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

Eugene W. Stark

February 3, 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 feature, 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 -> 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* ⇒ *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* ⇒ *bool*
assumes *lepoll-small-ax*: ⟦*sml X*; *lepoll Y X*⟧ ⟹ *sml Y*
begin

definition *small* :: *'a set* ⇒ *bool*
where *small X* ≡ ∃ X_0. *sml* X_0 ∧ *X* ≈ X_0

lemma *smallI*:
assumes *sml* X_0 **and** *X* ≈ X_0
shows *small X*
using *assms small-def* **by** *auto*

lemma *smallE*:
assumes *small X*
and ⋀X_0. ⟦*sml* X_0; *X* ≈ X_0⟧ ⟹ *T*
shows *T*
using *assms small-def* **by** *blast*

lemma *small-iff-sml*:
shows *small X* ⟷ *sml X*
using *eqpoll-imp-lepoll small-def lepoll-small-ax* **by** *blast*

lemma *lepoll-small*:
assumes *small X* **and** *lepoll Y X*
shows *small Y*
by (*metis assms*(*1*,*2*) *eqpoll-sym image-lepoll inj-on-image-eqpoll-self*
lepoll-def' lepoll-small-ax lepoll-trans lepoll-trans2 small-def)

lemma *smaller-than-small*:
assumes *small X* **and** *Y* ⊆ *X*
shows *small Y*
using *assms lepoll-small subset-imp-lepoll* **by** *blast*

lemma *small-image* [*intro*, *simp*]:
assumes *small X*
shows *small* (*f* ‘ *X*)
using *assms small-def image-lepoll lepoll-small* **by** *blast*

```
  lemma small-image-iff [simp]: inj-on f A ⟹ small (f ‘ A) ⟷ small A
    by (metis small-image the-inv-into-onto)

  lemma small-Collect [simp]: small X ⟹ small {x ∈ X. P x}
    by (simp add: smaller-than-small subset-imp-lepoll)

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 ⟹ small X
    using small-finite-ax
    by (meson eqpoll-def eqpoll-sym eqpoll-trans ex-bij-betw-nat-finite-1 small-def)

  lemma small-insert:
  assumes small X
  shows small (insert a X)
    by (meson assms eqpoll-imp-lepoll finite.insertI infinite-insert-eqpoll
      small-finite lepoll-small)

  lemma small-insert-iff [iff]: small (insert a X) ⟷ small X
    by (meson small-insert smaller-than-small subset-imp-lepoll subset-insertI)

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⟧ ⟹ ∃ Z. sml Z ∧ eqpoll (X × Y) Z
begin

  lemma small-product [simp]:
  assumes small X small Y shows small (X × Y)
    by (metis assms(1,2) eqpoll-trans small-def small-product-ax times-eqpoll-cong)
```

```
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; ⋀x. x ∈ X ⟹ sml (F x)⟧
                        ⟹ ∃ U. sml U ∧ eqpoll (Sigma X F) U
begin

  lemma small-binary-sum:
  assumes small X and small Y
  shows small (({False} × X) ∪ ({True} × Y))
  proof −
    obtain X_0 ϱ where X_0: sml X_0 ∧ bij-betw ϱ X X_0
      using assms(1) small-def eqpoll-def by blast
    obtain Y_0 σ where Y_0: sml Y_0 ∧ bij-betw σ Y Y_0
      using assms(2) small-def eqpoll-def by blast
    obtain B_0 β where B_0: sml B_0 ∧
                         bij-betw β {None, Some ({} :: 'b set)} B_0
      by (metis eqpoll-def finite.emptyI smallE small-finite.small-finite
        small-finite.small-insert-iff small-finite-axioms)
    let ?False = β None and ?True = β (Some {})
    have ne: ?False ≠ ?True
      by (metis B_0 bij-betw-inv-into-left insertCI option.discI)
    let ?ι = λz. if fst z = False then (?False, ϱ (snd z)) else (?True, σ (snd z))
    have small (({?False} × X_0) ∪ ({?True} × Y_0))
    proof −
      have Sigma B_0 (λx. if x = ?False then X_0 else Y_0) =
          ({?False} × X_0) ∪ ({?True} × Y_0)
      proof
        show Sigma B_0 (λx. if x = ?False then X_0 else Y_0) ⊆
              ({?False} × X_0) ∪ ({?True} × Y_0)
        proof
          fix bx
          assume bx: bx ∈ Sigma B_0 (λx. if x = ?False then X_0 else Y_0)
          have fst bx = ?False ∨ fst bx = ?True
            using B_0 bij-betw-imp-surj-on bx by fastforce
          moreover have fst bx = ?False ⟹ snd bx ∈ X_0
            using bx by force
          moreover have fst bx ≠ ?False ⟹ snd bx ∈ Y_0
            using bx by force
          ultimately show bx ∈ ({?False} × X_0) ∪ ({?True} × Y_0)
            by (metis Un-iff insertCI mem-Times-iff)
        qed
        show ({?False} × X_0) ∪ ({?True} × Y_0) ⊆
```

```
            Sigma B_0 (λx. if x = ?False then X_0 else Y_0)
        using B_0 bij-betw-apply ne by fastforce
    qed
    moreover have small (Sigma B_0 (λx. if x = ?False then X_0 else Y_0))
      using X_0 Y_0 B_0 small-sum-ax small-def by force
    ultimately show ?thesis by auto
  qed
  moreover have bij-betw ?ι
                  (({False} × X) ∪ ({True} × Y))
                  (({?False} × X_0) ∪ ({?True} × Y_0))
  proof (intro bij-betwI)
    let ?ι' = λz. if fst z = ?False then (False, inv-into X ϱ (snd z))
                  else (True, inv-into Y σ (snd z))
    show ?ι ∈ ({False} × X) ∪ ({True} × Y) → ({?False} × X_0) ∪ ({?True} × Y_0)
      using X_0 Y_0 bij-betw-def
      by (auto simp add: bij-betw-apply)
    show ?ι' ∈ ({?False} × X_0) ∪ ({?True} × Y_0) → ({False} × X) ∪ ({True} × Y)
    proof
      fix z
      assume z: z ∈ ({?False} × X_0) ∪ ({?True} × Y_0)
      show ?ι' z ∈ ({False} × X) ∪ ({True} × Y)
        using z
        by (metis Un-iff X_0 Y_0 bij-betw-def inv-into-into mem-Sigma-iff ne prod.collapse
            singleton-iff)
    qed
    show ⋀x. x ∈ {False} × X ∪ {True} × Y ⟹ ?ι' (?ι x) = x
    proof −
      fix x
      assume x: x ∈ {False} × X ∪ {True} × Y
      have ?ι x ∈ ({?False} × X_0) ∪ ({?True} × Y_0)
        using X_0 Y_0 bij-betwE fst-conv mem-Times-iff x by fastforce
      thus ?ι' (?ι x) = x
        using x X_0 Y_0 bij-betw-inv-into-left ne
        by auto[1] fastforce+
    qed
    show ⋀y. y ∈ ({?False} × X_0) ∪ ({?True} × Y_0) ⟹ ?ι (?ι' y) = y
      using X_0 Y_0 bij-betw-inv-into-right ne by fastforce
  qed
  ultimately show ?thesis
    by (meson eqpoll-def eqpoll-trans small-def)
qed

lemma small-union:
assumes X: small X and Y: small Y
shows small (X ∪ Y)
proof −
  have lepoll (X ∪ Y) (({False} × X) ∪ ({True} × Y))
  proof −
    let ?ι = λz. if z ∈ X then (False, z) else (True, z)
    have ?ι ∈ X ∪ Y → ({False} × X) ∪ ({True} × Y) ∧ inj-on ?ι (X ∪ Y)
      by (simp add: inj-on-def)
    thus ?thesis
      using lepoll-def' by blast
  qed
  moreover have small (({False} × X) ∪ ({True} × Y))
    using assms small-binary-sum by blast
  ultimately show ?thesis
    using lepoll-small by blast
qed

lemma small-Union-spc:
assumes A0: sml A0 and B: ⋀x. x ∈ A0 ⟹ small (B x)
shows small (⋃x∈A0. B x)
proof −
  have 1: ∃B0. ∀x. x ∈ A0 ⟶ sml (B0 x) ∧ eqpoll (B x) (B0 x)
    using A0 B small-def by meson
  obtain B0 where B0: ⋀x. x ∈ A0 ⟹ sml (B0 x) ∧ eqpoll (B0 x) (B x)
    using assms 1 eqpoll-sym by blast
  have 2: ∃σ. ∀x. x ∈ A0 ⟶ bij-betw (σ x) (B0 x) (B x)
    using B0 eqpoll-def
    by (meson ‹⋀x. x ∈ A0 ⟹ sml (B0 x) ∧ B0 x ≈ B x› eqpoll-def)
  obtain σ where σ: ⋀x. x ∈ A0 ⟹ bij-betw (σ x) (B0 x) (B x)
    using 2 by blast
  have small (Sigma A0 B0)
    using assms small-sum-ax [of A0 B0] B0 small-def by blast
  moreover have lepoll (⋃x∈A0. B x) (Sigma A0 B0)
  proof −
    have (λz. σ (fst z) (snd z)) ‘ Sigma A0 B0 = (⋃x∈A0. B x)
    proof
      show (λz. σ (fst z) (snd z)) ‘ Sigma A0 B0 ⊆ ⋃ (B ‘ A0)
        unfolding Sigma-def
        using σ bij-betwE by fastforce
      show ⋃ (B ‘ A0) ⊆ (λz. σ (fst z) (snd z)) ‘ Sigma A0 B0
      proof
        fix z
        assume z: z ∈ (⋃ (B ‘ A0))
        obtain x where x: x ∈ A0 ∧ z ∈ B x
          using z by blast
        have (x, inv-into (B0 x) (σ x) z) ∈ Sigma A0 B0
          by (metis SigmaI σ bij-betw-def inv-into-into x)
        moreover have (λz. σ (fst z) (snd z)) (x, inv-into (B0 x) (σ x) z) = z
          using σ bij-betw-inv-into-right x by fastforce
        ultimately show z ∈ (λz. σ (fst z) (snd z)) ‘ Sigma A0 B0
          by force
      qed
    qed
    thus ?thesis
      by (metis image-lepoll)
  qed
  ultimately show ?thesis
    using lepoll-small by blast
qed

lemma small-Union [simp, intro]:
assumes A: small A and B: ⋀x. x ∈ A ⟹ small (B x)
shows small (⋃x∈A. B x)
proof −
  obtain A0 ϱ where A0: sml A0 ∧ bij-betw ϱ A0 A
    using assms(1) small-def eqpoll-def eqpoll-sym by blast
  have eqpoll (⋃x∈A. B x) (⋃x∈A0. (B ∘ ϱ) x)
    by (metis A0 bij-betw-def eqpoll-refl image-comp)
  moreover have small (⋃x∈A0. (B ∘ ϱ) x)
    by (metis A0 B bij-betwE comp-apply small-Union-spc)
  ultimately show ?thesis
    using eqpoll-imp-lepoll lepoll-small by blast
qed
```

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
    show ⋀X Y. ⟦sml X; sml Y⟧ ⟹ ∃Z. sml Z ∧ X × Y ≈ Z
      by (simp add: small-sum-ax)
  qed

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 ⟹ ∃PX. sml PX ∧ eqpoll (Pow X) PX
begin

  lemma small-powerset:
  assumes small X
  shows small (Pow X)
    using assms small-powerset-ax
    by (meson bij-betw-Pow eqpoll-def eqpoll-trans small-def)

  lemma large-UNIV:
  shows ¬ small (UNIV :: 'a set)
    using small-powerset-ax Cantors-theorem
    by (metis Pow-UNIV UNIV-I eqpoll-iff-bijections small-iff-sml surjI)
```

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*: $\exists X.\ sml\ X \wedge eqpoll\ X\ (UNIV :: nat\ set)$
begin

  lemma *small-nat*:
  shows *small* (*UNIV* :: *nat set*)
    using *small-nat-ax small-def eqpoll-sym* **by** *auto*

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 ⇒ 'c) ⇒ 'c ⇒ bool
  where popular-value F y ≡ ¬ small {x. F x = y}

  definition some-popular-value :: ('b ⇒ 'c) ⇒ 'c
  where some-popular-value F ≡ SOME y. popular-value F y

  lemma popular-value-some-popular-value:
  assumes ∃ y. popular-value F y
  shows popular-value F (some-popular-value F)
    using assms someI-ex [of λy. popular-value F y] some-popular-value-def by metis

  abbreviation at-most-one-popular-value
  where at-most-one-popular-value F ≡ ∃≤1 y. popular-value F y

  definition small-function
  where small-function F ≡ small (range F) ∧ 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
    using assms small-function-def by blast

  lemma small-functionD [dest]:
  assumes small-function f
  shows small (range f) and at-most-one-popular-value f
    using assms small-function-def by auto

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 ∈ range F
    using assms not-finite-existsD small-finite by auto

  lemma small-function-const:
  shows small-function (λx. y)
    by (auto simp add: Uniq-def small-finite)

  definition inv-into_E
  where inv-into_E X f ≡ λy. 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
   show small (range (inv-into_E X f))
   proof −
     have small X
       by (meson assms(1,2) small-functionD(1) small-image-iff smaller-than-small
           subset-UNIV subset-image-iff)
     moreover have range (inv-into_E X f) ⊆ X ∪ {SOME x. popular-value f (f x)}
       unfolding inv-into_E-def
       using assms(2) inf-sup-aci(5) by auto
     ultimately show ?thesis
       using smaller-than-small by auto
   qed
   show at-most-one-popular-value (inv-into_E X f)
   proof −
     have ⋀x. popular-value (inv-into_E X f) x ⟹ x = (SOME x. popular-value f (f x))
     proof −
       fix x
       assume x: popular-value (inv-into_E X f) x
       have f x ∈ {y. y ∈ f ‘ X ∧ x = inv-into X f y} ∨ x = (SOME x. popular-value f (f x))
         using assms x
         unfolding inv-into_E-def
         using not-finite-existsD small-finite  by fastforce
       moreover have x ≠ (SOME x. popular-value f (f x)) ⟹
                     f x ∉ {y. y ∈ f ‘ X ∧ x = inv-into X f y}
       proof −
         assume 1: x ≠ (SOME x. popular-value f (f x))
         have small {y. y ∈ f ‘ X ∧ x = inv-into X f y}
           using assms
           by (metis (no-types, lifting) image-subset-iff mem-Collect-eq rangeI
               small-functionD(1) smaller-than-small subsetI)
         thus ?thesis
           using x 1
           unfolding inv-into_E-def
           by (simp add: Collect-mono smallness.smaller-than-small smallness-axioms)
       qed
       ultimately show x = (SOME x. popular-value f (f x)) by blast
     qed
     thus ?thesis
       using Uniq-def by blast
   qed
 qed

end
```

```
context small-sum
begin

  lemma small-function-comp:
  assumes small-function f and small-function g
  shows small-function (g ∘ f)
  proof
    show small (range (g ∘ f))
      by (metis assms(1) fun.set-map small-image small-functionD(1))
    show at-most-one-popular-value (g ∘ f)
    proof −
      have ∗: ⋀z. popular-value (g ∘ f) z ⟹ ∃ y. popular-value f y ∧ g y = z
      proof −
        fix z
        assume z: popular-value (g ∘ f) z
        have ¬ small {x. g (f x) = z}
          using z by auto
        moreover have {x. g (f x) = z} = (⋃ y∈range f ∩ {y. g y = z}. {x. f x = y})
          by auto
        moreover have small (range f ∩ {y. g y = z})
          using assms(1) small-functionD(1) smaller-than-small by force
        ultimately have ∃ y. y ∈ range f ∩ {y. g y = z} ∧ popular-value f y
          by auto
        thus ∃ y. popular-value f y ∧ g y = z by blast
      qed
      show ?thesis
      proof
        fix y y′
        assume y: popular-value (g ∘ f) y and y′: popular-value (g ∘ f) y′
        have ∃ x. popular-value f x ∧ g x = y
          using y ∗ by blast
        moreover have ∃ x. popular-value f x ∧ g x = y′
          using y′ ∗ by blast
        ultimately show y = y′
          using assms(2)
          by (metis (mono-tags, lifting) assms(1) small-functionD(2) the1-equality′)
      qed
    qed
  qed
```

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
    show ∃ v. popular-value F v ⟹ ¬ small (UNIV :: ′b set)
      using smaller-than-small by blast
    have ¬ (∃ v. popular-value F v) ⟹ small (UNIV :: 'b set)
    proof −
      assume ¬ (∃ y. popular-value F y)
      hence ⋀y. small {x. F x = y}
        by blast
      moreover have UNIV = (⋃ y∈range F. {x. F x = y})
        by auto
      ultimately show small (UNIV :: 'b set)
        using assms(1) small-function-def by (metis small-Union)
    qed
    thus ¬ small (UNIV :: 'b set) ⟹ ∃ v. popular-value F v
      by blast
  qed
```

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 (cases ∃ y. popular-value F y)
    assume 1: ¬ (∃ y. popular-value F y)
    thus ?thesis
      using assms ex-popular-value-iff smaller-than-small by blast
    next
    assume 1: ∃ y. popular-value F y
    have popular-value F (some-popular-value F)
      using 1 popular-value-some-popular-value by metis
    hence 2: ⋀y. y ≠ some-popular-value F ⟹ small {x. F x = y}
      using assms
      unfolding small-function-def
      by (meson Uniq-D)
    moreover have {x. F x ≠ some-popular-value F} =
                  (⋃ y∈{y. y ∈ range F ∧ y ≠ some-popular-value F}. {x. F x = y})
      by auto
    ultimately show ?thesis
      using assms
      unfolding small-function-def
      by auto
  qed
```

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 ≡ (λx. (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 –
  let ?F = λy. {x. f x = y}
  have SF-Dom f = (⋃y ∈ SF-Rng f. ?F y)
  proof
    show SF-Dom f ⊆ (⋃y ∈ SF-Rng f. ?F y)
       by blast
   show (⋃y ∈ SF-Rng f. ?F y) ⊆ SF-Dom f
    proof
      fix x
      assume x: x ∈ (⋃y ∈ SF-Rng f. ?F y)
      obtain S y where S: x ∈ S ∧ y ∈ SF-Rng f ∧ S = {x. f x = y}
        using x by force
      show x ∈ SF-Dom f
        using S by fastforce
    qed
  qed
  moreover have ⋀y. y ∈ SF-Rng f ⟹ small (?F y)
    using assms by blast
  ultimately show ?thesis
    using small-Union [of SF-Rng f ?F]
    by (metis assms image-mono small-functionD(1) smaller-than-small subset-UNIV)
qed

lemma small-SF-Rng:
assumes small-function f
shows small (SF-Rng f)
  using assms small-SF-Dom by blast

lemma small-SF-Grph:
assumes small-function f
shows small (SF-Grph f)
  using assms small-SF-Dom by blast

lemma small-function-expansion:
assumes small-function f
shows f = (λx. if x ∈ fst ‘ SF-Grph f then (THE y. (x, y) ∈ SF-Grph f) else the-PV f)
proof
  fix x
  show f x = (if x ∈ fst ‘ SF-Grph f then (THE y. (x, y) ∈ SF-Grph f) else the-PV f)
```

```
      proof (cases x ∈ SF-Dom f)
        show x ∉ SF-Dom f ⟹ ?thesis
        proof –
          assume x ∉ SF-Dom f
          hence f x = the-PV f
            using assms the1-equality' by fastforce
          thus ?thesis
            by (simp add: image-iff)
        qed
        show x ∈ SF-Dom f ⟹ ?thesis
          by (simp add: image-iff)
      qed
    qed

  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 –
      let ?Rep = λf. (SF-Grph f, Collect (popular-value f))
      let ?SF = {f. small-function f ∧ SF-Dom f ⊆ X ∧ range f ⊆ Y}
      have *: ⋀f x. ⟦f ∈ ?SF; x ∉ SF-Dom f⟧ ⟹ {f x} = Collect (popular-value f)
      proof –
        fix f x
        assume f: f ∈ ?SF and x: x ∉ SF-Dom f
        show {f x} = Collect (popular-value f)
        proof –
          have 1: popular-value f (f x)
            using x by blast
          have ∃!y. popular-value f y
          proof –
            have ∃y. popular-value f y
              using 1 by blast
            moreover have ⋀y y'. ⟦popular-value f y; popular-value f y'⟧ ⟹ y = y'
              using f Uniq-def small-functionD(2)
              by (metis (mono-tags, lifting) mem-Collect-eq)
            ultimately show ?thesis by blast
          qed
          thus ?thesis
            using f 1 by blast
        qed
      qed
      have small (?Rep ‘ ?SF)
      proof −
        have ?Rep ∈ ?SF → Pow (X × Y) × Pow Y
          using popular-value-in-range by fastforce
        moreover have small (Pow (X × Y) × Pow Y)
          using assms by (simp add: small-powerset)
        ultimately show ?thesis
          by (simp add: image-subset-iff-funcset smaller-than-small)
      qed
      moreover have inj-on ?Rep ?SF
      proof
        fix f g :: 'b ⇒ 'c
        assume f: f ∈ ?SF and g: g ∈ ?SF
        assume eq: ?Rep f = ?Rep g
        show f = g
        proof
          fix x
          show f x = g x
          proof (cases x ∈ SF-Dom f)
            show x ∉ SF-Dom f ⟹ ?thesis
            proof −
              assume x: x ∉ SF-Dom f
              have {f x} = Collect (popular-value f)
                using f x ∗ by blast
              also have ... = Collect (popular-value g)
                using eq by force
              also have ... = {g x}
                using g x eq ∗ [of g x] by blast
              finally show f x = g x by blast
            qed
            show x ∈ SF-Dom f ⟹ ?thesis
              using f g eq small-function-expansion by blast
          qed
        qed
      qed
      ultimately show ?thesis
        using small-image-iff by blast
    qed

  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
    fix x y :: 'b list
    have 1: ⋀l :: 'b list. list-as-fn l (length l) = None
      unfolding list-as-fn-def by simp
    assume eq: list-as-fn x = list-as-fn y
    have length x = length y
      using eq 1
      by (metis (no-types, lifting) list-as-fn-def nle-le not-Some-eq)
    moreover have ⋀n. n < length x ⟹ x ! n = y ! n
      using eq list-as-fn-def
      by (metis calculation leD option.inject)
    ultimately show x = y
      using nth-equalityI by blast
  qed

  lemma small-function-list-as-fn:
  shows small-function (list-as-fn l)
    using Uniq-def small-function-def small-nat smaller-than-small by fastforce

  lemma small-listset:
  assumes small Y
  shows small {l. List.set l ⊆ Y}
  proof –
    let ?SF = λf. small-function f ∧ SF-Dom f ⊆ (UNIV :: nat set) ∧
                    range f ⊆ Some ‘ Y ∪ {None}
    have list-as-fn ‘ {l. List.set l ⊆ Y} ⊆ Collect ?SF
    proof
      fix f
      assume f: f ∈ list-as-fn ‘ {l. List.set l ⊆ Y}
      show f ∈ Collect ?SF
        using f small-function-list-as-fn
        unfolding list-as-fn-def
        apply auto
        by fastforce
    qed
```

```
      moreover have small (Collect ?SF)
        using assms small-nat small-funcset [of UNIV :: nat set Some ‘ Y ∪ {None}]
        by auto
      ultimately show ?thesis
        using small-image-iff [of list-as-fn {l. list.set l ⊆ Y}] inj-list-as-fn
          smaller-than-small
        by (metis (mono-tags, lifting) injD inj-onI)
    qed

