemma *mkpoint-in-hom* [*intro*, *simp*]:
assumes *k* ∈ {*1*..*n*}
shows *Fin.in-hom* (*mkpoint n k*) (*Fin.MkIde 1*) (*Fin.MkIde n*)
⟨*proof*⟩

lemma *valof-in-range*:
assumes *Fin.in-hom x* $\mathbf{1}^?$ *a*
shows *valof x* ∈ {*1*..*Fin.Dom a*}
⟨*proof*⟩

lemma *valof-mkpoint*:
shows *valof* (*mkpoint n k*) = *k*
⟨*proof*⟩

lemma *mkpoint-valof*:
assumes *Fin.in-hom x* $\mathbf{1}^?$ *a*
shows *mkpoint* (*Fin.Dom a*) (*valof x*) = *x*
⟨*proof*⟩

lemma *Map-arr-eq*:
assumes *Fin.in-hom f a b*
shows *Fin.Map f* = (λ*k*. *if k* ∈ {*1*..*Fin.Dom a*}
then Fin.Map (*Fun f* (*mkpoint* (*Fin.Dom a*) *k*)) *1*
else undefined)
(**is** *Fin.Map f* = *?F*)
⟨*proof*⟩

sublocale *sets-cat* ‹*finite* :: *nat set* ⇒ *bool*› *comp*
⟨*proof*⟩

lemma *is-sets-cat*:
shows *sets-cat* (*finite* :: *nat set* ⇒ *bool*) *comp*
⟨*proof*⟩

sublocale *small-product* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

sublocale *sets-cat-with-pairing* ‹*finite* :: *nat set* ⇒ *bool*› *comp*
⟨*proof*⟩

lemma *is-sets-cat-with-pairing*:

shows *sets-cat-with-pairing* (*finite* :: *nat set* ⇒ *bool*) *comp*
⟨*proof*⟩

sublocale *lifting* ‹*Collect Fin.arr*›
⟨*proof*⟩

sublocale *sets-cat-with-powering* ‹*finite* :: *nat set* ⇒ *bool*› *comp*
⟨*proof*⟩

lemma *is-sets-cat-with-powering*:
shows *sets-cat-with-powering* (*finite* :: *nat set* ⇒ *bool*) *comp*
⟨*proof*⟩

sublocale *small-sum* ‹*finite* :: *nat set* ⇒ *bool*›
⟨*proof*⟩

sublocale *sets-cat-with-tupling* ‹*finite* :: *nat set* ⇒ *bool*› *comp*
⟨*proof*⟩

theorem *is-sets-cat-with-tupling*:
shows *sets-cat-with-tupling* (*finite* :: *nat set* ⇒ *bool*) *comp*
⟨*proof*⟩

end

Here is the final top-level interpretation. Note that this is proved in "vanilla HOL" without any additional axioms.

interpretation $SetsCat_{fin}$: *finite-sets-cat* ⟨*proof*⟩

6.2 Category of ZFC Sets

In this section we construct an interpretation of *sets-cat-with-tupling-and-infinity*, which includes infinite sets. As this cannot be done in "vanilla HOL", for this construction we use *ZFC-in-HOL*, which extends HOL with axioms for a type V that models the set-theoretic universe provided by ZFC. Actually, we have previously given, in theory *Category3.ZFC-SetCat*, a construction of a category of small sets and functions based on *ZFC-in-HOL*. Since that work was already done, all we need to do here is to show that the previously constructed category interprets the *sets-cat-with-tupling-and-infinity* locale.

locale *ZFC-sets-cat*
begin

Here we import the previous construction from *Category3.ZFC-SetCat*.

interpretation *ZFC*: *ZFC-set-cat* ⟨*proof*⟩

We use the notion of "smallness" provided by *ZFC-in-HOL*.

sublocale *smallness* ‹*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*›

⟨*proof*⟩

sublocale *sets-cat-base* ‹*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*› *ZFC.comp*
⟨*proof*⟩

sublocale *sets-cat* ‹*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*› *ZFC.comp*
⟨*proof*⟩

lemma *is-sets-cat*:
shows *sets-cat* (*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*) *ZFC.comp*
⟨*proof*⟩

Arrows of the category can be encoded as elements of V.

abbreviation *arr-to-V*
where *arr-to-V f* ≡ *vpair*
(*vpair* (*ZFC.V-of-ide* (*ZFC.dom f*)) (*ZFC.V-of-ide* (*ZFC.cod f*)))
(*ZFC.V-of-arr f*)

lemma *inj-arr-to-V*:
shows *inj-on arr-to-V* (*Collect ZFC.arr*)
⟨*proof*⟩

As it happens, V also embeds into the collection of arrows, so the two are equipollent. Thus, the fact that V is a universe can be transferred to the collection of arrows. So we can save ourselves some work here.

lemma *eqpoll-Collect-arr-V*:
shows *Collect ZFC.arr* ∪ {*ZFC.null*} ≈ (*UNIV* :: *V set*)
and *Collect ZFC.arr* ≈ (*UNIV* :: *V set*)
⟨*proof*⟩

sublocale *universe* ‹*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*› ‹*Collect ZFC.arr*› *ZFC.null*
⟨*proof*⟩

sublocale *sets-cat-with-tupling-and-infinity*
‹*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*› *ZFC.comp*
⟨*proof*⟩

theorem *is-sets-cat-with-tupling-and-infinity*:
shows *sets-cat-with-tupling-and-infinity*
(*ZFC-in-HOL.small* :: *ZFC-in-HOL.V set* ⇒ *bool*) *ZFC.comp*
⟨*proof*⟩

end

Here is the final top-level interpretation.

interpretation $SetsCat_{ZFC}$: *ZFC-sets-cat* ⟨*proof*⟩

end

Bibliography

[1] F. W. Lavere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences of the U.S.A.*, 52:1506–1511, 1964.

[2] L. C. Paulson. Zermelo fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, October 2019. https://isa-afp.org/entries/ZFC_in_HOL.html, Formal proof development.

[3] E. W. Stark. Category theory with adjunctions and limits. *Archive of Formal Proofs*, June 2016. http://isa-afp.org/entries/Category3.shtml, Formal proof development.