  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* × *hom x b* ⊆ *univ* × *univ* and then use an embedding of *univ* × *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* × *b* with *hom x* (*a* × *b*) ≅ *hom x a* × *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 ι X* ≡ *inj-on ι X* ∧ *ι* ' *X* ⊆ *univ*

definition *some-embedding-of*
where *some-embedding-of X* ≡ *SOME ι. is-embedding-of ι X*

abbreviation *embeds*
where *embeds X* ≡ ∃ *ι. is-embedding-of ι X*

```
  lemma is-embedding-of-some-embedding-of:
  assumes embeds X
  shows is-embedding-of (some-embedding-of X) X
    unfolding some-embedding-of-def
    using assms someI-ex [of λι. is-embedding-of ι X] by force

  lemma embeds-subset:
  assumes embeds X and Y ⊆ X
  shows embeds Y
    using assms
    by (meson dual-order.trans image-mono inj-on-subset)

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)
    unfolding some-lifting-def
    using is-embedding-of-some-embedding-of embeds-lift by blast

  lemma some-lifting-in-univ [intro, simp]:
  shows some-lifting None ∈ univ
  and x ∈ univ ⟹ some-lifting (Some x) ∈ univ
    using some-lifting-is-embedding by auto

  lemma some-lifting-cancel:
  shows ⟦x ∈ univ; some-lifting (Some x) = some-lifting None⟧ ⟹ False
  and ⟦x ∈ univ; x' ∈ univ; some-lifting (Some x) = some-lifting (Some x')⟧ ⟹ x = x'
    using some-lifting-is-embedding
     apply (meson Un-iff imageI inj-on-contraD insertI1 option.simps(3))
    using some-lifting-is-embedding
    by (meson UnI2 imageI inj-on-contraD option.inject)

  lemma infinite-univ:
  shows infinite univ
    by (metis None-notin-image-Some card-image card-inj-on-le card-insert-disjoint
      embeds-lift finite-imageI inj-Some insert-is-Un le-imp-less-Suc linorder-neq-iff)

  lemma embeds-bool:
  shows embeds (UNIV :: bool set)
    by (metis comp-inj-on ex-inj image-comp image-mono infinite-univ
        infinite-iff-countable-subset inj-on-subset subset-trans top-greatest)

  lemma embeds-nat:
  shows embeds (UNIV :: nat set)
    by (metis infinite-univ infinite-iff-countable-subset)

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)
    unfolding some-pairing-def
    using embeds-pairs is-embedding-of-some-embedding-of by blast

  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
    using assms first-def some-pairing-is-embedding
    by (metis (mono-tags, lifting) fst-eqD inv-into-f-f mem-Times-iff snd-eqD)
  lemma second-conv:
  assumes x ∈ univ and y ∈ univ
  shows second (pair x y) = y
    using assms second-def some-pairing-is-embedding
    by (metis (mono-tags, lifting) fst-eqD inv-into-f-f mem-Times-iff snd-eqD)

  lemma pair-conv:
  assumes is-pair x
  shows pair (first x) (second x) = x
    using assms first-def second-def embeds-pairs is-embedding-of-some-embedding-of
    by (simp add: f-inv-into-f)

  lemma some-pairing-in-univ [intro, simp]:
  shows ⟦x ∈ univ; y ∈ univ⟧ ⟹ pair x y ∈ univ
    using some-pairing-is-embedding by blast

  lemma some-pairing-cancel:
  shows ⟦x ∈ univ; x' ∈ univ; y ∈ univ; y' ∈ univ; pair x y = pair x' y'⟧
          ⟹ x = x' ∧ y = y'
    using embeds-pairs
    by (metis first-conv second-conv)

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 ⇒ bool
and univ :: 'U set +
assumes embeds-small-sets: embeds {X. X ⊆ univ ∧ small X}
begin

  abbreviation some-embedding-of-small-sets :: ('U set) ⇒ 'U
  where some-embedding-of-small-sets ≡ some-embedding-of {X. X ⊆ univ ∧ small X}

  definition emb-set :: ('U set) ⇒ 'U
  where emb-set ≡ some-embedding-of-small-sets

  lemma emb-set-is-embedding:
  shows is-embedding-of emb-set {X. X ⊆ univ ∧ small X}
    unfolding emb-set-def
    using embeds-small-sets is-embedding-of-some-embedding-of by blast
```

```
lemma emb-set-in-univ [intro, simp]:
shows ⟦X ⊆ univ; small X⟧ ⟹ emb-set X ∈ univ
  using emb-set-is-embedding by blast

lemma emb-set-cancel:
shows ⟦X ⊆ univ; small X; X′ ⊆ univ; small X′; emb-set X = emb-set X′⟧ ⟹ X = X′
  using emb-set-is-embedding
  by (metis (mono-tags, lifting) inj-onD mem-Collect-eq)
```

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

```
lemma large-univ:
shows ¬ small univ
proof −
  have small univ ⟹ False
  proof −
    assume small: small univ
    have embeds (Pow univ)
      using small smaller-than-small embeds-small-sets
      by (metis (no-types, lifting) CollectI PowD embeds-subset subsetI)
    thus False
      using Cantors-theorem
      by (metis Pow-not-empty inj-on-iff-surj)
  qed
  thus ?thesis by blast
qed

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 ⇒ bool
and univ :: 'U set
and null :: 'U
begin
```

EF is the set of extensional functions on *univ*. These map *univ* to *univ* ∪ {*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 ≡ {f. f ‘ univ ⊆ univ ∪ {null} ∧ (∀ x. x ∉ univ ⟶ f x = null)}

abbreviation SEF
where SEF ≡ Collect small-function ∩ EF

lemma EF-apply:
assumes F ∈ EF
shows x ∈ univ ⟹ F x ∈ univ ∪ {null}
and x ∉ univ ⟹ F x = null
  using assms
  unfolding EF-def by auto
```

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 ∈ SEF
shows ∃ v. popular-value F v
  using assms ex-popular-value-iff large-UNIV
  by (metis Int-iff large-univ mem-Collect-eq smaller-than-small top-greatest)
```

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* ∪ {*null*} can be canonically described by a small subset of *univ* × (*univ* ∪ {*null*}) consisting of all pairs (*x*, *F x*) ⊆ *univ* × (*univ* ∪ {*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 (intro exI conjI)
  have range-F: ⋀F. F ∈ SEF ⟹ range F ⊆ univ ∪ {null}
    unfolding EF-def by blast
  let ?lift = some-embedding-of (univ ∪ {null})
  have lift: is-embedding-of ?lift (univ ∪ {null})
    using embeds-lift is-embedding-of-some-embedding-of
    by (metis bij-betw-imp-surj-on infinite-univ infinite-imp-bij-betw2
        inj-on-iff-surj insert-not-empty sup-bot.neutr-eq-iff)
  have lift-cancel [simp]: ⋀x y. ⟦x ∈ univ ∪ {null}; y ∈ univ ∪ {null}; ?lift x = ?lift y⟧
                              ⟹ x = y
    using lift by (meson UnI1 inj-on-eq-iff)
  have 0: ⋀F. F ∈ SEF ⟹ ?lift (some-popular-value F) ∈ univ
    using range-F popular-value-in-range popular-value-some-popular-value
      SEFs-have-popular-value
    by (metis image-subset-iff lift subset-eq)
  have 1: ⋀F. F ∈ SEF ⟹ small {x ∈ univ. ¬ popular-value F (F x)}
    by (metis (no-types) CollectD Collect-conj-eq IntE inf-le2 small-SF-Dom
        smaller-than-small)
  have 2: ⋀F. F ∈ SEF ⟹
               (λa. pair a (?lift (F a))) ‘ {x ∈ univ. ¬ popular-value F (F x)} ⊆ univ
    apply auto[1]
    by (metis (no-types, lifting) CollectD EF-def Un-commute image-subset-iff insert-is-Un
        lift some-pairing-in-univ)
  have 3: ⋀F. F ∈ SEF ⟹
               emb-set ((λa. pair a (?lift (F a))) ‘ {x ∈ univ. ¬ popular-value F (F x)})
                ∈ univ
    using 1 2 by blast

  let ?e = λF. pair (?lift (some-popular-value F))
                    (emb-set ((λa. pair a (?lift (F a))) ‘
                              {x ∈ univ. ¬ popular-value F (F x)}))
  show ?e ‘ SEF ⊆ univ
    using 0 3 some-pairing-in-univ by blast
  show inj-on ?e SEF
  proof (intro inj-onI)
    fix F F' :: 'U ⇒ 'U
    assume F: F ∈ SEF
    assume F': F' ∈ SEF
    assume eq: ?e F = ?e F'
    have ∗: ⋀x. x ∈ univ ⟹
                 first (pair x (?lift (F x))) = x ∧
                 second (pair x (?lift (F x))) = ?lift (F x) ∧
                 first (pair x (?lift (F' x))) = x ∧
                 second (pair x (?lift (F' x))) = ?lift (F' x)
      by (meson F F' first-conv image-subset-iff lift range-F range-subsetD second-conv)
    have 4: ?lift (some-popular-value F) = ?lift (some-popular-value F') ∧
             emb-set ((λa. pair a (?lift (F a))) ‘ {x ∈ univ. ¬ popular-value F (F x)}) =
             emb-set ((λa. pair a (?lift (F' a))) ‘ {x ∈ univ. ¬ popular-value F' (F' x)})
      using F F' 0 3 eq some-pairing-cancel by meson
    have 5: (λa. pair a (?lift (F a))) ‘ {x ∈ univ. ¬ popular-value F (F x)} =
             (λa. pair a (?lift (F' a))) ‘ {x ∈ univ. ¬ popular-value F' (F' x)}
      using F F' 1 2 4 small-preimage-unpopular smaller-than-small
        emb-set-cancel
          [of (λa. pair a (?lift (F a))) ‘ {x ∈ univ. ¬ popular-value F (F x)}
              (λa. pair a (?lift (F' a))) ‘ {x ∈ univ. ¬ popular-value F' (F' x)}]
      by blast
    have 6: {x ∈ univ. ¬ popular-value F (F x)} = {x ∈ univ. ¬ popular-value F' (F' x)}
    proof −
      have (λa. first (pair a (?lift (F a)))) ‘ {x ∈ univ. ¬ popular-value F (F x)} =
            (λa. first (pair a (?lift (F' a)))) ‘ {x ∈ univ. ¬ popular-value F' (F' x)} ∧
            (λa. second (pair a (?lift (F a)))) ‘ {x ∈ univ. ¬ popular-value F (F x)} =
            (λa. second (pair a (?lift (F' a)))) ‘ {x ∈ univ. ¬ popular-value F' (F' x)}
        using 5 by (metis image-image)
      thus ?thesis
        using ∗ embeds-pairs is-embedding-of-some-embedding-of by auto
```

```
qed
have 7: ⋀x. x ∈ univ ∧ ¬ popular-value F (F x) ⟹ F x = F′ x
proof −
  fix x
  assume x: x ∈ univ ∧ ¬ popular-value F (F x)
  have ?lift (F x) = ?lift (F′ x)
  proof −
    have ⋀y. ((x, y) ∈ (λx. (x, ?lift (F x))) ‘ {x ∈ univ. ¬ popular-value F (F x)}
              ⟷ y = ?lift (F x)) ∧
            ((x, y) ∈ (λx. (x, ?lift (F′ x))) ‘ {x ∈ univ. ¬ popular-value F (F x)}
              ⟷ y = ?lift (F′ x))
      using x by blast
    moreover have (λx. (x, ?lift (F x))) ‘ {x ∈ univ. ¬ popular-value F (F x)} =
                (λx. (x, ?lift (F′ x))) ‘ {x ∈ univ. ¬ popular-value F (F x)}
    proof −
      have (λx. (x, ?lift (F x))) ‘ {x ∈ univ. ¬ popular-value F (F x)} =
            (λx. (x, ?lift (F′ x))) ‘ {x ∈ univ. ¬ popular-value F′ (F′ x)}
      proof −
        have (λx. (first (pair x (?lift (F x))), second (pair x (?lift (F x)))))
                  ‘ {x ∈ univ. ¬ popular-value F (F x)} =
              (λx. (first (pair x (?lift (F′ x))), second (pair x (?lift (F′ x)))))
                  ‘ {x ∈ univ. ¬ popular-value F′ (F′ x)}
        proof −
          have (λx. (first x, second x)) ‘ (λa. pair a (?lift (F a)))
                ‘ {x ∈ univ. ¬ popular-value F (F x)} =
              (λx. (first x, second x)) ‘ (λa. pair a (?lift (F′ a)))
                ‘ {x ∈ univ. ¬ popular-value F′ (F′ x)}
            using 5 by argo
          thus ?thesis by blast
        qed
        thus ?thesis
          using ∗ some-pairing-cancel by auto
      qed
      thus ?thesis
        using 6 by blast
    qed
    ultimately show ?thesis by fastforce
  qed
  thus F x = F′ x
    by (metis EF-apply(1) F F′ Int-iff lift-cancel x)
qed
show F = F′
proof
  fix x
  show F x = F′ x
  proof (cases x ∈ univ)
    case False
    show ?thesis
      using F F′ False EF-def
```

```
          by (metis EF-apply(2) IntE)
        next
        assume x: x ∈ univ
        show ?thesis
        proof (cases popular-value F (F x))
          case False
          show ?thesis
            using 7 False x by blast
          next
          case True
          show ?thesis
          proof −
            have F x = some-popular-value F
              by (metis (mono-tags, lifting) CollectD Collect-mono F IntE True
                  small-preimage-unpopular smallness.smaller-than-small smallness-axioms)
            moreover have F′ x = some-popular-value F′
            proof −
              have popular-value F′ (F′ x)
                using True x 6 by blast
              thus ?thesis
                by (metis (mono-tags, lifting) CollectD Collect-mono F′ IntE
                    small-preimage-unpopular smallness.smaller-than-small smallness-axioms)
            qed
            moreover have some-popular-value F = some-popular-value F′
              using F F′ 4 calculation lift-cancel range-F range-subsetD
              by (metis (no-types, opaque-lifting))
            ultimately show ?thesis by auto
          qed
        qed
      qed
    qed
  qed
qed

definition some-embedding-of-small-functions :: ('U ⇒ 'U) ⇒ '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
  unfolding some-embedding-of-small-functions-def
  using embeds-SEF is-embedding-of-some-embedding-of by blast

lemma some-embedding-of-small-functions-in-univ [intro, simp]:
assumes F ∈ SEF
shows some-embedding-of-small-functions F ∈ univ
  using assms some-embedding-of-small-functions-is-embedding by blast

lemma some-embedding-of-small-functions-cancel:
assumes F ∈ SEF and F′ ∈ SEF
```

```
  and some-embedding-of-small-functions F = some-embedding-of-small-functions F′
  shows F = F′
    using assms some-embedding-of-small-functions-is-embedding
    by (meson inj-onD)

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
    obtain γ where γ: bij-betw γ univ univ′
      using assms eqpoll-def by blast
    show ∃ι. inj-on ι ({None} ∪ Some ‘ univ′) ∧ ι ‘ ({None} ∪ Some ‘ univ′) ⊆ univ′
    proof −
      let ?ι = λ None ⇒ γ (some-lifting None)
                | Some x ⇒ γ (some-lifting (Some (inv-into univ γ x)))
      have ?ι ‘ ({None} ∪ Some ‘ univ′) ⊆ univ′
        using γ is-embedding-of-some-embedding-of bij-betw-apply
        apply auto[1]
         apply fastforce
        by (simp add: bij-betw-imp-surj-on inv-into-into)
      moreover have inj-on ?ι ({None} ∪ Some ‘ univ′)
      proof
        fix x y
        assume x: x ∈ {None} ∪ Some ‘ univ′
        assume y: y ∈ {None} ∪ Some ‘ univ′
        assume eq: ?ι x = ?ι y
        show x = y
          using x y eq γ some-lifting-cancel
          apply auto[1]
          by (metis bij-betw-def inv-into-f-eq inv-into-into inv-into-injective
      inv-into-into some-lifting-in-univ(1,2))+
  qed
  ultimately show ?thesis by blast
qed
show ∃ι. inj-on ι (univ′ × univ′) ∧ ι ‘ (univ′ × univ′) ⊆ univ′
proof −
  let ?ι = λx. γ (some-pairing (inv-into univ γ (fst x), inv-into univ γ (snd x)))
  have ?ι ‘ (univ′ × univ′) ⊆ univ′
  proof −
    have ⋀x. x ∈ univ′ × univ′ ⟹ ?ι x ∈ univ′
      by (metis γ bij-betw-def imageI inv-into-into mem-Times-iff some-pairing-in-univ)
    thus ?thesis by blast
  qed
  moreover have inj-on ?ι (univ′ × univ′)
  proof
    fix x y
    assume x: x ∈ univ′ × univ′ and y: y ∈ univ′ × univ′
    assume eq: ?ι x = ?ι y
    show x = y
    proof −
      have pair (inv-into univ γ (fst x)) (inv-into univ γ (snd x)) =
            pair (inv-into univ γ (fst y)) (inv-into univ γ (snd y))
      proof −
        have inv-into univ γ (fst x) ∈ univ ∧ inv-into univ γ (snd x) ∈ univ ∧
              inv-into univ γ (fst y) ∈ univ ∧ inv-into univ γ (snd y) ∈ univ
          by (metis γ bij-betw-imp-surj-on inv-into-into mem-Times-iff x y)
        thus ?thesis
          by (metis γ bij-betw-inv-into-left eq some-pairing-in-univ)
      qed
      hence inv-into univ γ (fst x) = inv-into univ γ (fst y) ∧
             inv-into univ γ (snd x) = inv-into univ γ (snd y)
        using x y eq γ
        by (metis bij-betw-imp-surj-on first-conv inv-into-into mem-Times-iff second-conv)
      hence fst x = fst y ∧ snd x = snd y
        by (metis (full-types) γ bij-betw-inv-into-right mem-Times-iff x y)
      thus x = y
        by (simp add: prod-eq-iff)
    qed
  qed
  ultimately show ?thesis by blast
qed
show ∃ι. inj-on ι {X. X ⊆ univ′ ∧ small X} ∧ ι ‘ {X. X ⊆ univ′ ∧ small X} ⊆ univ′
proof −
  let ?ι = λX. γ (emb-set (inv-into univ γ ‘ X))
  have ?ι ‘ {X. X ⊆ univ′ ∧ small X} ⊆ univ′
  proof
    fix X′
    assume X′: X′ ∈ ?ι ‘ {X. X ⊆ univ′ ∧ small X}
    obtain X where X: X ⊆ univ′ ∧ small X ∧ ?ι X = X′
```

```
        using X′ by blast
      have ?ι X ∈ univ′
        by (metis X γ bij-betw-def bij-betw-inv-into imageI image-mono emb-set-in-univ
            small-image)
      thus X′ ∈ univ′
        using X by blast
    qed
    moreover have inj-on ?ι {X. X ⊆ univ′ ∧ small X}
    proof
      fix X X′
      assume X: X ∈ {X. X ⊆ univ′ ∧ small X}
      assume X′: X′ ∈ {X. X ⊆ univ′ ∧ small X}
      assume eq: ?ι X = ?ι X′
      show X = X′
      proof −
         have emb-set (inv-into univ γ ‘ X) = emb-set (inv-into univ γ ‘ X′)
         proof −
           have emb-set (inv-into univ γ ‘ X) ∈ univ ∧ emb-set (inv-into univ γ ‘ X′) ∈ univ
             by (metis (no-types, lifting) Int-Collect Int-iff X X′ γ bij-betw-def
                 bij-betw-inv-into powering.emb-set-in-univ powering-axioms small-image
                 subset-image-iff)
           thus ?thesis
             by (metis γ bij-betw-inv-into-left eq)
         qed
         hence inv-into univ γ ‘ X = inv-into univ γ ‘ X′
           by (metis (no-types, lifting) Int-Collect Int-iff X X′ γ bij-betw-def
               bij-betw-inv-into powering.emb-set-cancel powering-axioms small-image
               subset-image-iff)
         thus ?thesis
           by (metis X X′ γ bij-betw-imp-surj-on image-inv-into-cancel mem-Collect-eq)
       qed
    qed
    ultimately show ?thesis by blast
  qed
qed
```

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

```
sublocale small-funcset-and-nat ..

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 −
  show is-embedding-of some-embedding-of-lists {l. List.set l ⊆ univ}
```

```
proof
  show *: some-embedding-of-lists ‘ {l. list.set l ⊆ univ} ⊆ univ
  proof −
    have ⋀l. List.set l ⊆ univ ⟹ some-embedding-of-lists l ∈ univ
    proof −
      fix l
      show List.set l ⊆ univ ⟹ some-embedding-of-lists l ∈ univ
        by (induct l) auto
    qed
    thus ?thesis by blast
  qed
  show inj-on some-embedding-of-lists {l. list.set l ⊆ univ}
  proof −
    have ⋀n l m. ⟦l ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
                  m ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
                  some-embedding-of-lists l = some-embedding-of-lists m⟧
                    ⟹ l = m
    proof −
      fix n l m
      show ⟦l ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
            m ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
            some-embedding-of-lists l = some-embedding-of-lists m⟧
              ⟹ l = m
      proof (induct n arbitrary: l m)
        show ⋀l m. ⟦l ∈ {l. list.set l ⊆ univ ∧ length l ≤ 0};
                    m ∈ {l. list.set l ⊆ univ ∧ length l ≤ 0};
                    some-embedding-of-lists l = some-embedding-of-lists m⟧
                      ⟹ l = m
          by auto
        fix n l m
        assume ind: ⋀l m. ⟦l ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
                           m ∈ {l. list.set l ⊆ univ ∧ length l ≤ n};
                           some-embedding-of-lists l = some-embedding-of-lists m⟧
                             ⟹ l = m
        assume l: l ∈ {l. list.set l ⊆ univ ∧ length l ≤ Suc n}
        assume m: m ∈ {l. list.set l ⊆ univ ∧ length l ≤ Suc n}
        assume eq: some-embedding-of-lists l = some-embedding-of-lists m
        show l = m
        proof (cases l; cases m)
          show ⟦l = []; m = []⟧ ⟹ l = m by simp
          show ⋀a m'. ⟦l = []; m = a # m'⟧ ⟹ l = m
            by (metis (no-types, lifting) * eq image-subset-iff insert-subset
                list.simps(15) m mem-Collect-eq some-pairing-in-univ
                some-embedding-of-lists.simps(1,2) some-lifting-cancel(1))
          show ⋀a l'. ⟦l = a # l'; m = []⟧ ⟹ l = m
            by (metis (lifting) * eq image-subset-iff l some-lifting-cancel(1)
                list.set-intros(1) mem-Collect-eq some-pairing-in-univ set-subset-Cons
                some-embedding-of-lists.simps(1,2) subset-code(1))
          show ⋀a b l' m'. ⟦l = a # l'; m = b # m'⟧ ⟹ l = m
```

```
            proof –
              fix a b l′ m′
              assume al′: l = a # l′ and bm′: m = b # m′
              have some-pairing (a, some-embedding-of-lists l′) =
                    some-pairing (b, some-embedding-of-lists m′)
                using l m al′ bm′ eq some-lifting-is-embedding embeds-pairs
                apply simp
                by (metis (no-types, lifting) ∗ image-subset-iff mem-Collect-eq
                    some-lifting-cancel(2) some-pairing-in-univ)
              hence a = b ∧ some-embedding-of-lists l′ = some-embedding-of-lists m′
                using l m al′ bm′ embeds-pairs
                by (metis (lifting) ∗ image-subset-iff insert-subset list.simps(15)
                    mem-Collect-eq first-conv second-conv)
              hence a = b ∧ l′ = m′
                using l m al′ bm′ ind by auto
              thus l = m
                using al′ bm′ by auto
            qed
          qed
        qed
      qed
      thus ?thesis
        using inj-on-def [of some-embedding-of-lists {l. list.set l ⊆ univ}]
        by (metis (lifting) linorder-le-cases mem-Collect-eq)
    qed
  qed
  thus embeds {l. List.set l ⊆ univ} by blast
qed
```

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 (λX. 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 –
  show is-embedding-of (λX. some-embedding-of-small-sets (some-embedding-of-lists ‘ X))
          {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X}
  proof
    show inj-on (λX. some-embedding-of-small-sets (some-embedding-of-lists ‘ X))
            {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X}
    proof
      fix X Y :: 'U list set
      assume X: X ∈ {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X}
      and Y: Y ∈ {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X}
      assume eq: some-embedding-of-small-sets (some-embedding-of-lists ‘ X) =
                  some-embedding-of-small-sets (some-embedding-of-lists ‘ Y)
      have some-embedding-of-lists ‘ X = some-embedding-of-lists ‘ Y
        by (metis (mono-tags, lifting) CollectD X Y emb-set-cancel emb-set-def
            embeds-lists(2) eq image-mono small-image subset-trans)
        thus X = Y
          using X Y embeds-lists inj-on-image-eq-iff by fastforce
      qed
      show (λX. some-embedding-of-small-sets (some-embedding-of-lists ‘ X)) ‘
              {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X} ⊆ univ
      proof
        fix X′
        assume X′: X′ ∈ (λX. some-embedding-of {X. X ⊆ univ ∧ small X}
                               (some-embedding-of-lists ‘ X))
                             ‘ {X. X ⊆ {l. set l ⊆ univ} ∧ small X}
        obtain X where X: X ⊆ {l. set l ⊆ univ} ∧ small X ∧
                            (λX. some-embedding-of {X. X ⊆ univ ∧ small X}
                               (some-embedding-of-lists ‘ X)) X = X′
          using X′ by blast
        have some-embedding-of-lists ‘ X ⊆ univ ∧ small (some-embedding-of-lists ‘ X)
          using X embeds-lists small-image by blast
        hence (λX. some-embedding-of {X. X ⊆ univ ∧ small X}
                (some-embedding-of-lists ‘ X)) X ∈ univ
          by (metis emb-set-def emb-set-in-univ)
        thus X′ ∈ univ
          using X by blast
      qed
    qed
    thus embeds {X. X ⊆ {l. list.set l ⊆ univ} ∧ small X} by blast
  qed

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› .
```

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 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
   using assms Fun-def by auto

 lemma Set-some-terminal:
 shows Set some-terminal = {some-terminal}
   using ide-in-hom terminal-def terminal-some-terminal by auto

 lemma Fun-some-terminator:
 assumes ide a
 shows Fun t?[a] = (λx. if x ∈ Set a then 1? else null)
   unfolding Fun-def
   using assms elementary-category-with-terminal-object.trm-naturality
         elementary-category-with-terminal-object.trm-one
         extends-to-elementary-category-with-terminal-object
   by fastforce
```

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
  by (metis assms(1,2) eqpoll-trans inj-on-image-eqpoll-self repleteness-ax small-image-iff)
```

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 −
  have ide (mkide A) ∧ Set (mkide A) ≈ A
    using assms repleteness mkide-def someI-ex
    by (metis (lifting) HOL.ext)
  thus ide (mkide A) and Set (mkide A) ≈ A
    using assms by auto
qed
```

```
lemma bij-OUT:
assumes small A and embeds A
shows bij-betw (OUT A) (Set (mkide A)) A
  unfolding OUT-def
  using assms ide-mkide(2) someI-ex [of λF. bij-betw F (Set (mkide A)) A] eqpoll-def
  by blast

lemma bij-IN:
assumes small A and embeds A
shows bij-betw (IN A) A (Set (mkide A))
  using assms bij-OUT bij-betw-inv-into by blast

lemma OUT-elem-of:
assumes small A and embeds A and «x : 1? → mkide A»
shows OUT A x ∈ A
  by (metis CollectI assms(1,2,3) bij-betw-apply bij-OUT)

lemma IN-in-hom:
assumes small A and embeds A and x ∈ A and a = mkide A
shows «IN A x : 1? → a»
  by (metis (mono-tags, lifting) Ball-Collect assms(1,2,3,4) bij-betw-def bij-OUT
      inv-into-into set-eq-subset)

lemma IN-OUT:
assumes small A and embeds A
shows x ∈ Set (mkide A) ⟹ IN A (OUT A x) = x
  using assms bij-OUT(1)
  by (metis bij-betw-inv-into-left)

lemma OUT-IN:
assumes small A and embeds A
shows x ∈ A ⟹ OUT A (IN A x) = x
  using assms bij-OUT(1)
  by (metis bij-betw-inv-into-right)

lemma Fun-IN:
assumes small A and embeds A and y ∈ A
shows Fun (IN A y) = (λx. if x = 1? then IN A y else null)
proof
  fix x
  show Fun (IN A y) x = (if x = 1? then IN A y else null)
  proof (cases x ∈ Set 1?)
    case False
    show ?thesis
      using False Fun-def
      by (metis IN-in-hom Set-some-terminal assms(1,2,3) in-homE singleton-iff)
    next
    case True
    have x: x = 1^?
      using True Set-some-terminal by blast
    have Fun (IN A y) x = IN A y · 1^?
      using Fun-def dom-eqI ide-some-terminal ext x by auto
    also have ... = (if x = 1^? then IN A y else null)
      by (metis (lifting) HOL.ext IN-in-hom assms(1,2,3) comp-arr-dom in-homE x)
    finally show ?thesis by blast
  qed
qed
```

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 ⇒ 'U ⇒ ('U ⇒ 'U) ⇒ 'U
where mkarr a b F ≡ if ide a ∧ ide b ∧ F ∈ Hom a b
                    then SOME f. «f : a → b» ∧ Fun f = F
                    else null

lemma mkarr-in-hom [intro]:
assumes ide a and ide b and F ∈ Hom a b
shows «mkarr a b F : a → b»
proof −
  have ∃f. «f : a → b» ∧ Fun f = F
    using assms surj-Fun [of a b] by blast
  thus ?thesis
    unfolding mkarr-def
    using assms someI-ex [of λf. «f : a → b» ∧ Fun f = F] by auto
qed

lemma arr-mkarr [intro, simp]:
assumes ide a and ide b and F ∈ Hom a b
shows arr (mkarr a b F)
  using assms mkarr-in-hom by blast

lemma arr-mkarrD [dest]:
assumes arr (mkarr a b F)
shows ide a and ide b and F ∈ Hom a b
  by (metis (lifting) assms mkarr-def not-arr-null)+

lemma arr-mkarrE [elim]:
assumes arr (mkarr a b F)
and ⟦ide a; ide b; F ∈ Hom a b⟧ ⟹ T
shows T
  using assms by auto

lemma dom-mkarr [simp]:
assumes arr (mkarr a b F)
shows dom (mkarr a b F) = a
  by (meson arr-mkarrE assms in-homE mkarr-in-hom)
```

```
lemma cod-mkarr [simp]:
assumes arr (mkarr a b F)
shows cod (mkarr a b F) = b
  by (meson arr-mkarrE assms in-homE mkarr-in-hom)

lemma Fun-mkarr [simp]:
assumes arr (mkarr a b F)
shows Fun (mkarr a b F) = F
proof –
  have ∃f. «f : a → b» ∧ Fun f = F
    using assms surj-Fun [of a b] by blast
  thus ?thesis
    unfolding mkarr-def
    using assms someI-ex [of λf. «f : a → b» ∧ Fun f = F] by auto
qed

lemma mkarr-Fun:
assumes «f : a → b»
shows mkarr a b (Fun f) = f
proof –
  have «mkarr a b (Fun f) : a → b» ∧ Fun (mkarr a b (Fun f)) = Fun f
   by (metis (lifting) Fun-in-Hom Fun-mkarr assms ide-cod ide-dom in-homE mkarr-in-hom)
  thus ?thesis
    using assms inj-Fun inj-onD [of Fun hom a b mkarr a b (Fun f) f]
    by blast
qed
```

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 –
  have 1: Fun ∈ hom a b → Hom a b
    using Fun-in-Hom by blast
  have 2: mkarr a b ∈ Hom a b → hom a b
    using assms mkarr-in-hom by auto
  have 3: ⋀F. F ∈ Hom a b ⟹ Fun (mkarr a b F) = F
    using Fun-mkarr assms(1,2) mkarr-in-hom by auto
  have 4: ⋀f. f ∈ hom a b ⟹ mkarr a b (Fun f) = f
    using assms mkarr-Fun by auto
  show bij-betw Fun (hom a b) (Hom a b)
    using 1 2 3 4
    by (intro bij-betwI) auto
  show bij-betw (mkarr a b) (Hom a b) (hom a b)
    using 1 2 3 4
    by (intro bij-betwI) auto
```

qed

lemma *arr-eqI*:
assumes *par t u* **and** *Fun t* = *Fun u*
shows *t* = *u*
  using *assms* **by** (*metis* (*lifting*) *arr-iff-in-hom mkarr-Fun*)

lemma *arr-eqI′*:
assumes *in-hom f a b* **and** *in-hom g a b*
and $\bigwedge x.$ *in-hom x* $\mathbf{1}^?$ *a* ⟹ *f* · *x* = *g* · *x*
shows *f* = *g*
  using *assms arr-eqI* [*of f g*] *in-homE Fun-def* **by** *fastforce*

lemma *Fun-arr*:
assumes «*f* : *a* → *b*»
shows *Fun f* = (λ*x*. *if x* ∈ *Set a then f* · *x else null*)
  using *assms Fun-def* **by** *auto*

lemma *Fun-ide*:
assumes *ide a*
shows *Fun a* = (λ*x*. *if x* ∈ *Set a then x else null*)
  by (*metis* (*lifting*) *CollectD CollectI assms comp-cod-arr in-homE ide-char Fun-def*)

lemma *Fun-comp*:
assumes *seq t u*
shows *Fun* (*t* · *u*) = *Fun t* ∘ *Fun u*
  unfolding *Fun-def*
  using *assms comp-assoc* **by** *force*

lemma *mkarr-comp*:
assumes *seq g f*
shows *mkarr* (*dom f*) (*cod g*) (*Fun g* ∘ *Fun f*) = *g* · *f*
  by (*metis* (*lifting*) *Fun-comp assms cod-comp dom-comp in-homI mkarr-Fun*)

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*)
  using *assms Fun-mkarr mkarr-comp* [*of mkarr b c G mkarr a b F*] **by** *simp*

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*
  using *assms Fun-mkarr*
  by (*metis Fun-def in-homE mem-Collect-eq*)

lemma *ide-as-mkarr*:
assumes *ide a*
shows *mkarr a a* (λ*x*. *if x* ∈ *Set a then x else null*) = *a*
  using *assms Fun-ide Fun-mkarr*

```
  by (intro arr-eqI) auto
```

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
  show terminal a ⟹ ide a ∧ (∃!x. x ∈ Set a)
    using terminal-def terminal-some-terminal by auto
  assume a: ide a ∧ (∃!x. x ∈ Set a)
  show terminal a
  proof
    show ide a
      using a by blast
    show ⋀b. ide b ⟹ ∃!f. «f : b → a»
    proof −
      fix b
      assume b: ide b
      have «mkarr b a (λx. if x ∈ Set b then THE y. y ∈ Set a else null) : b → a»
        using a b theI [of λy. y ∈ Set a]
        by (intro mkarr-in-hom) fastforce+
      moreover have ⋀t u. ⟦«t : b → a»; «u : b → a»⟧ ⟹ t = u
        using a Fun-def by (intro arr-eqI) fastforce+
      ultimately show ∃!f. «f : b → a» by blast
    qed
  qed
qed
```

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 −
  have ∀b. ide b ⟶ terminal b ⟹ ∀b. ide b ⟶ initial b
    by (simp add: initialI terminal-def)
  moreover have ∃b. ide b ∧ ¬ terminal b ⟹ ∀a. initial a ⟷ ide a ∧ Set a = {}
  proof −
    assume 1: ∃b. ide b ∧ ¬ terminal b
    obtain b where b: ide b ∧ ¬ terminal b
      using 1 by blast
    show ∀a. initial a ⟷ ide a ∧ Set a = {}
    proof (intro allI iffI conjI)
      fix a
      assume a: initial a
      show ide a
        using a initial-def by blast
      show Set a = {}
      proof (cases Set b = {})
```

```
        case True
        show ?thesis
          using a b True by blast
        next
        case False
        have Set a ≠ {} ⟹ ¬ (∃!f. «f : a → b»)
        proof -
          assume 2: Set a ≠ {}
          obtain x y where 3: x ∈ Set b ∧ y ∈ Set b ∧ x ≠ y
            using b False terminal-char by auto
          show ?thesis
          proof -
            have «mkarr a b (λz. if z ∈ Set a then x else null) : a → b»
              using ‹ide a› b 3 by auto
            moreover have «mkarr a b (λz. if z ∈ Set a then y else null) : a → b»
              using ‹ide a› b 3 by auto
            moreover have mkarr a b (λz. if z ∈ Set a then x else null) ≠
                          mkarr a b (λz. if z ∈ Set a then y else null)
              by (metis (full-types, lifting) 2 3 Fun-mkarr arrI calculation(2) ex-in-conv)
            ultimately show ?thesis by auto
          qed
        qed
        thus ?thesis
          using a b initial-def by auto
      qed
      next
      fix a
      assume a: ide a ∧ Set a = {}
      show initial a
      proof -
        have ⋀b. ide b ⟹ ∃!f. «f : a → b»
        proof -
          fix b
          assume b: ide b
          have «mkarr a b (λ-. null) : a → b»
            by (simp add: a b mkarr-in-hom)
          moreover have ⋀f g. ⟦«f : a → b»; «g : a → b»⟧ ⟹ f = g
            using a arr-eqI' by fastforce
          ultimately show ∃!f. «f : a → b» by blast
        qed
        thus ?thesis
          using a initial-def by blast
      qed
    qed
  qed
  ultimately show ?thesis
    by (metis initial-def)
qed
```

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
  assume f: mono f
  have arr f
    using f mono-implies-arr by simp
  moreover have inj-on (Fun f) (Set (dom f))
    by (intro inj-onI)
        (metis Fun-def calculation f in-homE mem-Collect-eq mono-cancel seqI)
  ultimately show arr f ∧ inj-on (Fun f) (Set (dom f)) by blast
  next
  assume f: arr f ∧ inj-on (Fun f) (Set (dom f))
  show mono f
  proof
    show arr f
      using f by blast
    fix g h
    assume seq: seq f g and eq: f · g = f · h
    show g = h
    proof (intro arr-eqI)
      show par: par g h
        by (metis dom-comp eq seq seqE)
      show Fun g = Fun h
      proof −
        have ⋀x. x ∈ Set (dom g) ⟹ Fun g x = Fun h x
        proof −
          fix x
          assume x: x ∈ Set (dom g)
          have f · (g · x) = f · (h · x)
            using eq by (metis comp-assoc)
          moreover have g · x ∈ Set (dom f) ∧ h · x ∈ Set (dom f)
            by (metis seq par comp-in-homI in-homI mem-Collect-eq seq seqE x)
          ultimately have g · x = h · x
            using f inj-on-def [of Fun f Set (dom f)] Fun-def by auto
          thus Fun g x = Fun h x
            using par Fun-def by presburger
        qed
        thus ?thesis
          using par Fun-def by force
      qed
    qed
  qed
qed
```

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

```
lemma retraction-char:
shows retraction f ⟷ arr f ∧ Fun f ‘ Set (dom f) = Set (cod f)
proof (intro iffI conjI)
  assume f: retraction f
```

```
show 1: arr f
  using f by blast
obtain g where g: f · g = cod f
  using f by blast
show Fun f ‘ Set (dom f) = Set (cod f)
proof
  show Fun f ‘ Set (dom f) ⊆ Set (cod f)
    using ‹arr f› Fun-def by auto
  show Set (cod f) ⊆ Fun f ‘ Set (dom f)
  proof −
    have Set (cod f) ⊆ Fun f ‘ Fun g ‘ Set (cod f)
    proof −
      have Set (cod f) ⊆ Fun (cod f) ‘ Set (cod f)
        using 1 Fun-ide by auto
      also have ... = (Fun f ∘ Fun g) ‘ Set (cod f)
        using 1 g Fun-comp
        by (metis (no-types, lifting) arr-cod)
      also have ... = Fun f ‘ Fun g ‘ Set (cod f)
        by (metis image-comp)
      finally show ?thesis by blast
    qed
    also have ... ⊆ Fun f ‘ Set (dom f)
    proof −
      have «g : cod f → dom f»
        using g
        by (metis 1 arr-iff-in-hom ide-cod ide-compE seqE)
      thus ?thesis
        using Fun-def by auto
    qed
    finally show ?thesis by blast
  qed
qed
next
assume f: arr f ∧ Fun f ‘ Set (dom f) = Set (cod f)
let ?G = λy. if y ∈ Set (cod f) then inv-into (Set (dom f)) (Fun f) y else null
let ?g = mkarr (cod f) (dom f) ?G
have f · ?g = cod f
proof (intro arr-eqI)
  have seq: seq f ?g
  proof
    show «f : dom f → cod f»
      using f by blast
    show «?g : cod f → dom f»
    proof (intro mkarr-in-hom)
      show ide (cod f) and ide (dom f)
        using f by auto
      show ?G ∈ (Set (cod f) → Set (dom f)) ∩ {F. ∀x. x ∉ Set (cod f) ⟶ F x = null}
      proof
        show ?G ∈ Set (cod f) → Set (dom f)
```

```
          proof
            fix x
            assume x: x ∈ Set (cod f)
            show ?G x ∈ Set (dom f)
              by (metis f inv-into-into x)
          qed
          show ?G ∈ {F. ∀ x. x ∉ Set (cod f) ⟶ F x = null}
            using f by auto
        qed
      qed
    qed
    thus par: par (f · ?g) (cod f) by auto
    show Fun (f · ?g) = Fun (cod f)
    proof −
      have Fun (f · ?g) = Fun f ∘ ?G
        using par Fun-comp Fun-mkarr by fastforce
      also have ... = Fun (cod f)
      proof
        fix y
        show (Fun f ∘ ?G) y = Fun (cod f) y
        proof (cases y ∈ Set (cod f))
          case False
          show ?thesis
            using False Fun-def dom-cod by auto
          next
          case True
          show ?thesis
          proof −
            have Fun f (inv-into (Set (dom f)) (Fun f) y) = y
              by (metis (no-types) True f f-inv-into-f)
            thus ?thesis
              using Fun-ide True f by force
          qed
        qed
      qed
      finally show ?thesis by blast
    qed
  qed
  thus retraction f
    by (metis (lifting) f ide-cod retraction-def)
qed
```

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

```
lemma iso-char:
shows iso f ⟷ arr f ∧ bij-betw (Fun f) (Set (dom f)) (Set (cod f))
  using retraction-char mono-char bij-betw-def
  by (metis (no-types, lifting) iso-iff-mono-and-retraction)

lemma isomorphic-char:
  shows isomorphic a b ⟷ ide a ∧ ide b ∧ Set a ≈ Set b
  proof
    assume 1: isomorphic a b
    show ide a ∧ ide b ∧ Set a ≈ Set b
      using 1 isomorphic-def iso-char eqpoll-def [of Set a Set b] by auto
    next
    assume 1: ide a ∧ ide b ∧ Set a ≈ Set b
    obtain F where F: bij-betw F (Set a) (Set b)
      using 1 eqpoll-def by blast
    let ?F′ = λx. if x ∈ Set a then F x else null
    let ?f = mkarr a b (λx. if x ∈ Set a then F x else null)
    have f: «?f : a → b»
    proof
      show ide a and ide b
        using 1 by auto
      show (λx. if x ∈ Set a then F x else null) ∈ Hom a b
        using F Pi-mem bij-betw-imp-funcset by fastforce
    qed
    moreover have bij-betw (Fun ?f) (Set a) (Set b)
      using F Fun-mkarr arrI bij-betw-cong f
      apply (unfold bij-betw-def)
      by (auto simp add: inj-on-def)
    ultimately have iso ?f ∧ dom ?f = a ∧ cod ?f = b
      using iso-char Fun-mkarr by auto
    thus isomorphic a b
      using isomorphicI by force
  qed

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) ≈ Collect (partial-composition.arr D)
shows equivalent-categories C D
proof
  interpret smallness sml
    using assms(1) sets-cat-def sets-cat-base-def by blast
  interpret C: sets-cat sml C
    using assms(1) by blast
  interpret D: sets-cat sml D
    using assms(2) by blast
  have D-embeds-C-Set: ⋀a. C.ide a ⟹ D.embeds (C.Set a)
    using assms(3) D.embeds-subset [of Collect C.arr]
  by (metis (no-types, lifting) Collect-mono bij-betw-def C.in-homE eqpoll-def)
let ?F_o = λa. D.mkide (C.Set a)
have F_o: ⋀a. C.ide a ⟹ D.ide (?F_o a)
  by (simp add: C.small-Set D.ide-mkide(1) D-embeds-C-Set)
have bij-OUT: ⋀a. C.ide a ⟹ bij-betw (D.OUT (C.Set a)) (D.Set (?F_o a)) (C.Set a)
  by (simp add: C.small-Set D.bij-OUT(1) D-embeds-C-Set)
let ?F_Fun = λf. λx. if x ∈ D.Set (?F_o (C.dom f))
                     then (D.IN (C.Set (C.cod f)) ∘ C.Fun f ∘ D.OUT (C.Set (C.dom f))) x
                     else D.null
have F_Fun: ⋀f. C.arr f ⟹ ?F_Fun f ∈ D.Hom (?F_o (C.dom f)) (?F_o (C.cod f))
proof
  fix f
  assume f: C.arr f
  show ?F_Fun f ∈ {F. ∀x. x ∉ D.Set (?F_o (C.dom f)) ⟶ F x = D.null}
    by simp
  show ?F_Fun f ∈ D.Set (?F_o (C.dom f)) → D.Set (?F_o (C.cod f))
  proof
    fix x
    assume x: x ∈ D.Set (?F_o (C.dom f))
    show ?F_Fun f x ∈ D.Set (D.mkide (C.Set (C.cod f)))
    proof −
      have D.in-hom (D.IN (C.Set (C.cod f)) (C f (D.OUT (C.Set (C.dom f)) x)))
              D.some-terminal (D.mkide (C.Set (C.cod f)))
      proof −
        have «C f (D.OUT (C.Set (C.dom f)) x) : 1? → C.cod f»
          using x f C.ide-dom bij-betwE bij-OUT by blast
        moreover have small (C.Set (C.cod f))
          using C.small-Set f by force
        moreover have D.embeds (C.Set (C.cod f))
          by (simp add: D-embeds-C-Set f)
        ultimately show ?thesis
          using x f D.bij-IN [of C.Set (C.cod f)] bij-betwE by auto
      qed
      moreover have «D.OUT (C.Set (C.dom f)) x : 1? → C.dom f»
        using x f C.ide-dom bij-betwE bij-OUT by blast
      ultimately show ?thesis
        using x f C.Fun-def by force
    qed
  qed
qed
let ?F = λf. if C.arr f then D.mkarr (?F_o (C.dom f)) (?F_o (C.cod f)) (?F_Fun f) else D.null
interpret functor C D ?F
proof
  show ⋀f. ¬ C.arr f ⟹ ?F f = D.null
    by simp
  show arrF: ⋀f. C.arr f ⟹ D.arr (?F f)
    using F_o F_Fun by auto
  show domF: ⋀f. C.arr f ⟹ D.dom (?F f) = ?F (C.dom f)
  proof −
  fix f
  assume f: C.arr f
  have D.dom (?F f) = D.mkide (C.Set (C.dom f))
    using f arrF by auto
  also have ... = ?F (C.dom f)
  proof −
    have ?F_Fun (C.dom f) =
          (λx. if x ∈ D.Set (D.mkide (C.Set (C.dom f))) then x else D.null)
    proof
      fix x
      have x ∈ D.Set (D.mkide (C.Set (C.dom f))) ⟹
            «D.OUT (C.Set (C.dom f)) x : 1^? → C.dom f»
        using f C.ide-dom bij-betwE bij-OUT by blast
      thus ?F_Fun (C.dom f) x =
            (if x ∈ D.Set (D.mkide (C.Set (C.dom f))) then x else D.null)
        using f C.ide-dom bij-betwE bij-OUT arrF F_o C.Fun-ide
              D.IN-OUT [of C.Set (C.dom f) x]
        by (auto simp add: C.small-Set D-embeds-C-Set)
    qed
    moreover have D.mkide (C.Set (C.dom f)) =
                  D.mkarr (D.mkide (C.Set (C.dom f))) (D.mkide (C.Set (C.dom f)))
                      (λx. if D.in-hom x D.some-terminal (D.mkide (C.Set (C.dom f)))
                           then x else D.null)
      using f arrF F_o D.ide-as-mkarr by auto
    ultimately show ?thesis
      using f by auto
  qed
  finally show D.dom (?F f) = ?F (C.dom f) by blast
qed
show codF: ⋀f. C.arr f ⟹ D.cod (?F f) = ?F (C.cod f)
proof −
  fix f
  assume f: C.arr f
  have D.cod (?F f) = D.mkide (C.Set (C.cod f))
    using f arrF by auto
  also have ... = ?F (C.cod f)
  proof −
    have ?F_Fun (C.cod f) =
          (λx. if x ∈ D.Set (D.mkide (C.Set (C.cod f))) then x else D.null)
    proof
      fix x
      have x ∈ D.Set (D.mkide (C.Set (C.cod f))) ⟹
            «D.OUT (C.Set (C.cod f)) x : 1^? → C.cod f»
        using f C.ide-cod bij-betwE bij-OUT by blast
      thus ?F_Fun (C.cod f) x =
            (if x ∈ D.Set (D.mkide (C.Set (C.cod f))) then x else D.null)
        using f C.ide-cod bij-betwE bij-OUT arrF F_o C.Fun-ide
              D.IN-OUT [of C.Set (C.cod f) x]
        by (auto simp add: C.small-Set D-embeds-C-Set)
    qed
    moreover have D.mkide (C.Set (C.cod f)) =
                   D.mkarr (D.mkide (C.Set (C.cod f))) (D.mkide (C.Set (C.cod f)))
                       (λx. if D.in-hom x D.some-terminal (D.mkide (C.Set (C.cod f)))
                            then x else D.null)
      using f arrF F_o D.ide-as-mkarr [of D.mkide (C.Set (C.cod f))] by auto
    ultimately show ?thesis
      using f by auto
  qed
  finally show D.cod (?F f) = ?F (C.cod f) by blast
qed
fix f g
assume seq: C.seq g f
have f: C.arr f and g: C.arr g
  using seq by auto
show ?F (C g f) = D (?F g) (?F f)
proof (intro D.arr-eqI [of ?F (C g f)])
  show par: D.par (?F (C g f)) (D (?F g) (?F f))
  proof (intro conjI)
    show 1: D.arr (?F (C g f))
      using seq arrF [of C g f] by fastforce
    show 2: D.arr (D (?F g) (?F f))
      using seq arrF domF codF by (intro D.seqI) auto
    show D.dom (?F (C g f)) = D.dom (D (?F g) (?F f))
      using 1 2 by fastforce
    show D.cod (?F (C g f)) = D.cod (D (?F g) (?F f))
      using 1 2 by fastforce
  qed
  show D.Fun (?F (C g f)) = D.Fun (D (?F g) (?F f))
  proof -
    have D.Fun (D (?F g) (?F f)) = D.Fun (?F g) ∘ D.Fun (?F f)
      using seq par D.Fun-comp [of ?F g ?F f] by fastforce
    also have ... = ?F_Fun g ∘ ?F_Fun f
      using f g arrF D.Fun-mkarr by auto
    also have ... = D.Fun (?F (C g f))
    proof
      fix x
      show (?F_Fun g ∘ ?F_Fun f) x = D.Fun (?F (C g f)) x
      proof (cases x ∈ D.Set (D.mkide (C.Set (C.dom f))))
        case False
        show ?thesis
          using False f par by auto
        next
        case True
        have 1: «D.OUT (C.Set (C.dom f)) x : 1^? → C.dom f»
          using True D.OUT-elem-of [of C.Set (C.dom f) x]
                C.ide-dom C.small-Set D-embeds-C-Set f
          by blast
        have (?F_Fun g ∘ ?F_Fun f) x =
                D.IN (C.Set (C.cod g))
                  (C.Fun g
                     (D.OUT (C.Set (C.dom g))
                        (D.IN (C.Set (C.cod f))
                           (C.Fun f
                              (D.OUT (C.Set (C.dom f)) x)))))
          proof –
            have D.in-hom (D.IN (C.Set (C.cod f)) (C f (D.OUT (C.Set (C.dom f)) x)))
                             D.some-terminal (D.mkide (C.Set (C.dom g)))
              using True f seq 1 C.ide-cod C.small-Set D-embeds-C-Set
              by (intro D.IN-in-hom) auto
            thus ?thesis
              using True 1 C.Fun-def by auto
          qed
          also have ... =
                D.IN (C.Set (C.cod g))
                  (C.Fun g
                     (C.Fun f
                        (D.OUT (C.Set (C.dom f)) x)))
            using True 1 seq f g C.small-Set D-embeds-C-Set C.Fun-def D.Fun-def
                  D.OUT-IN [of C.Set (C.dom g) C f (D.OUT (C.Set (C.dom f)) x)]
            by auto[1] (metis C.comp-in-homI' C.in-homE C.seqE)
          also have ... = ?F_Fun (C g f) x
            using True seq 1 C.comp-assoc C.Fun-def D.Fun-def
            by auto[1] fastforce
          also have ... = D.Fun (?F (C g f)) x
            using True par seq D.Fun-mkarr D.app-mkarr D.in-homI by force
          finally show ?thesis by blast
        qed
      qed
      finally show ?thesis by simp
    qed
  qed
qed
interpret F: fully-faithful-and-essentially-surjective-functor C D ?F
proof
  show ⋀f f'. ⟦C.par f f'; ?F f = ?F f'⟧ ⟹ f = f'
  proof –
    fix f f'
    assume par: C.par f f'
    assume eq: ?F f = ?F f'
    show f = f'
    proof (intro C.arr-eqI' [of f])
      show f: «f : C.dom f → C.cod f»
        using par by blast
      show f': «f' : C.dom f → C.cod f»
        using par by auto
      show ⋀x. «x : 1? → C.dom f» ⟹ C f x = C f' x
      proof –
```

```
fix x
assume x: «x : 1? → C.dom f»
have fx: «C f x : 1? → C.cod f» ∧ C.ide (C.dom f) ∧ C.ide (C.cod f)
  by (metis (no-types) C.arrI C.comp-in-homI C.ide-cod C.seqE f x)
have f'x: «C f' x : 1? → C.cod f'» ∧ C.ide (C.dom f') ∧ C.ide (C.cod f')
    by (metis (no-types) C.arrI C.comp-in-homI C.ide-cod C.seqE f' x par)
have 1: D.in-hom (D.IN (C.Set (C.dom f)) x)
          D.some-terminal (D.mkide (C.Set (C.dom f)))
  by (metis C.ide-dom C.small-Set D.IN-in-hom D-embeds-C-Set mem-Collect-eq
      par x)
have C f x = C.Fun f x
  using C.Fun-def x by auto
also have ... = D.OUT (C.Set (C.cod f))
                  (D.IN (C.Set (C.cod f))
                     (C.Fun f
                        (D.OUT (C.Set (C.dom f))
                           (D.IN (C.Set (C.dom f)) x))))
  by (simp add: fx C.small-Set D.OUT-IN D-embeds-C-Set x C.Fun-def)
also have ... = D.OUT (C.Set (C.cod f)) (?F_Fun f (D.IN (C.Set (C.dom f)) x))
  using par 1 by auto
also have ... =
          D.OUT (C.Set (C.cod f)) (D.Fun (?F f) (D.IN (C.Set (C.dom f)) x))
proof -
  have D.arr (?F f)
    using f by blast
  thus ?thesis
    using x f par by auto
qed
also have ... =
          D.OUT (C.Set (C.cod f)) (D.Fun (?F f') (D.IN (C.Set (C.dom f)) x))
  using eq by simp
also have ... = D.OUT (C.Set (C.cod f)) (?F_Fun f' (D.IN (C.Set (C.dom f)) x))
proof -
  have D.arr (?F f')
    using f' by blast
  thus ?thesis
    using x f par by auto
qed
also have ... = D.OUT (C.Set (C.cod f'))
                  (D.IN (C.Set (C.cod f'))
                     (C.Fun f'
                        (D.OUT (C.Set (C.dom f'))
                           (D.IN (C.Set (C.dom f')) x))))
  using par 1 by auto
also have ... = C.Fun f' x
by (metis f'x C.small-Set D.OUT-IN D-embeds-C-Set mem-Collect-eq par x C.Fun-def)
also have ... = C f' x
  using C.Fun-def x par by auto
finally show C f x = C f' x by blast
```

```
    qed
  qed
qed
have ∗: ⋀a. C.ide a ⟹ ?F a = ?F_o a
proof −
  fix a
  assume a: C.ide a
  show ?F a = ?F_o a
  proof −
    have (λx. if D.in-hom x D.some-terminal (D.mkide (C.Set a))
                then (D.IN (C.Set (C.cod a)) ∘ C.Fun a ∘ D.OUT (C.Set (C.dom a))) x
                else D.null) =
          (λx. if D.in-hom x D.some-terminal (D.mkide (C.Set a)) then x else D.null)
    proof
      fix x
      show (if D.in-hom x D.some-terminal (D.mkide (C.Set a))
              then (D.IN (C.Set (C.cod a)) ∘ C.Fun a ∘ D.OUT (C.Set (C.dom a))) x
              else D.null) =
            (if D.in-hom x D.some-terminal (D.mkide (C.Set a)) then x else D.null)
      using a C.Fun-ide D.IN-OUT [of C.Set a] C.small-Set D-embeds-C-Set
      apply auto[1]
      by (metis (lifting) D.OUT-elem-of mem-Collect-eq)
    qed
    thus ?thesis
      using a D.ide-as-mkarr F_o by auto
  qed
qed
show ⋀a b g. ⟦C.ide a; C.ide b; D.in-hom g (?F a) (?F b)⟧
                ⟹ ∃h. «h : a → b» ∧ ?F h = g
proof −
  fix a b g
  assume a: C.ide a and b: C.ide b and g: D.in-hom g (?F a) (?F b)
  have ?F a = ?F_o a
    using a ∗ by blast
  have dom-g: D.dom g = ?F_o a
    using a g ∗ by auto
  have cod-g: D.cod g = ?F_o b
    using b g ∗ by auto
  have Fun-g: D.Fun g ∈ D.Hom (?F_o a) (?F_o b)
    using g D.Fun-in-Hom dom-g cod-g by blast
  let ?H = λx. if x ∈ C.Set a
                then (D.OUT (C.Set b) ∘ D.Fun g ∘ D.IN (C.Set a)) x
                else C.null
  have H: ?H ∈ C.Hom a b
  proof
    show ?H ∈ C.Set a → C.Set b
    proof
      fix x
      assume x: x ∈ C.Set a
      show ?H x ∈ C.Set b
      proof -
        have ?H x = D.OUT (C.Set b) (D.Fun g (D.IN (C.Set a) x))
          using x by simp
        moreover have ... ∈ C.Set b
        proof -
          have D.IN (C.Set a) x ∈ D.Set (?F_o a)
            by (metis (lifting) a bij-betw-iff-bijections bij-betw-inv-into bij-OUT x)
          hence D.Fun g (D.IN (C.Set a) x) ∈ D.Set (?F_o b)
            using Fun-g by blast
          thus ?thesis
            using b C.small-Set D-embeds-C-Set bij-OUT bij-betw-apply D.Fun-def
            by fastforce
        qed
        ultimately show ?thesis by auto
      qed
    qed
    show ?H ∈ {F. ∀x. x ∉ C.Set a ⟶ F x = C.null} by simp
  qed
  let ?h = C.mkarr a b ?H
  have h: «?h : a → b»
    using a b H by blast
  moreover have ?F ?h = g
  proof (intro D.arr-eqI)
    have Fh: D.in-hom (?F ?h) (?F_o a) (?F_o b)
    proof -
      have D.in-hom (?F ?h) (?F a) (?F b)
        using h preserves-hom by blast
      moreover have ?F a = ?F_o a ∧ ?F b = ?F_o b
        using a b * by auto
      ultimately show ?thesis by simp
    qed
    show par: D.par (?F ?h) g
      using Fh h g cod-g dom-g D.in-homE by auto
    show D.Fun (?F ?h) = D.Fun g
    proof
      fix x
      show D.Fun (?F ?h) x = D.Fun g x
      proof (cases x ∈ D.Set (?F_o a))
        case False
        show ?thesis
          using False par D.Fun-def by auto
        next
        case True
        have D.Fun (?F ?h) x = ?F_Fun ?h x
          using True h Fh D.Fun-def D.app-mkarr by auto
        also have ... = (if x ∈ D.Set (?F_o a)
                         then (D.IN (C.Set b) ∘ C.Fun ?h ∘ D.OUT (C.Set a)) x
                         else D.null)
```

```
            using h by auto
          also have ... = D.IN (C.Set b) (?H (D.OUT (C.Set a) x))
            using True h C.app-mkarr by auto
          also have ... = D.IN (C.Set b)
                           (D.OUT (C.Set b)
                              (D.Fun g
                                 (D.IN (C.Set a)
                                    (D.OUT (C.Set a) x))))
          proof −
            have D.OUT (C.Set a) x ∈ C.Set a
              using True a bij-betw-apply bij-OUT by force
            thus ?thesis by simp
          qed
          also have ... = D.Fun g x
            using True a b g D.IN-OUT [of C.Set a x] D.IN-OUT [of C.Set b D.Fun g x]
                  C.small-Set D-embeds-C-Set dom-g cod-g D.Fun-def
            by auto
          finally show ?thesis by blast
        qed
      qed
    qed
    ultimately show ∃ h. «h : a → b» ∧ ?F h = g by blast
  qed
  show ⋀b. D.ide b ⟹ ∃ a. C.ide a ∧ D.isomorphic (?F a) b
  proof −
    fix b
    assume b: D.ide b
    let ?a = C.mkide (D.Set b)
    have 1: C.ide ?a ∧ C.Set ?a ≈ D.Set b
    proof −
      have ∃ ι. C.is-embedding-of ι (D.Set b)
        by (metis (no-types, lifting) D.in-homE Set.basic-monos(6) assms(3)
            bij-betw-def bij-betw-inv-into eqpoll-def image-mono inj-on-subset)
      thus ?thesis
        using b C.ide-mkide [of D.Set b] D.small-Set by force
    qed
    have D.Set (?F ?a) ≈ D.Set b
    proof −
      have ⋀a. C.ide a ⟹ D.Set (?F a) ≈ C.Set a
        using ∗ C.small-Set D-embeds-C-Set D.ide-mkide(2) by fastforce
      thus ?thesis
        using 1 eqpoll-trans by blast
    qed
    moreover have ⋀a. C.ide a ⟹ D.isomorphic (?F a) b ⟷ D.Set (?F a) ≈ D.Set b
      using D.isomorphic-char b preserves-ide by force
    ultimately show ∃ a. C.ide a ∧ D.isomorphic (?F a) b
      using 1 by blast
  qed
qed
```

```
  show equivalence-functor C D ?F
    using F.is-equivalence-functor by blast
qed
```

4.3 Well-Pointedness

```
context sets-cat
begin

  lemma is-well-pointed:
  assumes par f g and ⋀x. x ∈ Set (dom f) ⟹ f · x = g · x
  shows f = g
    by (metis CollectI arr-eqI' assms(1,2) in-homI)

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
    using two-def ide-mkide embeds-bool-ax small-finite UNIV-bool
          finite.simps insert-commute infinite-imp-nonempty finite.emptyI
          bij-IN [of {True, False}] bij-OUT [of {True, False}]
    by metis+

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

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

lemma *tt-in-hom* [*intro*]:
shows «*tt* : $\mathbf{1}^?$ → **2**»
using *bij-betwE tt-def* **by** *force*

lemma *ff-in-hom* [*intro*]:
shows «*ff* : $\mathbf{1}^?$ → **2**»
using *bij-betwE ff-def* **by** *force*

lemma *tt-simps* [*simp*]:
shows *arr tt* **and** *dom tt* = $\mathbf{1}^?$ **and** *cod tt* = **2**
using *tt-in-hom* **by** *blast+*

lemma *ff-simps* [*simp*]:
shows *arr ff* **and** *dom ff* = $\mathbf{1}^?$ **and** *cod ff* = **2**
using *ff-in-hom* **by** *blast+*

lemma *Fun-tt*:
shows *Fun tt* = ($\lambda x.$ *if x* ∈ *Set* $\mathbf{1}^?$ *then tt else null*)
unfolding *Fun-def*
using *tt-def*
by (*metis Set-some-terminal comp-arr-dom emptyE insertE tt-simps*(*1*,*2*))

lemma *Fun-ff*:
shows *Fun ff* = ($\lambda x.$ *if x* ∈ *Set* $\mathbf{1}^?$ *then ff else null*)
unfolding *Fun-def*
using *ff-def*
by (*metis Set-some-terminal comp-arr-dom emptyE insertE ff-simps*(*1*,*2*))

lemma *mono-tt*:
shows *mono tt*
using *Fun-tt mono-char*
by (*metis point-is-mono terminal-some-terminal tt-simps*(*1*,*2*))

lemma *mono-ff*:
shows *mono ff*
using *Fun-ff mono-char*
by (*metis point-is-mono terminal-some-terminal ff-simps*(*1*,*2*))

lemma *tt-ne-ff*:
shows *tt* ≠ *ff*
using *tt-def ff-def two-def*
by (*metis bij-betw-inv-into-right ide-two*(*3*) *iso-tuple-UNIV-I*)

lemma *Set-two*:
shows *Set* **2** = {*tt*, *ff*}
proof –
have *Set* **2** = *IN* {*True*, *False*} ‘ *UNIV*
using *bij-betw-imp-surj-on* **by** *blast*

```
  thus ?thesis
    using tt-def ff-def
    by (simp add: UNIV-bool insert-commute)
qed
```

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
  show arr f ∧ Fun f ` Set (dom f) = Set (cod f) ⟹ epi f
    using retraction-char retraction-is-epi by presburger
  assume f: epi f
  show arr f ∧ Fun f ` Set (dom f) = Set (cod f)
  proof (intro conjI)
    show arr f
      using epi-implies-arr f by blast
    show Fun f ` Set (dom f) = Set (cod f)
    proof
      show Fun f ` Set (dom f) ⊆ Set (cod f)
        using ‹arr f› Fun-def by auto
      show Set (cod f) ⊆ Fun f ` Set (dom f)
      proof
        fix y
        assume y: y ∈ Set (cod f)
        have y ∉ Fun f ` Set (dom f) ⟹ False
        proof −
          assume 1: y ∉ Fun f ` Set (dom f)
          let ?G = λz. if z ∈ Set (cod f) then if z = y then tt else ff else null
          let ?G' = λz. if z ∈ Set (cod f) then ff else null
          let ?g = mkarr (cod f) 2 ?G
          let ?g' = mkarr (cod f) 2 ?G'
          have g: «?g : cod f → 2»
            using f epi-implies-arr ide-two
            by (intro mkarr-in-hom) auto
          have g': «?g' : cod f → 2»
            using f epi-implies-arr ide-two
            by (intro mkarr-in-hom) auto
          have ?g ≠ ?g'
          proof −
            have ?g · y ≠ ?g' · y
              using app-mkarr g g' tt-ne-ff y by auto
            thus ?thesis by auto
          qed
          moreover have ?g · f = ?g' · f
          proof −
            have ?G ∘ Fun f = ?G' ∘ Fun f
            proof
              fix x
        show (?G ∘ Fun f) x = (?G' ∘ Fun f) x
         using 1 tt-ne-ff Fun-def by auto
       qed
       thus ?thesis
        using f g g' Fun-mkarr ‹arr f› in-homI Fun-comp
        by (intro arr-eqI) auto
      qed
      ultimately show False
       using f g g' ‹arr f› epi-cancel by blast
     qed
     thus y ∈ Fun f ‘ Set (dom f) by blast
    qed
   qed
  qed
 qed

 corollary epis-split:
 assumes epi e
 shows ∃ m. e · m = cod e
  using assms epi-char_SCB retraction-char
  by (meson ide-compE retraction-def)

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 $f \cdot x = g \cdot x$. 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-equ f g ≡ {x. x ∈ Set (dom f) ∧ f · x = g · x}

  definition dom-equ
  where dom-equ f g ≡ mkide (Dom-equ f g)

  abbreviation Equ
  where Equ f g ≡ λx. if x ∈ Set (dom-equ f g) then OUT (Dom-equ f g) x else null

  definition equ
  where equ f g ≡ mkarr (dom-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 ⋀x. x ∈ Set (dom-equ f g) ⟹ OUT (Dom-equ f g) x ∈ Set (dom f)
  and ⋀y. y ∈ Dom-equ f g ⟹ IN (Dom-equ f g) y ∈ Set (dom-equ f g)
  and ⋀x. x ∈ Set (dom-equ f g) ⟹ IN (Dom-equ f g) (OUT (Dom-equ f g) x) = x
  and ⋀y. y ∈ Dom-equ f g ⟹ OUT (Dom-equ f g) (IN (Dom-equ f g) y) = y
  proof −
    have 1: small (Dom-equ f g)
      by (metis (full-types) assms ide-dom small-Collect small-Set)
    have 2: embeds (Dom-equ f g)
      by (metis (no-types, lifting) Collect-mono arrI image-ident mem-Collect-eq
          subset-image-inj)
    show ide (dom-equ f g)
      by (unfold dom-equ-def, intro ide-mkide) fact+
    show 3: bij-betw (OUT (Dom-equ f g)) (Set (dom-equ f g)) (Dom-equ f g)
      unfolding dom-equ-def
      using assms ide-mkide bij-OUT 1 2 by auto
    show 4: bij-betw (IN (Dom-equ f g)) (Dom-equ f g) (Set (dom-equ f g))
      unfolding dom-equ-def
      using assms ide-mkide bij-OUT bij-IN 1 2 by fastforce
    show ⋀x. x ∈ Set (dom-equ f g) ⟹ OUT (Dom-equ f g) x ∈ Set (dom f)
      by (metis (no-types, lifting) 3 CollectD bij-betw-apply)
    show ⋀y. y ∈ Dom-equ f g ⟹ IN (Dom-equ f g) y ∈ Set (dom-equ f g)
      by (metis (no-types, lifting) 4 bij-betw-apply)
    show ⋀x. x ∈ Set (dom-equ f g) ⟹ IN (Dom-equ f g) (OUT (Dom-equ f g) x) = x
      using 1 2 IN-OUT dom-equ-def by auto
    show ⋀y. y ∈ Dom-equ f g ⟹ OUT (Dom-equ f g) (IN (Dom-equ f g) y) = y
      using 1 2 OUT-IN by force
  qed

  lemma Equ-in-Hom [intro]:
  assumes par f g
  shows Equ f g ∈ Hom (dom-equ f g) (dom f)
  proof
    show Equ f g ∈ Set (dom-equ f g) → Set (dom f)
      using assms ide-dom-equ(4) by auto
    show Equ f g ∈ {F. ∀ x. x ∉ Set (dom-equ f g) ⟶ F x = null}
      by simp
  qed

  lemma equ-in-hom [intro, simp]:
  assumes par f g
  shows «equ f g : dom-equ f g → dom f»
    using assms ide-dom-equ Equ-in-Hom
    unfolding equ-def
    by (intro mkarr-in-hom) auto

  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
    using assms equ-in-hom by blast+

  lemma Fun-equ:
  assumes par f g
  shows Fun (equ f g) = Equ f g
  proof –
    have arr (equ f g)
      using assms by auto
    thus ?thesis
      unfolding equ-def
      using assms Fun-mkarr by auto
  qed

  lemma equ-equalizes:
  assumes par f g
  shows f · equ f g = g · equ f g
  proof (intro arr-eqI [of f · equ f g])
    show par: par (f · equ f g) (g · equ f g)
      using assms by auto
    show Fun (f · equ f g) = Fun (g · equ f g)
    proof
      fix x
      show Fun (f · equ f g) x = Fun (g · equ f g) x
      proof (cases x ∈ Set (dom-equ f g))
        case False
        show ?thesis
          using assms False Fun-equ Fun-def by simp
        next
        case True
        show ?thesis
        proof -
          have Fun (f · equ f g) x = Fun f (Fun (equ f g) x)
            using assms Fun-comp comp-in-homI equ-in-hom comp-assoc by auto
          also have ... = Fun f (OUT (Dom-equ f g) x)
            using assms True Fun-equ by simp
          also have ... = f · (OUT (Dom-equ f g) x)
            using Fun-def True assms ide-dom-equ(4) by simp
          also have ... = g · (OUT (Dom-equ f g) x)
            using assms True ide-dom-equ(2) [of f g] bij-betw-apply by force
          also have ... = Fun g (Fun (equ f g) x)
            using assms True Fun-def Fun-equ ide-dom-equ by simp
          also have ... = Fun (g · equ f g) x
            using assms Fun-comp comp-in-homI equ-in-hom comp-assoc by auto
          finally show ?thesis by blast
        qed
      qed
    qed
  qed

  lemma equ-is-equalizer:
  assumes par f g
  shows has-as-equalizer f g (equ f g)
  proof
    show par f g by fact
    show 0: seq f (equ f g)
      using assms by auto
    show f · equ f g = g · equ f g
      using assms equ-equalizes by blast
    show ⋀e'. ⟦seq f e'; f · e' = g · e'⟧ ⟹ ∃!h. equ f g · h = e'
    proof -
      fix e'
      assume seq: seq f e' and eq: f · e' = g · e'
      let ?H = λx. if x ∈ Set (dom e') then IN (Dom-equ f g) (e' · x) else null
      have H: ?H ∈ Hom (dom e') (dom-equ f g)
      proof
        show ?H ∈ {F. ∀x. x ∉ Set (dom e') ⟶ F x = null} by simp
        show ?H ∈ Set (dom e') → Set (dom-equ f g)
        proof
          fix x
          assume x: x ∈ Set (dom e')
          have ?H x = IN (Dom-equ f g) (e' · x)
            using x by simp
          moreover have ... ∈ Set (dom-equ f g)
            using assms seq x ide-dom-equ(5)
        by (metis (mono-tags, lifting) CollectD CollectI arr-iff-in-hom
            comp-in-homI eq local.comp-assoc seqE)
      ultimately show ?H x ∈ Set (dom-equ f g) by auto
    qed
  qed
  let ?h = mkarr (dom e') (dom-equ f g) ?H
  have h: «?h : dom e' → dom-equ f g»
    using assms H seq ide-dom-equ
    by (intro mkarr-in-hom) auto
  have ∗: equ f g · ?h = e'
  proof (intro arr-eqI' [of equ f g · ?h])
    show 1: «equ f g · ?h : dom e' → dom f»
      using assms h by blast
    show e': «e' : dom e' → dom f»
      by (metis arr-iff-in-hom seq seqE)
    show ⋀x. «x : 1^? → dom e'» ⟹ (equ f g · ?h) · x = e' · x
    proof −
      fix x
      assume x: «x : 1^? → dom e'»
      have (equ f g · ?h) · x = equ f g · ?h · x
        using comp-assoc by blast
      also have ... = equ f g · ?H x
        using app-mkarr h x by presburger
      also have ... = OUT (Dom-equ f g) (IN (Dom-equ f g) (e' · x))
      proof −
        have ?H x ∈ Set (dom-equ f g)
          using 1 x by blast
        thus ?thesis
          using assms x equ-in-hom app-mkarr
          by (simp add: assms equ-def)
      qed
      also have ... = e' · x
      proof −
        have e' · x ∈ Dom-equ f g
          by (metis (mono-tags, lifting) e' comp-in-homI eq comp-assoc
              mem-Collect-eq x)
        thus ?thesis
          using assms ide-dom-equ(7) [of f g e' · x] by blast
      qed
      finally show (equ f g · ?h) · x = e' · x by blast
    qed
  qed
  moreover have ⋀h'. equ f g · h' = e' ⟹ h' = ?h
  proof −
    fix h'
    assume h': equ f g · h' = e'
    show h' = ?h
    proof (intro arr-eqI' [of h' - - ?h])
      show 1: «h' : dom e' → dom-equ f g»
```

```
          by (metis arr-iff-in-hom assms comp-in-homE equ-simps(2) h′ in-homE seq)
        show «?h : dom e′ → dom-equ f g»
          using h by blast
        show ⋀x. «x : 1? → dom e′» ⟹ h′ · x = ?h · x
        proof −
          fix x
          assume x: «x : 1? → dom e′»
          have 3: h′ · x = IN (Dom-equ f g) (Equ f g (h′ · x))
            using assms h′ x 1 seq eq ide-dom-equ(6) comp-in-homI in-homI
            by auto
          also have 4: ... = IN (Dom-equ f g) (Fun (equ f g) (h′ · x))
            using assms Fun-equ [of f g]
            by (metis (lifting))
          also have 5: ... = IN (Dom-equ f g) (equ f g · (h′ · x))
            using Fun-def
            by (metis (no-types, lifting) x CollectI comp-in-homI
                dom-comp h′ in-homI seq seqE)
          also have ... = IN (Dom-equ f g) ((equ f g · h′) · x)
            using comp-assoc by simp
          also have ... = IN (Dom-equ f g) ((equ f g · ?h) · x)
            using h h′ eq ∗ by argo
          also have ... = IN (Dom-equ f g) (equ f g · (?h · x))
            using comp-assoc by simp
          also have ... = IN (Dom-equ f g) (Fun (equ f g) (?h · x))
              using x Fun-def app-mkarr h h′ comp-assoc 3 4 5 by auto
          also have ... = IN (Dom-equ f g) (Equ f g (?h · x))
            using assms Fun-equ by (metis (lifting))
          also have ... = ?h · x
            using assms x ide-dom-equ(6) h by auto
          finally show h′ · x = ?h · x by blast
        qed
      qed
    qed
    ultimately show ∃!h. equ f g · h = e′ by auto
  qed
qed

lemma has-equalizers:
assumes par f g
shows ∃ e. has-as-equalizer f g e
 using assms equ-is-equalizer by blast

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 ..

  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 ⋀x. x ∈ Set (dom (equ f g)) ⟹ IN (Equ f g) (OUT (Equ f g) x) = x
  and ⋀y. y ∈ Equ f g ⟹ OUT (Equ f g) (IN (Equ f g) y) = y
    using assms Equ.ide-dom-equ [of f g] Equ.equ-simps(2) [of f g] by auto

  lemma equ-is-equalizer:
  assumes par f g
  shows has-as-equalizer f g (equ f g)
    using assms Equ.equ-is-equalizer by blast

  lemma Fun-equ:
  assumes par f g
  shows Fun (equ f g) = (λx. if x ∈ Set (dom (equ f g))
                             then OUT {x. x ∈ Set (dom f) ∧ f · x = g · x} x
                             else null)
    using assms Equ.Fun-equ by auto

  lemma has-equalizers:
  assumes par f g
  shows ∃ e. has-as-equalizer f g e
    using assms Equ.has-equalizers by blast

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)
    using assms small-Set by fastforce

  lemma embeds-product-sets:
  assumes ide a and ide b
  shows embeds (Set a × Set b)
  proof −
    have Set a × Set b ⊆ Collect arr × Collect arr
      using assms small-Set by auto
    thus ?thesis
      using assms embeds-pairs
      by (meson image-mono inj-on-subset subset-trans)
  qed
```

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 ≡ mkide (Set a × 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 ⋀x. x ∈ Set (prod_o a b) ⟹ OUT (Set a × Set b) x ∈ Set a × Set b
and ⋀y. y ∈ Set a × Set b ⟹ IN (Set a × Set b) y ∈ Set (prod_o a b)
and ⋀x. x ∈ Set (prod_o a b) ⟹ IN (Set a × Set b) (OUT (Set a × Set b) x) = x
and ⋀y. y ∈ Set a × Set b ⟹ OUT (Set a × Set b) (IN (Set a × Set b) y) = y
proof −
  have 1: small (Set a × Set b)
    using assms ide-char small-Set small-product by metis
  moreover have 2: is-embedding-of some-pairing (Set a × Set b)
  proof −
    have Set a × Set b ⊆ Collect arr × Collect arr
      using assms ide-char small-Set by blast
    thus ?thesis
      using assms some-pairing-is-embedding
      by (meson image-mono inj-on-subset subset-trans)
  qed
  ultimately show ide (prod_o a b)
  and 3: bij-betw (OUT (Set a × Set b)) (Set (prod_o a b)) (Set a × Set b)
    unfolding prod_o-def
    using assms ide-mkide bij-OUT by blast+
  show 4: bij-betw (IN (Set a × Set b)) (Set a × Set b) (Set (prod_o a b))
    using ‹bij-betw (OUT (Set a × Set b)) (Set (prod_o a b)) (Set a × Set b)›
          bij-betw-inv-into prod_o-def
    by auto
  show ⋀x. x ∈ Set (prod_o a b) ⟹ OUT (Set a × Set b) x ∈ Set a × Set b
    using 3 bij-betwE by blast
  show ⋀y. y ∈ Set a × Set b ⟹ IN (Set a × Set b) y ∈ Set (prod_o a b)
    using 4 bij-betwE by blast
  show ⋀x. x ∈ Set (prod_o a b) ⟹ IN (Set a × Set b) (OUT (Set a × Set b) x) = x
    using 1 2 IN-OUT prod_o-def by auto
  show ⋀y. y ∈ Set a × Set b ⟹ OUT (Set a × Set b) (IN (Set a × Set b) y) = y
    by (metis 1 2 OUT-IN)
qed
```

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 ≡ λx. if x ∈ Set (prod_o a b) then snd (OUT (Set a × Set b) x) else null

abbreviation P_1 :: 'U ⇒ 'U ⇒ 'U ⇒ 'U
where P_1 a b ≡ λx. if x ∈ Set (prod_o a b) then fst (OUT (Set a × Set b) x) else null
```

```
lemma P_0-in-Hom:
assumes ide a and ide b
shows P_0 a b ∈ Hom (prod_o a b) b
proof
  show P_0 a b ∈ Set (prod_o a b) → Set b
  proof
    fix x
    assume x: x ∈ Set (prod_o a b)
    have OUT (Set a × Set b) x ∈ Set a × Set b
      using assms x bij-betwE ide-prod_o(2) by blast
    thus P_0 a b x ∈ Set b
      using assms x by force
  qed
  show P_0 a b ∈ {F. ∀x. x ∉ Set (prod_o a b) ⟶ F x = null}
    by simp
qed

lemma P_1-in-Hom:
assumes ide a and ide b
shows P_1 a b ∈ Hom (prod_o a b) a
proof
  show P_1 a b ∈ Set (prod_o a b) → Set a
  proof
    fix x
    assume x: x ∈ Set (prod_o a b)
    have OUT (Set a × Set b) x ∈ Set a × Set b
      using assms x bij-betwE ide-prod_o(2) by blast
    thus P_1 a b x ∈ Set a
      using assms x by force
  qed
  show P_1 a b ∈ {F. ∀x. x ∉ Set (prod_o a b) ⟶ F x = null}
    by simp
qed

definition pr_0 :: 'U ⇒ 'U ⇒ 'U
where pr_0 a b ≡ mkarr (prod_o a b) b (P_0 a b)

definition pr_1 :: 'U ⇒ 'U ⇒ 'U
where pr_1 a b ≡ 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
  using assms pr_0-def pr_1-def mkarr-in-hom ide-prod_o P_0-in-Hom P_1-in-Hom by auto

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
  using assms pr-in-hom by blast+

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
  using assms Fun-mkarr pr_0-def pr_1-def pr-simps(1,4) by presburger+
```

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

```
definition Tuple :: 'U ⇒ 'U ⇒ 'U ⇒ 'U
where Tuple f g ≡ (λx. if x ∈ Set (dom f)
                      then IN (Set (cod f) × Set (cod g)) (Fun f x, Fun g x)
                      else null)

definition tuple :: 'U ⇒ 'U ⇒ 'U
where tuple f g ≡ mkarr (dom f) (prod_o (cod f) (cod g)) (Tuple f g)

lemma tuple-in-hom [intro]:
assumes «f : c → a» and «g : c → b»
shows «tuple f g : c → prod_o a b»
proof -
  have Tuple f g ∈ Set c → Set (prod_o a b)
  proof
    fix x
    assume x: x ∈ Set c
    have bij-betw (IN (Set a × Set b)) (Set a × Set b) (Set (mkide (Set a × Set b)))
      using assms embeds-pairs ide-prod_o(2) prod_o-def
      by (metis ide-cod ide-prod_o(3) in-homE)
    thus Tuple f g x ∈ Set (prod_o a b)
      unfolding Tuple-def prod_o-def Fun-def
      using assms x bij-betw-apply in-homE small-Set
      by auto fastforce
  qed
  moreover have ⋀x. x ∉ Set c ⟹ Tuple f g x = null
    unfolding Tuple-def
    using assms by auto
  ultimately show ?thesis
    unfolding tuple-def
    using assms mkarr-in-hom ide-prod_o(1) by fastforce
qed

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)
  using assms
  by (metis assms in-homE in-homI tuple-in-hom)+
```

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) · tuple f g = f
and pr_0 (cod f) (cod g) · tuple f g = g
proof −
  let ?c = dom f and ?a = cod f and ?b = cod g
  show pr_1 ?a ?b · tuple f g = f
  proof −
    have pr_1 ?a ?b · tuple f g =
          mkarr (prod_o ?a ?b) ?a (P_1 ?a ?b) · mkarr ?c (prod_o ?a ?b) (Tuple f g)
      unfolding pr_1-def tuple-def Tuple-def
      using assms by auto
    also have ... = mkarr ?c ?a (P_1 ?a ?b ∘ Tuple f g)
      using assms comp-mkarr
      by (metis (lifting) calculation ide-cod pr-simps(4,5) seqE seqI tuple-simps(1,3))
    also have ... = mkarr ?c ?a
                      (λx. if x ∈ Set ?c
                           then fst (OUT (Set ?a × Set ?b)
                                         (IN (Set ?a × Set ?b) (Fun f x, Fun g x)))
                           else null)
    proof −
      have (P_1 ?a ?b ∘ Tuple f g) =
            (λx. if «x : 1^? → ?c»
                 then fst (OUT (Set ?a × Set ?b)
                               (IN (Set ?a × Set ?b) (Fun f x, Fun g x)))
                 else null)
        using assms ide-prod_o(3) [of ?a ?b] bij-betw-apply Tuple-def Fun-def by fastforce
      thus ?thesis by simp
    qed
    also have ... = mkarr ?c ?a (λx. if x ∈ Set ?c then fst (Fun f x, Fun g x) else null)
    proof −
      have ⋀x. x ∈ Set ?c ⟹
                 OUT (Set ?a × Set ?b) (IN (Set ?a × Set ?b) (Fun f x, Fun g x)) =
                 (Fun f x, Fun g x)
        using assms OUT-IN [of Set ?a × Set ?b] small-product-set embeds-product-sets
              Fun-def
        by auto
      thus ?thesis
        by (metis (lifting))
    qed
    also have ... = mkarr ?c ?a (λx. if x ∈ Set ?c then Fun f x else null)
      using assms by (metis (lifting) fst-eqD)
    also have ... = f
    proof −
      have Fun f = (λx. if x ∈ Set ?c then Fun f x else null)
      unfolding Fun-def by meson
    thus ?thesis
      by (metis (no-types, lifting) arr-iff-in-hom assms mkarr-Fun)
  qed
  finally show ?thesis by simp
qed
show pr_0 ?a ?b · tuple f g = g
proof –
  have pr_0 ?a ?b · tuple f g =
        mkarr (prod_o ?a ?b) ?b (P_0 ?a ?b) · mkarr ?c (prod_o ?a ?b) (Tuple f g)
    unfolding pr_0-def tuple-def Tuple-def
    using assms comp-mkarr by auto
  also have ... = mkarr ?c ?b (P_0 ?a ?b ∘ Tuple f g)
    using assms comp-mkarr
    by (metis (lifting) calculation ide-cod seqE seqI pr-simps(1,2) tuple-simps(1,3))
  also have ... = mkarr ?c ?b
                    (λx. if x ∈ Set ?c
                         then snd (OUT (Set ?a × Set ?b)
                                     (IN (Set ?a × Set ?b) (Fun f x, Fun g x)))
                         else null)
  proof –
    have (P_0 ?a ?b ∘ Tuple f g) =
          (λx. if x ∈ Set ?c
               then snd (OUT (Set ?a × Set ?b)
                          (IN (Set ?a × Set ?b) (Fun f x, Fun g x)))
               else null)
      using assms ide-prod_o(3) [of ?a ?b] bij-betw-apply Tuple-def Fun-def by fastforce
    thus ?thesis by simp
  qed
  also have ... = mkarr ?c ?b (λx. if x ∈ Set ?c then snd (Fun f x, Fun g x) else null)
  proof –
    have ⋀x. x ∈ Set ?c ⟹
                 OUT (Set ?a × Set ?b) (IN (Set ?a × Set ?b) (Fun f x, Fun g x)) =
                 (Fun f x, Fun g x)
      using assms OUT-IN [of Set ?a × Set ?b] small-product-set embeds-product-sets
            Fun-def
      by auto
    thus ?thesis
      by (metis (lifting))
  qed
  also have ... = mkarr ?c ?b (λx. if x ∈ Set ?c then Fun g x else null)
    using assms by (metis (lifting) snd-eqD)
  also have ... = g
  proof –
    have Fun g = (λx. if x ∈ Set ?c then Fun g x else null)
      unfolding Fun-def by (metis assms)
    thus ?thesis
      by (metis (no-types, lifting) arr-iff-in-hom assms mkarr-Fun)
  qed
```

```
    finally show ?thesis by simp
  qed
qed

lemma Fun-tuple:
assumes span f g
shows Fun (tuple f g) =
       (λx. if x ∈ Set (dom f)
            then IN (Set (cod f) × Set (cod g)) (Fun f x, Fun g x)
            else null)
  using tuple-def Tuple-def Fun-mkarr assms tuple-simps(1) by presburger

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
  show has-as-binary-product a b (pr_1 a b) (pr_0 a b)
  proof
    show 1: span (pr_1 a b) (pr_0 a b)
      using assms by auto
    show cod (pr_1 a b) = a
      using assms by auto
    show cod (pr_0 a b) = b
      using assms by auto
    fix x f g
    assume f: «f : x → a» and g: «g : x → b»
    let ?H = λz. if z ∈ Set x then IN (Set a × Set b) (Fun f z, Fun g z) else null
    let ?h = mkarr x (prod_o a b) ?H
    have h: «?h : x → dom (pr_1 a b)» ∧ C (pr_1 a b) ?h = f ∧ C (pr_0 a b) ?h = g
      using assms f g tuple-in-hom [of f x a g b] comp-pr-tuple [of f g]
      unfolding tuple-def Tuple-def by auto
    moreover have ⋀h'. «h' : x → dom (pr_1 a b)» ∧ C (pr_1 a b) h' = f ∧
                       C (pr_0 a b) h' = g
                         ⟹ h' = ?h
    proof –
      fix h'
      assume h': «h' : x → dom (pr_1 a b)» ∧ C (pr_1 a b) h' = f ∧ C (pr_0 a b) h' = g
      show h' = ?h
      proof (intro arr-eqI' [of h'])
        show «h' : x → dom (prod_o a b)»
          using assms h' ide-prod_o(1) by auto
        show «?h : x → dom (prod_o a b)»
          using assms h ide-prod_o(1) by auto
        show ⋀z. «z : 1^? → x» ⟹ h' · z = ?h · z
        proof –
          fix z
          assume z: «z : 1^? → x»
          have h' · z = Fun h' z
            using h' z Fun-def by auto
            also have ... = IN (Set a × Set b) (Fun f z, Fun g z)
            proof -
              have fst (OUT (Set a × Set b) (Fun h' z)) = Fun f z
              proof -
                have Fun f z = Fun (pr1 a b · h') z
                  using h' by force
                also have ... = (P1 a b ∘ Fun h') z
                  using assms(1−2) f h' Fun-pr(1) Fun-comp arrI by auto
                also have ... = fst (OUT (Set a × Set b) (Fun h' z))
                  using assms(1,2) h' z Fun-def by auto
                finally show ?thesis by simp
              qed
              moreover have snd (OUT (Set a × Set b) (Fun h' z)) = Fun g z
              proof -
                have Fun g z = Fun (pr0 a b · h') z
                  using h' by force
                also have ... = (P0 a b ∘ Fun h') z
                  using assms(1−2) g h' Fun-pr(2) Fun-comp arrI by auto
                also have ... = snd (OUT (Set a × Set b) (Fun h' z))
                  using assms(1,2) h' z Fun-def by auto
               finally show ?thesis by simp
              qed
              ultimately have IN (Set a × Set b) (Fun f z, Fun g z) =
                              IN (Set a × Set b) (OUT (Set a × Set b) (Fun h' z))
                by (metis split-pairs2)
              also have ... = Fun h' z
                using assms h' z IN-OUT ‹C h' z = Fun h' z› prod_o-def Fun-def
                  small-product-set [of a b] embeds-product-sets [of a b]
                by auto
              finally show ?thesis by simp
            qed
            also have ... = C ?h z
              using app-mkarr assms(1,2) h z by auto
            finally show C h' z = C ?h z by blast
          qed
        qed
      qed
      ultimately show ∃!h. «h : x → dom (pr1 a b)» ∧ C (pr1 a b) h = f ∧
                          C (pr0 a b) h = g
        by auto
    qed
  qed

  lemma has-binary-products:
  shows has-binary-products
    using binary-product-pr
    by (meson binary-product.has-as-binary-product has-binary-products-def)

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 ..

  abbreviation pr0 :: 'U ⇒ 'U ⇒ 'U
  where pr0 ≡ Products.pr0

  abbreviation pr1 :: 'U ⇒ 'U ⇒ 'U
  where pr1 ≡ Products.pr1

  sublocale elementary-category-with-binary-products C pr0 pr1
  proof
    show ⋀f g. span f g ⟹ ∃!l. C (pr1 (cod f) (cod g)) l = f ∧ C (pr0 (cod f) (cod g)) l = g
    proof -
      fix f g
      assume fg: span f g
      interpret binary-product C ‹cod f› ‹cod g› ‹pr1 (cod f) (cod g)› ‹pr0 (cod f) (cod g)›
        using fg Products.binary-product-pr ide-cod by blast
      show ∃!l. C (pr1 (cod f) (cod g)) l = f ∧ C (pr0 (cod f) (cod g)) l = g
        by (metis (full-types) fg tuple-props(4,5,6))
    qed
  qed auto

  lemma bin-prod-comparison-map-props:
  assumes ide a and ide b
  shows OUT (Set a × Set b) ∈ Set (prod a b) → Set a × Set b
  and IN (Set a × Set b) ∈ Set a × Set b → Set (prod a b)
  and ⋀x. x ∈ Set (prod a b) ⟹ IN (Set a × Set b) (OUT (Set a × Set b) x) = x
  and ⋀y. y ∈ Set a × Set b ⟹ OUT (Set a × Set b) (IN (Set a × Set b) y) = y
  and bij-betw (OUT (Set a × Set b)) (Set (prod a b)) (Set a × Set b)
  and bij-betw (IN (Set a × Set b)) (Set a × Set b) (Set (prod a b))
    using assms Products.ide-prod_o [of a b] pr-simps(5) by auto

  lemma Fun-pr0:
  assumes ide a and ide b
  shows Fun (pr0 a b) = Products.P0 a b
    using assms Products.Fun-pr(2) by auto[1]

  lemma Fun-pr1:
  assumes ide a and ide b
  shows Fun (pr1 a b) = Products.P1 a b
    using assms Products.Fun-pr(1) by auto[1]
```

```
lemma Fun-prod:
assumes «f : a → b» and «g : c → d»
shows Fun (prod f g) = (λx. if x ∈ 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
  fix x
  show Fun (prod f g) x = (if x ∈ 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 (cases x ∈ Set (prod a c))
    case False
    show ?thesis
      using False
      by (metis assms(1,2) in-homE prod-simps(2) Fun-def)
    next
    case True
    show ?thesis
    proof –
      have «x : 1^? → dom (prod f g)»
        using True assms(1,2) by fastforce
      moreover have «pr_1 a c · x : 1^? → dom f» ∧ «pr_0 a c · x : 1^? → dom g»
        using assms True
        by (intro conjI comp-in-homI) fastforce+
      moreover have prod f g · x = tuple (f · pr_1 a c · x) (g · pr_0 a c · x)
        using assms True prod-tuple tuple-pr-arr
        by (metis calculation(2) ide-dom in-homE seqI)
      ultimately show ?thesis
        using assms True Fun-def by auto
    qed
  qed
qed

lemma prod-ide-eq:
assumes ide a and ide b
shows prod a b = mkide (Set a × Set b)
  using assms(1,2) pr-simps(2) Products.prod_o-def by force

lemma tuple-eq:
assumes «f : x → a» and «g : x → b»
shows tuple f g = mkarr x (prod a b)
                    (λz. if z ∈ Set x
                         then IN (Set a × Set b) (Fun f z, Fun g z)
                         else null)
proof –
  have tuple f g = Products.tuple f g
    by (metis Products.comp-pr-tuple(1,2) assms(1,2) in-homE pr-tuple(1,2) universal)
  thus ?thesis
```

```
    unfolding Products.tuple-def Products.Tuple-def
    using assms Products.prod_o-def prod-ide-eq by fastforce
qed

lemma tuple-point-eq:
assumes «x : 1^? → a» and «y : 1^? → b»
shows tuple x y = IN (Set a × Set b) (x, y)
proof -
  have 1: tuple x y = mkarr 1^? (prod a b)
                         (λz. if z ∈ Set 1^? then IN (Set a × Set b) (x, y) else null)
  proof -
    have ⋀z. z ∈ Set 1^? ⟹ Fun x z = x ∧ Fun y z = y
      unfolding Fun-def
      by (metis assms CollectD comp-arr-dom ide-dom ide-in-hom in-homE some-trm-eqI)
    hence (λz. if z ∈ Set 1^? then IN (Set a × Set b) (Fun x z, Fun y z) else null) =
          (λz. if z ∈ Set 1^? then IN (Set a × Set b) (x, y) else null)
      by fastforce
    thus ?thesis
      using assms tuple-eq by simp
  qed
  also have ... = IN (Set a × Set b) (x, y)
  proof -
    have mkarr 1^? (prod a b)
            (λz. if z ∈ Set 1^? then IN (Set a × Set b) (x, y) else null) =
          mkarr 1^? (prod a b)
            (λz. if z ∈ Set 1^? then IN (Set a × Set b) (x, y) else null) · 1^?
      by (metis (lifting) assms(1,2) calculation comp-arr-dom dom-mkarr in-homE
          tuple-simps(1))
    also have ... = IN (Set a × Set b) (x, y)
      using app-mkarr [of 1^? prod a b - 1^?]
      by (metis (full-types, lifting) CollectI
          assms(1,2) 1 ide-in-hom ide-some-terminal tuple-in-hom)
    finally show ?thesis by blast
  qed
  finally show ?thesis by blast
qed

lemma Fun-tuple:
assumes span f g
shows Fun (tuple f g) =
       (λx. if x ∈ Set (dom f)
            then IN (Set (cod f) × Set (cod g)) (Fun f x, Fun g x)
            else null)
  using assms Fun-mkarr tuple-eq [of f dom f cod f g cod g]
  by (metis (lifting) in-homI tuple-simps(1))

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)
    using assms small-product
    by (metis Set-two ide-two(1) small-Set small-insert-iff small-union)

  lemma embeds-Coprod:
  assumes ide a and ide b
  shows embeds (Coprod a b)
  proof −
    have Coprod a b ⊆ Collect arr × Collect arr
      using ff-simps(1) tt-simps(1) by blast
    thus ?thesis
      using embeds-pairs
      by (simp add: embeds-subset)
  qed

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

  lemma ide-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 −
  show ide (coprod_o a b)
  and 1: bij-betw (OUT (Coprod a b)) (Set (coprod_o a b)) (Coprod a b)
    unfolding coprod_o-def
    using assms ide-mkide(1) bij-OUT small-Coprod embeds-Coprod by metis+
  show 2: bij-betw (IN (Coprod a b)) (Coprod a b) (Set (coprod_o a b))
    using 1 bij-betw-inv-into coprod_o-def by auto
  show ⋀x. x ∈ Set (coprod_o a b) ⟹ OUT (Coprod a b) x ∈ Coprod a b
    using 1 bij-betwE by blast
  show ⋀y. y ∈ Coprod a b ⟹ IN (Coprod a b) y ∈ Set (coprod_o a b)
    using 2 bij-betwE by blast
  show ⋀x. x ∈ Set (coprod_o a b) ⟹ IN (Coprod a b) (OUT (Coprod a b) x) = x
    using assms small-Coprod embeds-Coprod IN-OUT coprod_o-def by metis
  show ⋀y. y ∈ Coprod a b ⟹ OUT (Coprod a b) (IN (Coprod a b) y) = y
    using assms small-Coprod embeds-Coprod coprod_o-def 1
          bij-betw-inv-into-right
            [of OUT (Coprod a b) Set (coprod_o a b) Coprod a b]
    by presburger
qed

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

lemma In_0-in-Hom:
assumes ide a and ide b
shows In_0 a b ∈ Hom b (coprod_o a b)
proof
  show In_0 a b ∈ {F. ∀x. x ∉ Set b ⟶ F x = null} by simp
  show In_0 a b ∈ Set b → Set (coprod_o a b)
  proof
    fix x
    assume x: x ∈ Set b
    have (ff, x) ∈ Coprod a b
      using assms x by blast
    thus In_0 a b x ∈ Set (coprod_o a b)
      using assms x ide-coprod_o(3) bij-betwE ide-coprod_o(5) by presburger
  qed
qed

lemma In_1-in-Hom:
assumes ide a and ide b
shows In_1 a b ∈ Hom a (coprod_o a b)
proof
  show In_1 a b ∈ {F. ∀x. x ∉ Set a ⟶ F x = null} by simp
  show In_1 a b ∈ Set a → Set (coprod_o a b)
  proof
    fix x
    assume x: x ∈ Set a
    have (tt, x) ∈ Coprod a b
      using assms x by blast
    thus In_1 a b x ∈ Set (coprod_o a b)
      using assms x ide-coprod_o(3) bij-betwE ide-coprod_o(5) by presburger
  qed
qed

definition in_0 :: 'U ⇒ 'U ⇒ 'U
where in_0 a b ≡ mkarr b (coprod_o a b) (In_0 a b)

definition in_1 :: 'U ⇒ 'U ⇒ 'U
where in_1 a b ≡ mkarr a (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 (coprod_o a b)
and in-hom (in_0 a b) b (coprod_o a b)
  using assms in_0-def in_1-def mkarr-in-hom ide-coprod_o In_0-in-Hom In_1-in-Hom by auto

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) = coprod_o a b
and arr (in_1 a b) and dom (in_1 a b) = a and cod (in_1 a b) = coprod_o a b
  using assms in-in-hom by blast+

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
  using assms Fun-mkarr in_0-def in_1-def in-simps(1,4) by presburger+

definition Cotuple :: 'U ⇒ 'U ⇒ 'U ⇒ 'U
where Cotuple f g ≡ (λx. 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)

definition cotuple :: 'U ⇒ 'U ⇒ 'U
where cotuple f g ≡ mkarr (coprod_o (dom f) (dom g)) (cod f) (Cotuple f g)

lemma cotuple-in-hom [intro, simp]:
assumes «f : a → c» and «g : b → c»
shows «cotuple f g : coprod_o a b → c»
```

```
proof −
  have bij: bij-betw (OUT (Coprod a b)) (Set (coprod_o a b)) (Coprod a b)
    using assms ide-coprod_o(2) ide-dom by blast
  have Cotuple f g ∈ Set (coprod_o a b) → Set c
  proof
    fix x
    assume x: x ∈ Set (coprod_o a b)
    have 1: OUT (Coprod a b) x ∈ Coprod a b
      using x bij bij-betwE by blast
    have fst (OUT (Coprod a b) x) = tt ∨ fst (OUT (Coprod a b) x) = ff
      using 1 by fastforce
    moreover have fst (OUT (Coprod a b) x) = tt ⟹ Cotuple f g x ∈ Set c
    proof −
      assume 2: fst (OUT (Coprod a b) x) = tt
      have snd (OUT (Coprod a b) x) ∈ Set a
        using 1 2 tt-ne-ff by auto
      thus ?thesis
        unfolding Cotuple-def
        using assms x 2 Fun-in-Hom [of f a c] tt-ne-ff
        by auto fastforce
    qed
    moreover have fst (OUT (Coprod a b) x) = ff ⟹ Cotuple f g x ∈ Set c
    proof −
      assume 2: fst (OUT (Coprod a b) x) = ff
      have snd (OUT (Coprod a b) x) ∈ Set b
        using 1 2 tt-ne-ff by auto
      thus ?thesis
        unfolding Cotuple-def
        using assms x 2 Fun-in-Hom [of g b c] tt-ne-ff by auto
    qed
    ultimately show Cotuple f g x ∈ Set c by blast
  qed
  moreover have ⋀x. x ∉ Set (coprod_o a b) ⟹ Cotuple f g x = null
    unfolding Cotuple-def
    using assms by auto
  ultimately show ?thesis
    unfolding cotuple-def
    using assms mkarr-in-hom ide-coprod_o(1) by fastforce
qed

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
  using assms
  by (metis assms in-homE in-homI cotuple-in-hom)+

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 -
  let ?a = dom f and ?b = dom g and ?c = cod f
  show cotuple f g · in_1 (dom f) (dom g) = f
  proof -
    have cotuple f g · in_1 (dom f) (dom g) =
          mkarr (coprod_o ?a ?b) ?c (Cotuple f g) · mkarr ?a (coprod_o ?a ?b) (In_1 ?a ?b)
      unfolding in_1-def cotuple-def
      using assms by auto
    also have ... = mkarr ?a ?c (Cotuple f g ∘ In_1 ?a ?b)
      using assms comp-mkarr cotuple-def cotuple-simps(1) ide-dom in_1-def in-simps(4)
      by presburger
    also have ... = mkarr ?a ?c
                    (λx. if x ∈ Set ?a
                         then Fun f (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (tt, x))))
                         else null)
    proof -
      have ⋀x. x ∈ Set ?a ⟹
                (Cotuple f g ∘ In_1 ?a ?b) x =
                Fun f (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (tt, x))))
        unfolding Cotuple-def tt-ne-ff
        using assms tt-ne-ff ide-coprod_o by auto
      hence Cotuple f g ∘ In_1 ?a ?b =
            (λx. if x ∈ Set ?a
                 then Fun f (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (tt, x))))
                 else null)
        unfolding Cotuple-def
        by fastforce
      thus ?thesis by simp
    qed
    also have ... = mkarr ?a ?c (λx. if x ∈ Set ?a then Fun f x else null)
    proof -
      have ⋀x. x ∈ Set ?a ⟹
                   Fun f (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (tt, x)))) = Fun f x
        using assms ide-coprod_o(7) by auto
      thus ?thesis
        by meson
    qed
    also have ... = f
    proof -
      have Fun f = (λx. if x ∈ Set ?a then Fun f x else null)
        unfolding Fun-def by meson
      thus ?thesis
        by (metis (no-types, lifting) arr-iff-in-hom assms mkarr-Fun)
    qed
    finally show ?thesis by blast
  qed
  show cotuple f g · in_0 (dom f) (dom g) = g
  proof −
    have cotuple f g · in_0 (dom f) (dom g) =
          mkarr (coprod_o ?a ?b) ?c (Cotuple f g) · mkarr ?b (coprod_o ?a ?b) (In_0 ?a ?b)
      unfolding in_0-def cotuple-def
      using assms by auto
    also have ... = mkarr ?b ?c (Cotuple f g ∘ In_0 ?a ?b)
      using assms comp-mkarr cotuple-def cotuple-simps(1) ide-dom in_0-def in-simps(1)
      by presburger
    also have ... = mkarr ?b ?c
                      (λx. if x ∈ Set ?b
                           then Fun g (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (ff, x))))
                           else null)
    proof −
      have ⋀x. x ∈ Set ?b ⟹
                (Cotuple f g ∘ In_0 ?a ?b) x =
                Fun g (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (ff, x))))
        unfolding Cotuple-def tt-ne-ff
        using assms tt-ne-ff ide-coprod_o by auto
      hence Cotuple f g ∘ In_0 ?a ?b =
            (λx. if x ∈ Set ?b
                 then Fun g (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (ff, x))))
                 else null)
        unfolding Cotuple-def
        by fastforce
      thus ?thesis by simp
    qed
    also have ... = mkarr ?b ?c (λx. if x ∈ Set ?b then Fun g x else null)
    proof −
      have ⋀x. x ∈ Set ?b ⟹
                   Fun g (snd (OUT (Coprod ?a ?b) (IN (Coprod ?a ?b) (ff, x)))) = Fun g x
        using assms ide-coprod_o(7) by auto
      thus ?thesis
        by meson
    qed
    also have ... = g
    proof −
      have Fun g = (λx. if x ∈ Set ?b then Fun g x else null)
        unfolding Fun-def by meson
      thus ?thesis
        by (metis (no-types, lifting) arr-iff-in-hom assms mkarr-Fun)
    qed
    finally show ?thesis by blast
  qed
qed

lemma Fun-cotuple:
assumes cospan f g
shows Fun (cotuple f g) =
      (λx. 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)
  using cotuple-def Cotuple-def Fun-mkarr assms cotuple-simps(1) by presburger

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 −
  have bij: bij-betw (OUT (Coprod a b)) (Set (coprod_o a b)) (Coprod a b)
    using assms ide-coprod_o(2) ide-dom by blast
  interpret Cop: dual-category C ..
  show ?thesis
  proof
    show Cop.has-as-binary-product a b (in_1 a b) (in_0 a b)
    proof
      show Cop.span (in_1 a b) (in_0 a b)
        using assms(1,2) by force
      show Cop.cod (in_1 a b) = a
        using assms(1,2) by fastforce
      show Cop.cod (in_0 a b) = b
        using assms(1,2) by fastforce
      fix c f g
      assume f: Cop.in-hom f c a and g: Cop.in-hom g c b
      show ∃!h. Cop.in-hom h c (Cop.dom (in_1 a b)) ∧ in_1 a b ·^op h = f ∧ in_0 a b ·^op h = g
      proof
        show Cop.in-hom (cotuple f g) c (Cop.dom (in_1 a b)) ∧
                in_1 a b ·^op (cotuple f g) = f ∧ in_0 a b ·^op (cotuple f g) = g
        proof (intro conjI)
          show Cop.in-hom (cotuple f g) c (Cop.dom (in_1 a b))
            using assms(1,2) f g by force
          show in_1 a b ·^op cotuple f g = f
            using assms(1,2) f g comp-cotuple-in by auto
          show in_0 a b ·^op cotuple f g = g
            using assms(1,2) f g comp-cotuple-in
            by (metis Cop.comp-def Cop.hom-char in-homE)
        qed
        show ⋀h. Cop.in-hom h c (Cop.dom (in_1 a b)) ∧ in_1 a b ·^op h = f ∧ in_0 a b ·^op h = g
                   ⟹ h = cotuple f g
        proof −
          fix h
          assume h: Cop.in-hom h c (Cop.dom (in_1 a b)) ∧
                      in_1 a b ·^op h = f ∧ in_0 a b ·^op h = g
          show h = cotuple f g
          proof (intro arr-eqI [of h])
```

```
show par: par h (cotuple f g)
  using assms(1,2) h by force
show Fun h = Fun (cotuple f g)
proof
  fix x
  show Fun h x = Fun (cotuple f g) x
  proof (cases x ∈ Set (coprod_o a b))
    case False
    show ?thesis
      using False assms(1,2) h par Fun-cotuple [of f g] Fun-def
      by (metis (lifting) Cop.cod-char Cop.dom-char Cop.in-homE
          in-simps(6) mem-Collect-eq)
    next
    case True
    show ?thesis
    proof -
      have 2: OUT (Coprod a b) x ∈ Coprod a b
        using True bij bij-betwE by blast
      hence fst (OUT (Coprod a b) x) = tt ∨ fst (OUT (Coprod a b) x) = ff
        using True bij bij-betwE
        unfolding coprod_o-def
        by auto
      moreover have fst (OUT (Coprod a b) x) = tt ⟹ ?thesis
      proof -
        assume 3: fst (OUT (Coprod a b) x) = tt
        have 4: snd (OUT (Coprod a b) x) ∈ Set a
          using True 2 3 tt-ne-ff by fastforce
        have Fun (cotuple f g) x = Fun f (snd (OUT (Coprod a b) x))
          using assms 2 3 4 coprod_o-def
          apply simp
         by (metis (lifting) HOL.ext Cop.cod-char Cop.dom-char Cop.in-homE True
              Fun-cotuple [of f g] arr-dom-iff-arr f g ide-char)
        also have ... = Fun (h · in_1 a b) (snd (OUT (Coprod a b) x))
          using h by auto
        also have ... = Fun h (Fun (in_1 a b) (snd (OUT (Coprod a b) x)))
          using Cop.arrI Fun-comp f h by force
        also have ... = Fun h (IN (Coprod a b) (tt, snd (OUT (Coprod a b) x)))
          using assms 4 Fun-in(1) [of a b] by auto
        also have ... = Fun h (IN (Coprod a b) (OUT (Coprod a b) x))
          by (metis 3 surjective-pairing)
        also have ... = Fun h x
          using assms True ide-coprod_o(6) by presburger
        finally show ?thesis by simp
      qed
      moreover have fst (OUT (Coprod a b) x) = ff ⟹ ?thesis
      proof -
        assume 3: fst (OUT (Coprod a b) x) = ff
        have 4: snd (OUT (Coprod a b) x) ∈ Set b
          using True 2 3 tt-ne-ff by fastforce
```

```
                      have Fun (cotuple f g) x = Fun g (snd (OUT (Coprod a b) x))
                        using True assms f g 2 3 4 tt-ne-ff coprod_o-def Fun-cotuple [of f g]
                        apply auto[1]
                        by (metis (lifting) HOL.ext fst-conv in-homE snd-conv)
                      also have ... = Fun (h · in_0 a b) (snd (OUT (Coprod a b) x))
                        using h by auto
                      also have ... = Fun h (Fun (in_0 a b) (snd (OUT (Coprod a b) x)))
                        using Cop.arrI Fun-comp g h by force
                      also have ... = Fun h (IN (Coprod a b) (ff, snd (OUT (Coprod a b) x)))
                        using assms 4 Fun-in(2) [of a b] by auto
                      also have ... = Fun h (IN (Coprod a b) (OUT (Coprod a b) x))
                        by (metis 3 surjective-pairing)
                      also have ... = Fun h x
                        using assms True ide-coprod_o(6) by presburger
                      finally show ?thesis by simp
                    qed
                    ultimately show ?thesis by blast
                  qed
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed

  lemma has-binary-coproducts:
  shows category.has-binary-products (dual-category.comp C)
  proof −
    interpret Cop: dual-category C ..
    show Cop.has-binary-products
    proof (unfold Cop.has-binary-products-def, intro allI impI, elim conjE)
      fix a b
      assume a: Cop.ide a and b: Cop.ide b
      interpret binary-product Cop.comp a b ‹in_1 a b› ‹in_0 a b›
        using a b binary-coproduct-in [of a b] Cop.ide-char by blast
      show ∃ p. Ex (Cop.has-as-binary-product a b p)
        using has-as-binary-product by blast
    qed
  qed

end
```

4.7.1 Exported Notions

```
context sets-cat-with-cotupling
begin
```

interpretation *Coproducts*: *coproducts-in-sets-cat* **..**

abbreviation in_0 :: $'U \Rightarrow 'U \Rightarrow 'U$
where $in_0 \equiv Coproducts.in_0$

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* $(coprod_o\ a\ b)$
  using *assms Coproducts.ide-coprod*$_o$ **by** *blast*

lemma in_1*-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *in-hom* $(in_1\ a\ b)\ a\ (coprod_o\ a\ b)$
  using *assms Coproducts.in-in-hom* **by** *blast*

lemma in_0*-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *in-hom* $(in_0\ a\ b)\ b\ (coprod_o\ a\ b)$
  using *assms Coproducts.in-in-hom* **by** *blast*

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* $(in_1\ a\ b) = coprod_o\ a\ b$
  using *assms Coproducts.in-simps* **by** *auto*

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* $(in_0\ a\ b) = coprod_o\ a\ b$
  using *assms Coproducts.in-simps* **by** *auto*

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 $\bigwedge y.\ y \in Coprod\ a\ b \Longrightarrow OUT\ (Coprod\ a\ b)\ (IN\ (Coprod\ a\ b)\ y) = y$
  using *assms Coproducts.ide-coprod*$_o$ **by** *auto*

```
lemma Fun-in_1:
assumes ide a and ide b
shows Fun (in_1 a b) = Coproducts.In_1 a b
  using assms Coproducts.Fun-in(1) by auto[1]

lemma Fun-in_0:
assumes ide a and ide b
shows Fun (in_0 a b) = Coproducts.In_0 a b
  using assms Coproducts.Fun-in(2) by auto[1]

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»
  using assms Coproducts.cotuple-in-hom by blast

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
  using assms Coproducts.cotuple-simps by auto

abbreviation Cotuple
where Cotuple f g ≡ (λx. 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)
  unfolding Coproducts.cotuple-def Coproducts.Cotuple-def
  using assms by auto

lemma Fun-cotuple:
assumes cospan f g
shows Fun (cotuple f g) = Cotuple f g
  using assms Coproducts.Fun-cotuple by blast

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)
  using assms Coproducts.binary-coproduct-in by blast
  lemma has-binary-coproducts:
  shows category.has-binary-products (dual-category.comp C)
    using Coproducts.has-binary-coproducts by blast

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
    using embeds-bool
    by unfold-locales auto
  sublocale sets-cat-with-pairing sml C ..
  sublocale sets-cat-with-cotupling ..

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 = {λx. null}
    by auto

  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 −
  have 1: small ((⋃i∈I. Set (A i)) ∪ {null})
    using assms small-Set by auto
  have 2: ⋀F v. ⟦F ∈ ProdX I A; popular-value F v⟧ ⟹ v = null
  proof −
    fix F v
    assume F: F ∈ ProdX I A
    assume v: popular-value F v
    have (∃ i. i ∈ I ∧ v ∈ Set (A i)) ∨ v = null
      using v F popular-value-in-range [of F v] by blast
    hence v ≠ null ⟹ {i. F i = v} ⊆ I
      using F by blast
    hence v ≠ null ⟹ ¬ popular-value F v
      using assms(1) smaller-than-small by blast
    thus v = null
      using v by blast
  qed
  show 3: range F ⊆ (⋃i∈I. Set (A i)) ∪ {null}
    using assms(4) by auto
  show small-function F
  proof
    show small (range F)
      using 1 3 smaller-than-small by blast
    show at-most-one-popular-value F
      using assms(4) 2 Uniq-def
      by (metis (mono-tags, lifting))
  qed
qed

lemma small-ProdX:
assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
shows small (ProdX I A)
proof (cases small (UNIV :: 'U set))
  case True
  show ?thesis
    using True small-function-tuple smaller-than-small
    by (metis large-univ subset-UNIV)
  next
  case False
  have ⋀F. F ∈ ProdX I A ⟹ SF-Dom F ⊆ I
  proof −
    fix F
    assume F: F ∈ ProdX I A
    have popular-value F null
    proof −
      have ¬ small (UNIV − I)
        using assms False small-union by fastforce
      moreover have UNIV − I ⊆ {i. F i = null}
        using F by blast
      ultimately show ?thesis
        using smaller-than-small by blast
    qed
    thus SF-Dom F ⊆ I
      using F by auto
  qed
  hence ProdX I A ⊆ {f. small-function f ∧ SF-Dom f ⊆ I ∧
                         range f ⊆ (⋃i∈I. Set (A i)) ∪ {null}}
    using assms small-function-tuple by blast
  moreover have 1: small ((⋃i∈I. Set (A i)) ∪ {null})
    using assms small-Set by auto
  ultimately show ?thesis
    using assms(1) small-Set small-funcset [of I (⋃i∈I. Set (A i)) ∪ {null}]
          smaller-than-small
    by blast
qed

lemma embeds-ProdX:
assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
shows embeds (ProdX I A)
proof −
  obtain ι where ι: is-embedding-of ι SEF
    using embeds-SEF by blast
  have ProdX I A ⊆ SEF
    using assms EF-def small-function-tuple by auto
  hence is-embedding-of ι (ProdX I A)
    using ι by (meson dual-order.trans image-mono inj-on-subset)
  thus ?thesis by blast
qed

lemma ide-prodX:
assumes small I and A ∈ I → Collect ide and I ⊆ 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 ⋀x. x ∈ Set (prodX I A) ⟹ OUT (ProdX I A) x ∈ ProdX I A
and ⋀y. y ∈ ProdX I A ⟹ IN (ProdX I A) y ∈ 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
proof −
  have 2: small ((⋃i∈I. Set (A i)) ∪ {null})
    using assms(1−2) small-Set by auto
  have ∗: ⋀F. F ∈ ProdX I A ⟹ small-function F ∧ range F ⊆ (⋃i∈I. Set (A i)) ∪ {null}
    using assms small-function-tuple by blast
  show ide (prodX I A)
    unfolding prodX-def
    using assms small-ProdX embeds-ProdX by auto
  show 1: bij-betw (OUT (ProdX I A)) (Set (prodX I A)) (ProdX I A)
    unfolding prodX-def
    using assms small-ProdX embeds-ProdX bij-OUT [of ProdX I A] by fastforce
  show 2: bij-betw (IN (ProdX I A)) (ProdX I A) (Set (prodX I A))
    unfolding prodX-def
    using assms small-ProdX embeds-ProdX bij-IN [of ProdX I A] by fastforce
  show ⋀x. x ∈ Set (prodX I A) ⟹ OUT (ProdX I A) x ∈ ProdX I A
    using 1 bij-betwE by blast
  show ⋀y. y ∈ ProdX I A ⟹ IN (ProdX I A) y ∈ Set (prodX I A)
    using 2 bij-betwE by blast
  show ⋀x. x ∈ Set (prodX I A) ⟹ IN (ProdX I A) (OUT (ProdX I A) x) = x
  proof −
    fix x
    assume x: x ∈ Set (prodX I A)
    show IN (ProdX I A) (OUT (ProdX I A) x) = x
    proof −
      have x = inv-into (Set (prodX I A)) (OUT (ProdX I A)) (OUT (ProdX I A) x)
        using x 1
              bij-betw-inv-into-left
                [of OUT (ProdX I A) Set (prodX I A) ProdX I A]
        by auto
      thus ?thesis
        by (simp add: prodX-def)
    qed
  qed
  show ⋀y. y ∈ ProdX I A ⟹ OUT (ProdX I A) (IN (ProdX I A) y) = y
  proof −
    fix y
    assume y: y ∈ ProdX I A
    show OUT (ProdX I A) (IN (ProdX I A) y) = y
      using assms(1,2,3) y OUT-IN [of ProdX I A y] small-ProdX embeds-ProdX [of I A]
      by blast
  qed
qed

lemma terminal-prodX-empty:
shows terminal (prodX {} (A :: 'U ⇒ 'U))
proof −
  let ?I = {} :: 'U set
  have 1: {F. ∀ i. i ∉ ?I ⟶ F i = null} = {λi. null}
     by auto
  have ∃!x. x ∈ Set (prodX ?I A)
  proof −
    have eqpoll (Set (prodX ?I A)) {F. ∀ i. i ∉ ?I ⟶ F i = null}
    proof −
      have small {F. ∀ i. i ∉ ?I ⟶ F i = null}
        using 1 small-finite by force
      moreover have ∃ ι. is-embedding-of ι {F. ∀ i :: 'U. F i = null}
      proof −
```

```
        have is-embedding-of (λ-. 1^?) {λi. null}
          using ide-char ide-some-terminal by blast
        thus ?thesis
          using 1 by auto
      qed
      ultimately show ?thesis
        unfolding prodX-def
        using 1 bij-OUT [of {F. ∀ i. i ∉ ?I ⟶ F i = null}] eqpoll-def
        by auto blast
    qed
    moreover have ∃!x. x ∈ {F. ∀ i. i ∉ ?I ⟶ F i = null}
      using 1 by auto
    ultimately show ?thesis
      by (metis (no-types, lifting) eqpoll-iff-bijections)
  qed
  thus ?thesis
    using terminal-char ide-prodX(1)
    by (metis Pi-I empty-subsetI ex-in-conv small-Set smaller-than-small
      terminal-some-terminal)
qed

abbreviation PrX :: 'a set ⇒ ('a ⇒ 'U) ⇒ 'a ⇒ 'U ⇒ 'U
where PrX I A i ≡ λx. if x ∈ Set (prodX I A) then OUT (ProdX I A) x i else null

definition prX :: 'a set ⇒ ('a ⇒ 'U) ⇒ 'a ⇒ 'U
where prX I A i ≡ mkarr (prodX I A) (A i) (PrX I A i)

lemma prX-in-hom [intro, simp]:
assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
and i ∈ I
shows in-hom (prX I A i) (prodX I A) (A i)
proof (unfold prX-def, intro mkarr-in-hom)
  show ide (prodX I A)
    using assms ide-prodX by blast
  show ide (A i)
    using assms by blast
  show PrX I A i ∈ Hom (prodX I A) (A i)
  proof
    show PrX I A i ∈ Set (prodX I A) → Set (A i)
    proof
      fix x
      assume x: x ∈ Set (prodX I A)
      have OUT (ProdX I A) x ∈ ProdX I A
        using assms(1,2,3) x ide-prodX(2)
          bij-betwE [of OUT (ProdX I A) Set (prodX I A) ProdX I A]
        by blast
      thus PrX I A i x ∈ Set (A i)
        using assms x by force
    qed
    show PrX I A i ∈ {F. ∀x. x ∉ Set (prodX I A) ⟶ F x = null}
      by simp
  qed
qed

lemma prX-simps [simp]:
assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
and i ∈ I
shows arr (prX I A i) and dom (prX I A i) = prodX I A and cod (prX I A i) = A i
  using assms prX-in-hom by blast+

lemma Fun-prX:
assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
and i ∈ I
shows Fun (prX I A i) = PrX I A i
proof –
  have arr (prX I A i)
    using assms by auto
  thus ?thesis
    using assms Fun-mkarr [of prodX I A A i PrX I A i] prX-def by metis
qed

definition TupleX :: 'a set ⇒ 'U ⇒ ('a ⇒ 'U) ⇒ ('a ⇒ 'U) ⇒ 'U ⇒ 'U
where TupleX I c A F ≡ (λx. if x ∈ Set c then IN (ProdX I A) (λi. Fun (F i) x) else null)

lemma TupleX-in-Hom:
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
shows TupleX I c A F ∈ Hom c (prodX I A)
proof
  show TupleX I c A F ∈ {F. ∀x. x ∉ Set c ⟶ F x = null}
    unfolding TupleX-def
    using assms by auto
  show TupleX I c A F ∈ Set c → Set (prodX I A)
  proof (cases I = {})
    case False
    show ?thesis
    proof
      fix x
      assume x: x ∈ Set c
      have ∀i. i ∈ I ⟶ x ∈ Set (dom (F i))
        using False assms x by blast
      moreover have (λi. Fun (F i) x) ∈ ProdX I A
        using False assms x Fun-def by auto
      ultimately show TupleX I c A F x ∈ Set (prodX I A)
        unfolding TupleX-def
        using False assms x ide-prodX(3) [of I A] bij-betw-apply
        by (metis (mono-tags, lifting))
    qed
    next
    case True
    show ?thesis
      unfolding TupleX-def
      using True assms ide-prodX(3) bij-betw-apply Fun-def
      by auto[1] fastforce
  qed
qed

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 ∈ 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 «tupleX I c A F : c → prodX I A»
  unfolding tupleX-def
  using assms ide-prodX TupleX-in-Hom
  by (intro mkarr-in-hom) auto

lemma tupleX-simps [simp]:
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 arr (tupleX I c A F)
and dom (tupleX I c A F) = c
and cod (tupleX I c A F) = prodX I A
  using assms in-homE tupleX-in-hom by metis+

lemma comp-prX-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
shows i ∈ I ⟹ C (prX I A i) (tupleX I c A F) = F i
proof −
  assume i: i ∈ I
  have I: I ≠ {}
    using i by blast
  hence c: ide c
    using assms(4) ide-dom by blast
  show C (prX I A i) (tupleX I c A F) = F i
  proof −
    have C (prX I A i) (tupleX I c A F) =
          mkarr (prodX I A) (A i) (PrX I A i) · mkarr c (prodX I A) (TupleX I c A F)
      unfolding prX-def tupleX-def TupleX-def
      using assms i I comp-mkarr by simp
    also have ... = mkarr c (A i) (PrX I A i ∘ TupleX I c A F)
    proof −
      have «mkarr c (prodX I A) (TupleX I c A F) : c → prodX I A»
        by (metis assms c tupleX-def tupleX-in-hom)
      moreover have «mkarr (prodX I A) (A i) (PrX I A i) : prodX I A → A i»
      proof −
        have «prX I A i : prodX I A → A i»
          using assms(1−3) i by blast
        thus ?thesis
          by (simp add: prX-def)
      qed
      ultimately show ?thesis
        using assms i comp-mkarr [of c prodX I A TupleX I c A F A i PrX I A i]
        by auto
    qed
    also have ... = mkarr c (A i)
                      (λx. if TupleX I c A F x ∈ Set (prodX I A)
                           then OUT (ProdX I A) (TupleX I c A F x) i
                           else null)
      using I by (simp add: comp-def)
    also have ... = mkarr c (A i)
                      (λx. if x ∈ Set c then OUT (ProdX I A) (TupleX I c A F x) i else null)
    proof −
      have (λx. if TupleX I c A F x ∈ Set (prodX I A)
                 then OUT (ProdX I A) (TupleX I c A F x) i
                 else null) =
            (λx. if x ∈ Set c then OUT (ProdX I A) (TupleX I c A F x) i else null)
      proof
        fix x
        show (if TupleX I c A F x ∈ Set (prodX I A)
               then OUT (ProdX I A) (TupleX I c A F x) i
               else null) =
              (if x ∈ Set c then OUT (ProdX I A) (TupleX I c A F x) i else null)
          using assms TupleX-in-Hom
          by auto blast
      qed
      thus ?thesis by simp
    qed
    also have ... = mkarr c (A i)
                      (λx. if x ∈ Set c
                           then OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i
                           else null)
    proof −
      have (λx. if x ∈ Set c then OUT (ProdX I A) (TupleX I c A F x) i else null) =
            (λx. if x ∈ Set c
                 then OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i
                 else null)
      proof
        fix x
        show (if x ∈ Set c then OUT (ProdX I A) (TupleX I c A F x) i else null) =
              (if x ∈ Set c
               then OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i
               else null)
          unfolding TupleX-def by argo
      qed
      thus ?thesis by simp
    qed
    also have ... = mkarr c (A i) (λx. if x ∈ Set c then Fun (F i) x else null)
    proof −
      have (λx. if x ∈ Set c
                  then OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i
                  else null) =
             (λx. if x ∈ Set c then Fun (F i) x else null)
      proof
        fix x
        show (if x ∈ Set c
                then OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i
                else null) =
               (if x ∈ Set c then Fun (F i) x else null)
        proof (cases x ∈ Set c)
          case False
          show ?thesis
            using False by simp
          next
          case True
          show ?thesis
          proof −
            have (λi. Fun (F i) x) ∈ ProdX I A
              using assms(4−5) True Fun-def by auto
            hence OUT (ProdX I A) (IN (ProdX I A) (λi. Fun (F i) x)) i = Fun (F i) x
              using assms OUT-IN [of ProdX I A λi. Fun (F i) x]
                    small-ProdX embeds-ProdX
              by presburger
            thus ?thesis by simp
          qed
        qed
      qed
      thus ?thesis by simp
    qed
    also have ... = F i
    proof −
      have Fun (F i) = (λx. if x ∈ Set c then Fun (F i) x else null)
        using assms(4) i Fun-def by fastforce
      thus ?thesis
        using assms(4) i mkarr-Fun by force
    qed
    finally show ?thesis by blast
  qed
qed

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) =
    (λx. if x ∈ Set c then IN (ProdX I A) (λi. Fun (F i) x) else null)
proof –
  have Fun (tupleX I c A F) =
      (λx. if x ∈ Set c then mkarr c (prodX I A) (TupleX I c A F) · x else null)
    unfolding tupleX-def Fun-def
    apply simp
    by (metis ext mem-Collect-eq dom-mkarr seqE)
  also have ... = (λx. if x ∈ Set c then TupleX I c A F x else null)
    using assms app-mkarr
    by (metis (no-types, lifting) CollectD tupleX-def tupleX-in-hom)
  also have ... = (λx. if x ∈ Set c then IN (ProdX I A) (λi. Fun (F i) x) else null)
    unfolding TupleX-def by auto
  finally show ?thesis by blast
qed

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 –
  interpret J: category J
    using assms(1) discrete-diagram-def by blast
  interpret D: discrete-diagram J C D
    using assms(1) by blast
  let ?π = prX I D
  let ?a = prodX I D
  interpret A: constant-functor J C ?a
    using assms ide-prodX
    apply unfold-locales
    using D.is-discrete by auto
  interpret π: natural-transformation J C A.map D ?π
  proof
    fix j
    show ¬ J.arr j ⟹ prX I D j = null
      by (metis (no-types, lifting) D.as-nat-trans.extensionality ideD(1) mkarr-def
          not-arr-null prX-def)
    assume j: J.arr j
    show 1: arr (prX I D j)
      using D.is-discrete assms j by force
    show D j · prX I D (J.dom j) = prX I D j
      by (metis (lifting) 1 D.is-discrete J.ideD(2) comp-cod-arr cod-mkarr j prX-def)
    show prX I D (J.cod j) · A.map j = prX I D j
      by (metis (lifting) 1 A.map-simp D.is-discrete J.ide-char comp-arr-dom j
          dom-mkarr prX-def)
  qed
  show product-cone J C D ?a ?π
  proof
```

```
fix a' χ'
assume χ': D.cone a' χ'
interpret χ': cone J C D a' χ'
  using χ' by blast
show ∃!f. «f : a' → prodX I D» ∧ D.cones-map f (prX I D) = χ'
proof –
  let ?f = tupleX I a' D χ'
  have f: «?f : a' → prodX I D»
    using assms tupleX-in-hom
    by (metis D.is-discrete D.preserves-ide J.ide-char Pi-I'
      χ'.component-in-hom χ'.extensionality χ'.ide-apex mem-Collect-eq)
  moreover have D.cones-map ?f (prX I D) = χ'
  proof
    fix i
    show D.cones-map ?f (prX I D) i = χ' i
    proof –
      have J.arr i ⟹ prX I D i · ?f = χ' i
        using assms comp-prX-tupleX [of I D χ' a' i]
        by (metis D.is-discrete D.preserves-ide J.ide-char Pi-I'
          χ'.component-in-hom χ'.extensionality mem-Collect-eq)
      moreover have ¬ J.arr i ⟹ null = χ' i
        using χ'.extensionality by auto
      moreover have D.cone (cod ?f) (prX I D)
      proof –
        have D.cone (prodX I D) (prX I D) ..
        moreover have cod ?f = prodX I D
          using f by blast
        ultimately show ?thesis by auto
      qed
      ultimately show ?thesis
        using assms χ'.cone-axioms by auto
    qed
  qed
  moreover have ⋀f'. ⟦«f' : a' → prodX I D»; D.cones-map f' (prX I D) = χ'⟧
                    ⟹ f' = ?f
  proof –
    fix f'
    assume f': «f' : a' → prodX I D»
    assume 1: D.cones-map f' (prX I D) = χ'
    show f' = ?f
    proof (intro arr-eqI [of f'])
      show par: par f' ?f
        using f f' by fastforce
      show Fun f' = Fun (tupleX I a' D χ')
      proof
        fix x
        show Fun f' x = Fun (tupleX I a' D χ') x
        proof (cases x ∈ Set a')
          case False
```

```
show ?thesis
  using False par f' Fun-def by auto
next
case True
have 2: D.cone (cod f') (prX I D)
by (metis A.constant-functor-axioms Limit.cone-def
  π.natural-transformation-axioms χ' f' in-homE)
have Fun (tupleX I a' D χ') x = IN (ProdX I D) (λi. Fun (χ' i) x)
proof -
  have dom (tupleX I a' D χ') = a'
    using f by auto
  have *: (λx. if «x : 1^? → a'» then tupleX I a' D χ' · x else null) =
          (λx. if «x : 1^? → a'» then IN (ProdX I D) (λi. Fun (χ' i) x) else null)
  proof -
    have D ∈ I → Collect ide
      using assms(2) D.is-discrete by force
    moreover have ⋀i. i ∈ I ⟹ «χ' i : a' → D i»
      using assms(2) D.is-discrete χ'.component-in-hom by fastforce
    moreover have ⋀i. i ∉ I ⟹ χ' i = null
      using assms(2) χ'.extensionality by blast
    moreover have ide a'
      using χ'.ide-apex by auto
    ultimately show ?thesis
      using assms f Fun-tupleX [of I D χ' a'] Fun-arr by force
  qed
  have Fun (tupleX I a' D χ') x = tupleX I a' D χ' · x
    using True ‹dom (tupleX I a' D χ') = a'› Fun-def by presburger
  also have ... = (λx. if «x : 1^? → a'» then tupleX I a' D χ' · x else null) x
    using True by simp
  also have ... = (λx. if «x : 1^? → a'»
                    then IN (ProdX I D) (λi. Fun (χ' i) x)
                    else null) x
    using * by meson
  also have ... = IN (ProdX I D) (λi. Fun (χ' i) x)
    using True by simp
  finally show ?thesis by blast
qed
also have ... = IN (ProdX I D) (λi. χ' i · x)
  unfolding Fun-def
  by (metis J.dom-cod True χ'.A.map-simp χ'.cod-determines-component
    χ'.preserves-dom χ'.preserves-reflects-arr local.ext seqE)
also have ... = IN (ProdX I D) (λi. D.cones-map f' (prX I D) i · x)
  using 1 by simp
also have ... = IN (ProdX I D) (λi. (if J.arr i then prX I D i · f' else null) · x)
  using 2 by simp
also have ... = IN (ProdX I D) (λi. if J.arr i then prX I D i · (f' · x) else null)
proof -
  have (λi. (if J.arr i then prX I D i · f' else null) · x) =
        (λi. if J.arr i then prX I D i · (f' · x) else null)
  proof
    fix i
    show (if J.arr i then prX I D i · f′ else null) · x =
          (if J.arr i then prX I D i · (f′ · x) else null)
      using comp-assoc by auto
  qed
  thus ?thesis by simp
qed
also have ... = IN (ProdX I D)
                  (λi. if J.arr i then prX I D i · (Fun f′ x) else null)
  unfolding Fun-def
  using True f′ by auto
also have ... = IN (ProdX I D)
                  (λi. if J.arr i then Fun (prX I D i) (Fun f′ x) else null)
proof −
  have (λi. if J.arr i then prX I D i · (Fun f′ x) else null) =
        (λi. if J.arr i then Fun (prX I D i) (Fun f′ x) else null)
  proof
    fix i
    show (if J.arr i then prX I D i · (Fun f′ x) else null) =
          (if J.arr i then Fun (prX I D i) (Fun f′ x) else null)
      using f′ Fun-def by fastforce
  qed
  thus ?thesis by simp
qed
also have ... = IN (ProdX I D)
                  (λi. if J.arr i
                       then (if Fun f′ x ∈ Set (prodX I D)
                             then OUT (ProdX I D) (Fun f′ x) i else null)
                       else null)
proof −
  have ⋀i. J.arr i ⟹ Fun (prX I D i) =
                       (λx. if x ∈ Set (prodX I D)
                             then OUT (ProdX I D) x i else null)
    using assms Fun-prX D.is-discrete by force
  hence (λi. if J.arr i then Fun (prX I D i) (Fun f′ x) else null) =
        (λi. if J.arr i
             then (λx. if x ∈ Set (prodX I D)
                        then OUT (ProdX I D) x i else null)
                  (Fun f′ x)
             else null)
    by auto
  thus ?thesis by simp
qed
also have ... = IN (ProdX I D)
                  (λi. if J.arr i then OUT (ProdX I D) (Fun f′ x) i else null)
proof −
  have (λi. if J.arr i
             then (λx. if x ∈ Set (prodX I D)
                      then OUT (ProdX I D) x i else null)
                 (Fun f' x)
             else null) =
        (λi. if J.arr i then OUT (ProdX I D) (Fun f' x) i else null)
    using True f' Fun-def Fun-arr comp-in-homI by auto
  thus ?thesis by simp
qed
also have ... = IN (ProdX I D) (OUT (ProdX I D) (Fun f' x))
proof -
  have (λi. if J.arr i then OUT (ProdX I D) (Fun f' x) i else null) =
        OUT (ProdX I D) (Fun f' x)
  proof
    fix i
    show (if J.arr i then OUT (ProdX I D) (Fun f' x) i else null) =
          OUT (ProdX I D) (Fun f' x) i
    proof (cases J.arr i)
      case True
      show ?thesis
        using True by simp
      next
      case False
      have 1: Fun f' x ∈ Set (prodX I D)
        using True f' Fun-def by auto
      moreover have small (ProdX I D) and embeds (ProdX I D)
        using assms small-ProdX [of I D] embeds-ProdX [of I D]
              D.is-discrete D.preserves-ide
        by auto
      moreover have «Fun f' x : 1^? → mkide (ProdX I D)»
        using True f'
        by (metis 1 prodX-def mem-Collect-eq)
      ultimately have OUT (ProdX I D) (Fun f' x) ∈ ProdX I D
        using OUT-elem-of [of ProdX I D Fun f' x] Fun-in-Hom
        by fastforce
      thus ?thesis
        using False assms(2) by fastforce
    qed
  qed
  thus ?thesis by simp
qed
also have ... = Fun f' x
proof -
  have small (ProdX I D)
    using assms small-ProdX D.is-discrete by fastforce
  moreover have ∃ι. is-embedding-of ι (ProdX I D)
    using assms embeds-ProdX [of I D] D.is-discrete by auto
  moreover have Fun f' x ∈ Set (mkide (ProdX I D))
  proof -
    have Fun f' x ∈ Set (prodX I D)
      using Fun-in-Hom True f' by blast
```

```
          thus ?thesis
           by (simp add: prodX-def)
         qed
         ultimately show ?thesis
          using assms IN-OUT [of ProdX I D Fun f′ x] by blast
        qed
        finally show ?thesis by simp
       qed
      qed
     qed
    qed
    ultimately show ?thesis by blast
   qed
  qed
  thus has-as-product J D (prodX I D)
   using has-as-product-def by blast
 qed

 lemma has-small-products:
 assumes small I and I ⊆ Collect arr
 shows has-products I
 proof (unfold has-products-def, intro conjI)
  show I ≠ UNIV
   using assms not-arr-null by blast
  show ∀ J D. discrete-diagram J (·) D ∧ Collect (partial-composition.arr J) = I
         ⟶ (∃ a. has-as-product J D a)
   using assms product-cone-prodX by blast
 qed

end
```

4.8.1 Exported Notions

```
context sets-cat-with-tupling
begin

 interpretation Products: small-products-in-sets-cat ..

 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 −
  show OUT (ProdX I A) ∈ Set (prodX I A) → ProdX I A
  proof −
    have bij-betw
           (OUT ({f. ∀a. a ∈ I ⟶ f a ∈ Set (A a)} ∩ {f. ∀a. a ∉ I ⟶ f a = null}))
           (Set (prodX I A))
           ({f. ∀a. a ∈ I ⟶ f a ∈ Set (A a)} ∩ {f. ∀a. a ∉ I ⟶ f a = null})
      using Products.ide-prodX(2) assms(1−3) by blast
    then show ?thesis
      by (simp add: bij-betw-imp-funcset)
  qed
  show IN (ProdX I A) ∈ ProdX I A → Set (prodX I A)
  proof −
    have bij-betw
           (OUT ({f. ∀a. a ∈ I ⟶ f a ∈ Set (A a)} ∩ {f. ∀a. a ∉ I ⟶ f a = null}))
           (Set (prodX I A))
           ({f. ∀a. a ∈ I ⟶ f a ∈ Set (A a)} ∩ {f. ∀a. a ∉ I ⟶ f a = null})
      using Products.ide-prodX(2) assms(1−3) by blast
    then show ?thesis
      by (simp add: Products.prodX-def bij-betw-imp-funcset bij-betw-inv-into)
  qed
  show ⋀x. x ∈ Set (prodX I A) ⟹ IN (ProdX I A) (OUT (ProdX I A) x) = x
    using assms IN-OUT [of ProdX I A] Products.small-ProdX Products.embeds-ProdX
    by (simp add: Products.prodX-def)
  show ⋀y. y ∈ ProdX I A ⟹ OUT (ProdX I A) (IN (ProdX I A) y) = y
    using assms OUT-IN [of ProdX I A] Products.small-ProdX Products.embeds-ProdX
    by (simp add: Products.prodX-def)
  show bij-betw (OUT (ProdX I A)) (Set (prodX I A)) (ProdX I A)
    using assms Products.ide-prodX by fastforce
  show bij-betw (IN (ProdX I A)) (ProdX I A) (Set (prodX I A))
    using assms Products.ide-prodX by fastforce
qed

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
  using assms Products.Fun-prX by auto

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) =
      (λx. if x ∈ Set c then IN (Products.ProdX I A) (λi. Fun (F i) x) else null)
  using assms Products.Fun-tupleX by auto

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)
  using assms Products.product-cone-prodX by auto

lemma has-small-products:
assumes small I and I ⊆ Collect arr
shows has-products I
  using assms Products.has-small-products by blast
```

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 ≠ UNIV
shows has-products I
proof –
  obtain I' where I': I' ⊆ Collect arr ∧ I ≈ I'
    using assms inj-on-image-eqpoll-1 by auto
  have has-products I'
    using assms I'
    by (meson eqpoll-sym eqpoll-trans has-small-products small-def)
  thus ?thesis
    using assms(3) I' has-products-preserved-by-bijection
    by (metis eqpoll-def eqpoll-sym)
qed

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 ≡ ⋃i∈I. {i} × 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)
    using assms small-Set small-Union
    by (simp add: Pi-iff smaller-than-small)

  lemma embeds-CoprodX:
  assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
  shows embeds (CoprodX I A)
  proof
    let ?ι = (λx. pair (fst x) (snd x))
    show is-embedding-of ?ι (CoprodX I A)
    proof
      show ?ι ‘ CoprodX I A ⊆ Collect arr
        using arrI assms(3) some-pairing-in-univ by auto
      show inj-on ?ι (CoprodX I A)
      proof -
        have inj-on ?ι (Collect arr × Collect arr)
          using some-pairing-is-embedding by auto
        moreover have CoprodX I A ⊆ Collect arr × Collect arr
          using arrI assms(3) by auto
        ultimately show ?thesis
          by (meson inj-on-subset)
      qed
    qed
  qed

  lemma ide-coprodX:
  assumes small I and A ∈ I → Collect ide and I ⊆ 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 ⋀x. x ∈ Set (coprodX I A) ⟹ OUT (CoprodX I A) x ∈ CoprodX I A
  and ⋀y. y ∈ CoprodX I A ⟹ IN (CoprodX I A) y ∈ 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
  proof -
    show ide (coprodX I A)
      unfolding coprodX-def
      by (simp add: assms(1,2,3) small-CoprodX embeds-CoprodX ide-mkide(1))
    show 1: bij-betw (OUT (CoprodX I A)) (Set (coprodX I A)) (CoprodX I A)
      unfolding coprodX-def
      using assms small-CoprodX embeds-CoprodX bij-OUT [of CoprodX I A] by fastforce
    show 2: bij-betw (IN (CoprodX I A)) (CoprodX I A) (Set (coprodX I A))
      unfolding coprodX-def
      using assms small-CoprodX embeds-CoprodX bij-IN [of CoprodX I A] by fastforce
    show ⋀x. x ∈ Set (coprodX I A) ⟹ OUT (CoprodX I A) x ∈ CoprodX I A
      using 1 bij-betwE by blast
    show ⋀y. y ∈ CoprodX I A ⟹ IN (CoprodX I A) y ∈ Set (coprodX I A)
      using 2 bij-betwE by blast
    show ⋀x. x ∈ Set (coprodX I A) ⟹ IN (CoprodX I A) (OUT (CoprodX I A) x) = x
      using 1 bij-betw-inv-into-left
               [of OUT (CoprodX I A) Set (coprodX I A) CoprodX I A]
      by (auto simp add: coprodX-def)
    show ⋀y. y ∈ CoprodX I A ⟹ OUT (CoprodX I A) (IN (CoprodX I A) y) = y
      by (simp add: OUT-IN assms(1,2,3) small-CoprodX embeds-CoprodX)
  qed

  abbreviation InX :: 'a set ⇒ ('a ⇒ 'U) ⇒ 'a ⇒ 'U ⇒ 'U
  where InX I A i ≡ λx. if x ∈ Set (A i) then IN (CoprodX I A) (i, x) else null

  definition inX
  where inX I A i ≡ mkarr (A i) (coprodX I A) (InX I A i)

  lemma InX-in-Hom:
  assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
  and i ∈ I
  shows InX I A i ∈ Hom (A i) (coprodX I A)
    using assms ide-coprodX(2−3,5) by auto

  lemma inX-in-hom [intro, simp]:
  assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
  and i ∈ I
  shows in-hom (inX I A i) (A i) (coprodX I A)
    using assms ide-coprodX InX-in-Hom
    by (unfold inX-def, intro mkarr-in-hom) auto

  lemma inX-simps [simp]:
  assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
  and i ∈ I
  shows arr (inX I A i) and dom (inX I A i) = A i and cod (inX I A i) = coprodX I A
    using assms inX-in-hom by blast+

  lemma Fun-inX:
  assumes small I and A ∈ I → Collect ide and I ⊆ Collect arr
  and i ∈ I
```

```
shows Fun (inX I A i) = InX I A i
proof –
  have arr (inX I A i)
    by (simp add: assms)
  thus ?thesis
    by (simp add: inX-def)
qed

definition CotupleX :: 'a set ⇒ ('a ⇒ 'U) ⇒ ('a ⇒ 'U) ⇒ 'U ⇒ 'U
where CotupleX I A F ≡
       (λx. if x ∈ 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 ∈ I → Collect ide and I ⊆ Collect arr
and ⋀i. i ∈ I ⟹ «F i : A i → c» and ⋀i. i ∉ I ⟹ F i = null
shows CotupleX I A F ∈ Hom (coprodX I A) c
proof
  show CotupleX I A F ∈ {F. ∀x. x ∉ Set (coprodX I A) ⟶ F x = null}
    by (cases I = {}) (auto simp add: CotupleX-def)
  show CotupleX I A F ∈ Set (coprodX I A) → Set c
  proof (cases I = {})
    case False
    show ?thesis
    proof
      fix x
      assume x: x ∈ Set (coprodX I A)
      have OUT (CoprodX I A) x ∈ CoprodX I A
        using assms x ide-coprodX
        by (meson bij-betwE)
      hence ⋀i. i = fst (OUT (CoprodX I A) x) ⟹
                    «F i : A i → c» ∧ snd (OUT (CoprodX I A) x) ∈ Set (A i)
        using assms(4) by force
      thus CotupleX I A F x ∈ Set c
        using x CotupleX-def [of I A F] Fun-def by auto
    qed
    next
    case True
    show ?thesis
      by (metis (no-types, lifting) Pi-I' True True True True UN-E all-not-in-conv
          assms(1,3) bij-betwE ide-coprodX(2))
  qed
qed

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 ∈ 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 «cotupleX I c A F : coprodX I A → c»
  using assms ide-coprodX CotupleX-in-Hom
  unfolding cotupleX-def CotupleX-def
  by (intro mkarr-in-hom) auto

lemma cotupleX-simps [simp]:
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 arr (cotupleX I c A F)
and dom (cotupleX I c A F) = coprodX I A
and cod (cotupleX I c A F) = c
  using assms cotupleX-in-hom in-homE by blast+

lemma comp-cotupleX-inX:
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 i ∈ I ⟹ cotupleX I c A F · inX I A i = F i
proof −
  assume i: i ∈ I
  have I: I ≠ {}
    using i by blast
  show cotupleX I c A F · inX I A i = F i
  proof −
    have 1: cotupleX I c A F · inX I A i =
          mkarr (coprodX I A) c (CotupleX I A F) · mkarr (A i) (coprodX I A) (InX I A i)
      unfolding inX-def cotupleX-def CotupleX-def
      using assms i I comp-mkarr by simp
    also have ... = mkarr (A i) c (CotupleX I A F ∘ InX I A i)
      using assms i comp-mkarr
      by (metis (no-types, lifting) 1 seqI cotupleX-def cotupleX-simps(1)
          dom-mkarr inX-simps(1,3) seqE)
    also have ... = mkarr (A i) c
                      (λx. if x ∈ Set (A i)
                           then CotupleX I A F (IN (CoprodX I A) (i, x))
                           else null)
    proof −
      have CotupleX I A F ∘ InX I A i =
            (λx. if x ∈ Set (A i) then CotupleX I A F (IN (CoprodX I A) (i, x)) else null)
      proof
        fix x
        show (CotupleX I A F ∘ InX I A i) x =
              (if x ∈ Set (A i) then CotupleX I A F (IN (CoprodX I A) (i, x)) else null)
          unfolding CotupleX-def by auto
      qed
      thus ?thesis by simp
    qed
    also have ... = mkarr (A i) c
```

```
                  (λx. if x ∈ Set (A i)
                       then Fun (F (fst (OUT (CoprodX I A) (IN (CoprodX I A) (i, x)))))
                                (snd (OUT (CoprodX I A) (IN (CoprodX I A) (i, x))))
                       else null)
      proof −
        have ⋀x. x ∈ Set (A i) ⟹ IN (CoprodX I A) (i, x) ∈ Set (coprodX I A)
          using assms(1,2,3) i bij-betwE ide-coprodX(3) by blast
        hence (λx. if x ∈ Set (A i)
                   then CotupleX I A F (IN (CoprodX I A) (i, x))
                   else null) =
              (λx. if x ∈ Set (A i)
                   then Fun (F (fst (OUT (CoprodX I A) (IN (CoprodX I A) (i, x)))))
                            (snd (OUT (CoprodX I A) (IN (CoprodX I A) (i, x))))
                   else null)
          unfolding CotupleX-def by force
        thus ?thesis by simp
      qed
      also have ... = mkarr (A i) c (λx. if x ∈ Set (A i) then Fun (F i) x else null)
      proof −
        have ⋀x. x ∈ Set (A i) ⟹ OUT (CoprodX I A) (IN (CoprodX I A) (i, x)) = (i, x)
          using assms i ide-coprodX by auto
        hence (λx. if «x : 1? → A i»
                   then Fun (F (fst (OUT (CoprodX I A) (IN (CoprodX I A) (i, x)))))
                            (snd (OUT (CoprodX I A) (IN (CoprodX I A) (i, x))))
                   else null) =
              (λx. if «x : 1? → A i» then Fun (F i) x else null)
          by force
        thus ?thesis by simp
      qed
      also have ... = mkarr (A i) c (Fun (F i))
        by (metis (lifting) Fun-def assms(4) category.in-homE category-axioms
            i mem-Collect-eq)
      also have ... = F i
        using assms(4) i mkarr-Fun by blast
      finally show ?thesis by blast
    qed
  qed

  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) =
          (λx. if x ∈ Set (coprodX I A)
               then Fun (F (fst (OUT (CoprodX I A) x))) (snd (OUT (CoprodX I A) x))
               else null)
      using assms Fun-mkarr CotupleX-in-Hom CotupleX-def [of I A F] cotupleX-def cotupleX-simps(1)
    by (metis (lifting))
```

```
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 −
  interpret J: category J
    using assms(1) discrete-diagram-def by blast
  interpret D: discrete-diagram J C D
    using assms(1) by blast
  let ?π = inX I D
  let ?a = coprodX I D
  interpret A: constant-functor J C ?a
    using assms ide-coprodX
    using D.is-discrete by unfold-locales auto
  interpret π: natural-transformation J C D A.map ?π
  proof
    fix j
    show ¬ J.arr j ⟹ inX I D j = null
      by (metis (no-types, lifting) D.as-nat-trans.extensionality ideD(1)
          mkarr-def not-arr-null inX-def)
    assume j: J.arr j
    show 1: arr (inX I D j)
      using D.is-discrete assms j by force
    show inX I D (J.cod j) · D j = inX I D j
      by (metis (lifting) 1 D.is-discrete D.preserves-ide D.preserves-reflects-arr
          J.ideD(3) comp-arr-ide dom-mkarr ideD(3) j inX-def seqI)
    show A.map j · inX I D (J.dom j) = inX I D j
      by (metis (lifting) 1 A.map-simp D.is-discrete J.ide-char comp-cod-arr j
          cod-mkarr inX-def)
  qed
  show coproduct-cocone J C D ?a ?π
  proof
    fix a′ χ′
    assume χ′: D.cocone a′ χ′
    interpret χ′: cocone J C D a′ χ′
      using χ′ by blast
    show ∃!f. «f : coprodX I D → a′» ∧ D.cocones-map f (inX I D) = χ′
    proof −
      let ?f = cotupleX I a′ D χ′
      have f: «?f : coprodX I D → a′»
        using assms cotupleX-in-hom
        by (metis D.is-discrete D.preserves-ide J.ide-char Pi-I′
          χ′.component-in-hom χ′.extensionality χ′.ide-apex mem-Collect-eq)
      moreover have D.cocones-map ?f (inX I D) = χ′
      proof
        fix i
        show D.cocones-map ?f (inX I D) i = χ′ i
        proof −
          have J.arr i ⟹ ?f ⋅ inX I D i = χ' i
            using assms comp-cotupleX-inX
            by (metis D.is-discrete D.preserves-ide J.ide-char Pi-I'
                χ'.component-in-hom χ'.extensionality χ'.ide-apex mem-Collect-eq)
          moreover have ¬ J.arr i ⟹ null = χ' i
            using χ'.extensionality by auto
          moreover have D.cocone (dom ?f) (inX I D)
            by (metis A.constant-functor-axioms D.diagram-axioms
                π.natural-transformation-axioms cocone-def diagram-def f in-homE)
          ultimately show ?thesis
            using assms χ'.cocone-axioms by auto
        qed
      qed
      moreover have ⋀f'. ⟦«f' : coprodX I D → a'»; D.cocones-map f' (inX I D) = χ'⟧
                          ⟹ f' = ?f
      proof -
        fix f'
        assume f': «f' : coprodX I D → a'»
        assume 1: D.cocones-map f' (inX I D) = χ'
        show f' = ?f
        proof (intro arr-eqI [of f'])
          show par: par f' ?f
            using f f' by fastforce
          show Fun f' = Fun (cotupleX I a' D χ')
          proof
            fix x
            show Fun f' x = Fun (cotupleX I a' D χ') x
            proof (cases x ∈ Set (coprodX I D))
              case False
              show ?thesis
                using False par f' Fun-def by auto
              next
              case True
              have 2: D.cocone (dom f') (inX I D)
                by (metis A.constant-functor-axioms cocone-def
                    π.natural-transformation-axioms χ' f' in-homE)
              have Fun (cotupleX I a' D χ') x =
                    Fun (χ' (fst (OUT (CoprodX I D) x))) (snd (OUT (CoprodX I D) x))
              proof -
                have Fun (cotupleX I a' D χ') x = cotupleX I a' D χ' ⋅ x
                  using True f Fun-def by auto
                also have ... = (λx. if «x : 1^? → coprodX I D»
                                     then cotupleX I a' D χ' ⋅ x else null) x
                  using True by simp
                also have ... =
                          Fun (χ' (fst (OUT (CoprodX I D) x))) (snd (OUT (CoprodX I D) x))
                  using assms f True cotupleX-def [of I a' D χ'] CotupleX-def [of I D χ']
                        app-mkarr cotupleX-in-hom
                  by auto
        finally show ?thesis by blast
      qed
      also have ... = Fun f′ x
      proof (cases OUT (CoprodX I D) x)
        case (Pair i x′)
        have ix′: (i, x′) ∈ CoprodX I D
          using assms True Pair ide-coprodX(2) [of I D]
          by (metis (no-types, lifting) D.is-discrete D.preserves-ide Pi-I′
              bij-betwE mem-Collect-eq)
        have Fun (χ′ (fst (OUT (CoprodX I D) x))) (snd (OUT (CoprodX I D) x)) =
              Fun (χ′ i) x′
          by (simp add: Pair)
        also have ... = Fun (D.cocones-map f′ (inX I D) i) x′
          using 1 by simp
        also have ... = (f′ · inX I D i) · x′
          using assms 2 f′ ix′ inX-in-hom Fun-def D.extensionality D.is-discrete
            π.extensionality
          by auto
        also have ... = f′ · (inX I D i · x′)
          using comp-assoc by simp
        also have ... = f′ · IN (CoprodX I D) (i, x′)
        proof −
          have «inX I D i : D i → coprodX I D»
            using assms inX-in-hom D.is-discrete ix′ by fastforce
          hence «mkarr (D i) (coprodX I D) (InX I D i) : D i → coprodX I D»
            unfolding inX-def by simp
          thus ?thesis
            unfolding inX-def
            using assms ix′ app-mkarr by auto
        qed
        also have ... = f′ · x
        proof −
         have IN (CoprodX I D) (i, x′) = IN (CoprodX I D) (OUT (CoprodX I D) x)
            using Pair by simp
          also have ... = x
          proof −
            have small (CoprodX I D)
              using assms small-CoprodX D.is-discrete by fastforce
            thus ?thesis
              using assms True ide-coprodX(6) D.is-discrete D.preserves-ide
                Pi-I′ coprodX-def
              by force
          qed
          finally show ?thesis by simp
        qed
        finally show ?thesis
          using True f′ Fun-def by force
      qed
      finally show ?thesis by simp
```

```
            qed
          qed
        qed
      qed
      ultimately show ?thesis by blast
    qed
  qed
  thus has-as-coproduct J D (coprodX I D)
    using has-as-coproduct-def by blast
qed

lemma has-small-coproducts:
assumes small I and I ⊆ Collect arr
shows has-coproducts I
proof (unfold has-coproducts-def, intro conjI)
  show I ≠ UNIV
    using assms not-arr-null by blast
  show ∀ J D. discrete-diagram J (·) D ∧ Collect (partial-composition.arr J) = I
              ⟶ (∃ a. has-as-coproduct J D a)
    using assms coproduct-cocone-coprodX by blast
qed

end
```

4.9.1 Exported Notions

```
context sets-cat-with-cotupling
begin

  interpretation Coproducts: small-coproducts-in-sets-cat ..

  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))
    using assms Coproducts.ide-coprodX by auto

  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
    using assms Coproducts.Fun-inX by auto

  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) =
        (λx. 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)
    using assms Coproducts.Fun-cotupleX app-mkarr Coproducts.cotupleX-def by auto

  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)
    using assms Coproducts.coproduct-cocone-coprodX by auto

  lemma has-small-coproducts:
  assumes small I and I ⊆ Collect arr
  shows has-coproducts I
    using assms Coproducts.has-small-coproducts by blast

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 ..
```

```
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 −
  have 1: equiv A (A × A)
    using refl-on-def trans-on-def
    by (intro equivI symI) auto
  show 2: ∃r'. r ⊆ r' ∧ equiv A r' ∧ (∀s'. r ⊆ s' ∧ equiv A s' ⟶ r' ⊆ s')
  proof −
    let ?r' = ⋂ {s. equiv A s ∧ r ⊆ s}
    have r ⊆ ?r'
      by blast
    moreover have ∀s'. r ⊆ s' ∧ equiv A s' ⟶ ?r' ⊆ s'
      by blast
    moreover have equiv A ?r'
      using assms 1
      apply (intro equivI symI transI refl-onI)
         apply auto[4]
        apply (simp add: equiv-def refl-on-def)
       apply (meson equiv-def symD)
      by (meson equivE transE)
    ultimately show ?thesis by blast
  qed
  have r ⊆ equivcl A r ∧ equiv A (equivcl A r) ∧
          (∀s'. r ⊆ s' ∧ equiv A s' ⟶ equivcl A r ⊆ s')
    unfolding equivcl-def
    using 2 someI-ex [of λr'. r ⊆ r' ∧ equiv A r' ∧ (∀s'. r ⊆ s' ∧ equiv A s' ⟶ r' ⊆ s')]
    by fastforce
  thus r ⊆ equivcl A r and equiv A (equivcl A r)
  and ⋀s'. r ⊆ s' ∧ equiv A s' ⟹ equivcl A r ⊆ s'
    by auto
qed
```

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 ⇒ 'U ⇒ 'U set set
where Cod-coeq f g ≡ (λy. (equivcl (Set (cod f))
                          ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘ {y})) ‘ Set (cod f)

lemma small-Cod-coeq:
assumes par f g
shows small (Cod-coeq f g)
  using assms ide-cod small-Set by blast

lemma embeds-Cod-coeq:
assumes par f g
shows embeds (Cod-coeq f g)
and Cod-coeq f g ⊆ Pow (Set (cod f))
proof −
  show 1: Cod-coeq f g ⊆ Pow (Set (cod f))
  proof −
    let ?r = (λx. (f · x, g · x)) ‘ Set (dom f)
    have ?r ⊆ Set (cod f) × Set (cod f)
      using assms by auto
    hence equivcl (Set (cod f)) ?r ⊆ Set (cod f) × Set (cod f)
      using equivcl-props(3)
      by (metis (no-types, lifting) Sigma-cong equiv-type)
    thus ?thesis by blast
  qed
  show embeds (Cod-coeq f g)
  proof −
    have Cod-coeq f g ⊆ {X. X ⊆ Collect arr ∧ small X}
    proof −
      have Cod-coeq f g ⊆ {X. X ⊆ Collect arr}
        using 1 by blast
      moreover have Cod-coeq f g ⊆ {X. small X}
        using assms 1 small-Set smaller-than-small
        by (metis (no-types, lifting) HOL.ext Collect-mono Pow-def
            ide-cod subset-trans)
      ultimately show ?thesis by blast
    qed
    thus ?thesis
      using embeds-small-sets
      by (meson image-mono inj-on-subset subset-trans)
  qed
qed

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 ⋀x. x ∈ Set (cod-coeq f g) ⟹ OUT (Cod-coeq f g) x ∈ Cod-coeq f g
and ⋀y. y ∈ Cod-coeq f g ⟹ IN (Cod-coeq f g) y ∈ Set (cod-coeq f g)
and ⋀x. x ∈ Set (cod-coeq f g) ⟹ IN (Cod-coeq f g) (OUT (Cod-coeq f g) x) = x
and ⋀y. y ∈ Cod-coeq f g ⟹ OUT (Cod-coeq f g) (IN (Cod-coeq f g) y) = y
proof −
  have (λx. {f · x, g · x}) ‘ Set (dom f) ⊆ Pow (Set (cod f))
    using assms by auto
  show ide (cod-coeq f g)
    using small-Cod-coeq embeds-Cod-coeq assms cod-coeq-def by auto
  show 1: bij-betw (OUT (Cod-coeq f g)) (Set (cod-coeq f g)) (Cod-coeq f g)
    unfolding cod-coeq-def
    using assms ide-mkide bij-OUT small-Cod-coeq [of f g] embeds-Cod-coeq [of f g]
    by auto
  show 2: bij-betw (IN (Cod-coeq f g)) (Cod-coeq f g) (Set (cod-coeq f g))
    unfolding cod-coeq-def
    using assms ide-mkide bij-OUT bij-IN small-Cod-coeq [of f g] embeds-Cod-coeq
    by fastforce
  show ⋀x. x ∈ Set (cod-coeq f g) ⟹ OUT (Cod-coeq f g) x ∈ Cod-coeq f g
    using 1 bij-betwE by blast
  show ⋀y. y ∈ Cod-coeq f g ⟹ IN (Cod-coeq f g) y ∈ Set (cod-coeq f g)
    using 2 bij-betwE by blast
  show ⋀x. x ∈ Set (cod-coeq f g) ⟹ IN (Cod-coeq f g) (OUT (Cod-coeq f g) x) = x
    by (metis (no-types, lifting) HOL.ext 1 bij-betw-inv-into-left cod-coeq-def)
  show ⋀y. y ∈ Cod-coeq f g ⟹ OUT (Cod-coeq f g) (IN (Cod-coeq f g) y) = y
    by (metis (no-types, lifting) HOL.ext 1 bij-betw-inv-into-right cod-coeq-def)
qed

definition Coeq
where Coeq f g ≡ λy. if y ∈ Set (cod f)
                     then IN (Cod-coeq f g)
                            (equivcl (Set (cod f))
                               ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘ {y})
                     else null

lemma Coeq-in-Hom [intro]:
assumes par f g
shows Coeq f g ∈ Hom (cod f) (cod-coeq f g)
proof
  show Coeq f g ∈ Set (cod f) → Set (cod-coeq f g)
  proof
    fix y
    assume y: y ∈ Set (cod f)
    have Coeq f g y = IN (Cod-coeq f g)
                         (equivcl (Set (cod f))
                            ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘ {y})
      unfolding Coeq-def
      using y by simp
    moreover have ... ∈ Set (cod-coeq f g)
      using assms ide-cod-coeq(5) y by blast
    ultimately show Coeq f g y ∈ Set (cod-coeq f g) by simp
  qed
  show Coeq f g ∈ {F. ∀x. x ∉ Set (cod f) ⟶ F x = null}
    unfolding Coeq-def by simp
qed

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»
  using assms ide-cod-coeq(1) Coeq-in-Hom
  by (unfold coeq-def, intro mkarr-in-hom) auto

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
  using assms coeq-in-hom by blast+

lemma Fun-coeq:
assumes par f g
shows Fun (coeq f g) = Coeq f g
  using assms Fun-mkarr coeq-def coeq-simps(1) by presburger

lemma coeq-coequalizes:
assumes par f g
shows coeq f g · f = coeq f g · g
proof (intro arr-eqI)
  show par: par (coeq f g · f) (coeq f g · g)
    using assms by auto
  show Fun (coeq f g · f) = Fun (coeq f g · g)
  proof
    fix x
    show Fun (coeq f g · f) x = Fun (coeq f g · g) x
    proof (cases x ∈ Set (dom f))
      case False
      show ?thesis
        using assms False Fun-coeq Fun-def by simp
      next
      case True
      show ?thesis
      proof −
        have Fun (coeq f g · f) x = Fun (coeq f g) (Fun f x)
          using assms Fun-comp comp-in-homI coeq-in-hom comp-assoc by auto
        also have ... = Coeq f g (Fun f x)
```

```
          using assms True Fun-coeq
          by (metis (full-types, lifting))
        also have ... = IN (Cod-coeq f g)
                          (equivcl (Set (cod f))
                            ((λx. (f · x, g · x)) ‘ Set (dom f)) “ {f · x})
          unfolding Coeq-def
          using True assms Fun-def by auto
        also have ... = IN (Cod-coeq f g)
                          (equivcl (Set (cod f))
                            ((λx. (f · x, g · x)) ‘ Set (dom f)) “ {g · x})
        proof −
          have equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) “ {f · x} =
                equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) “ {g · x}
            using assms True
                  equivcl-props(2−3) [of (λx. (f · x, g · x)) ‘ Set (dom f) Set (cod f)]
                  equiv-class-eq-iff
                    [of Set (cod f)
                        equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f))
                        f · x g · x]
            by auto
          thus ?thesis by simp
        qed
        also have ... = Coeq f g (Fun g x)
          unfolding Coeq-def
          using True assms Fun-def by auto
        also have ... = Fun (coeq f g) (Fun g x)
          using assms True Fun-coeq
          by (metis (full-types, lifting))
        also have ... = Fun (coeq f g · g) x
          using assms Fun-comp comp-in-homI coeq-in-hom comp-assoc by auto
        finally show ?thesis by blast
      qed
    qed
  qed
qed

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 −
  have 1: (⋃x∈Set (dom f). {f · x, g · x}) ⊆ Set (cod f)
    using assms by auto
  have y: OUT (Cod-coeq f g) y ∈ Cod-coeq f g
    using assms ide-cod-coeq(2) [of f g] bij-betwE by blast
  obtain x where x: x ∈ Set (cod f) ∧
                    OUT (Cod-coeq f g) y =
                    equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) “{x}
    using assms y by blast
  hence 2: x ∈ OUT (Cod-coeq f g) y
  proof −
    have (λx. (f · x, g · x)) ‘ Set (dom f) ⊆ Set (cod f) × Set (cod f)
      using assms by auto
    hence x ∈ equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘{x}
      using assms x equivcl-props(3) [of (λx. (f · x, g · x)) ‘ Set (dom f) Set (cod f)]
            equiv-class-self
      by (metis (lifting))
    thus ?thesis
      using x by argo
  qed
  have Coeq f g x = y
  proof −
    have OUT (Cod-coeq f g) (Coeq f g x) =
          OUT (Cod-coeq f g)
            (IN (Cod-coeq f g)
               (equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘{x}))
      unfolding Coeq-def
      using x by presburger
    also have ... = equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘{x}
      using assms x y ide-cod-coeq(7) by (metis (lifting))
    also have ... = OUT (Cod-coeq f g) y
    proof −
      have OUT (Cod-coeq f g) y ∈ Cod-coeq f g
        using assms x by force

      thus ?thesis
        using assms x 1 2 by blast
    qed
    finally have IN (Cod-coeq f g) (OUT (Cod-coeq f g) (Coeq f g x)) =
                  IN (Cod-coeq f g) (OUT (Cod-coeq f g) y)
      by simp
    thus ?thesis
      using assms x y ide-cod-coeq(6) cod-coeq-def Coeq-def
      by (metis (lifting))
  qed
  thus ∃x. x ∈ Set (cod f) ∧ Coeq f g x = y
    using x by blast
qed

lemma coeq-is-coequalizer:
assumes par f g and Set (cod f) ≠ {}
shows has-as-coequalizer f g (coeq f g)
proof
  show par f g by fact
  show seq (coeq f g) f
    using assms by auto
  show coeq f g · f = coeq f g · g
    using assms coeq-coequalizes by blast
  show ⋀q'. ⟦seq q' f; q' · f = q' · g⟧ ⟹ ∃!h. h · coeq f g = q'
```

```
proof –
  fix q′
  assume seq: seq q′ f and eq: q′ · f = q′ · g
  let ?H = λy. if y ∈ Set (cod-coeq f g)
                then q′ · (SOME x. x ∈ Set (cod f) ∧ Coeq f g x = y)
                else null
  have H: ?H ∈ Hom (cod-coeq f g) (cod q′)
  proof
    show ?H ∈ Set (cod-coeq f g) → Set (cod q′)
    proof
      fix y
      assume y: y ∈ Set (cod-coeq f g)
      have ?H y = q′ · (SOME x. x ∈ Set (cod f) ∧ Coeq f g x = y)
        using y by simp
      moreover have ... ∈ Set (cod q′)
        using assms y someI-ex [of λx.  x ∈ Set (cod f) ∧ Coeq f g x = y]
              Coeq-surj seq in-homI
        by blast
      ultimately show ?H y ∈ Set (cod q′) by simp
    qed
    show ?H ∈ {F. ∀x. x ∉ Set (cod-coeq f g) ⟶ F x = null}
      by simp
  qed
  let ?h = mkarr (cod-coeq f g) (cod q′) ?H
  have h: «?h : cod-coeq f g → cod q′»
    using assms H ide-cod-coeq seq
    by (intro mkarr-in-hom) auto
  have ∗: ?h · coeq f g = q′
  proof (intro arr-eqI)
    show par: par (?h · coeq f g) q′
      using assms h seq by fastforce
    show Fun (?h · coeq f g) = Fun q′
    proof –
      have Fun (?h · coeq f g) = Fun ?h ∘ Fun (coeq f g)
        using Fun-comp par by blast
      also have ... = ?H ∘ Coeq f g
        using assms h Fun-coeq Fun-mkarr arrI by auto
      also have ... = Fun q′
      proof
        fix y
        show (?H ∘ Coeq f g) y = Fun q′ y
        proof (cases y ∈ Set (cod f))
          case False
          show ?thesis
            unfolding Coeq-def
            using False seq Fun-def by auto
          next
          case True
          have (?H ∘ Coeq f g) y =
      q' · (SOME x'. x' ∈ Set (cod f) ∧ Coeq f g x' = Coeq f g y)
  using Coeq-in-Hom True assms(1) by auto
also have ... = q' · y
proof −
  let ?e = (λx. (f · x, g · x)) ‘ Set (dom f)
  have e: ?e ⊆ Set (cod f) × Set (cod f)
    using assms by auto
  let ?ℰ = equivcl (Set (cod f)) ?e
  let ?ℰ' = {p ∈ Set (cod f) × Set (cod f). q' · fst p = q' · snd p}
  have ?ℰ ⊆ ?ℰ'
  proof −
    have equiv (Set (cod f)) ?ℰ'
      by (intro equivI symI) (auto simp add: refl-on-def trans-on-def)
    moreover have (λx. (f · x, g · x)) ‘ Set (dom f) ⊆ ?ℰ'
    proof −
      have ⋀x. x ∈ Set (dom f) ⟹ (f · x, g · x) ∈ ?ℰ'
      proof −
        fix x
        assume x: x ∈ Set (dom f)
        have (f · x, g · x) ∈ Set (cod f) × Set (cod f)
          using assms x by auto
        moreover have q' · f · x = q' · g · x
          using eq comp-assoc by metis
        ultimately show (f · x, g · x) ∈ ?ℰ' by fastforce
      qed
      thus ?thesis
        by (meson image-subsetI)
    qed
    ultimately show ?thesis
      by (meson equiv-type equivcl-props(4) subset-trans)
  qed
  moreover have ⋀y'. y' ∈ Set (cod f) ∧ Coeq f g y' = Coeq f g y
                       ⟹ (y', y) ∈ ?ℰ
  proof −
    fix y'
    assume y': y' ∈ Set (cod f) ∧ Coeq f g y' = Coeq f g y
    have eq: equivcl (Set (cod f)) ?e ‘‘ {y'} =
             equivcl (Set (cod f)) ?e ‘‘ {y}
      using assms(1) True y' ide-cod-coeq(7) [of f g]
      unfolding Coeq-def
      by (metis (mono-tags, lifting) image-eqI)
    moreover have y' ∈ equivcl (Set (cod f)) ?e ‘‘ {y'} ∧
                  y ∈ equivcl (Set (cod f)) ?e ‘‘ {y}
    proof
      have 1: equiv (Set (cod f)) (equivcl (Set (cod f)) ?e)
        by (simp add: e equivcl-props(3))
      show y' ∈ equivcl (Set (cod f)) ?e ‘‘ {y'}
        by (metis (lifting) 1 equiv-class-self y')
      show y ∈ equivcl (Set (cod f)) ((λx. (f · x, g · x)) ‘ Set (dom f)) ‘‘ {y}
```

```
                by (metis (no-types, lifting) 1 True equiv-class-self)
            qed
            ultimately show (y', y) ∈ ?ℰ by blast
          qed
          ultimately have ⋀y'. y' ∈ Set (cod f) ∧ Coeq f g y' = Coeq f g y
                                 ⟹ (y', y) ∈ ?ℰ'
            by (meson subsetD)
          thus ?thesis
            using True someI-ex [of λy'. y' ∈ Set (cod f) ∧ Coeq f g y' = Coeq f g y]
            by (metis (mono-tags, lifting) fst-conv mem-Collect-eq snd-conv)
        qed
        also have ... = Fun q' y
          using True seq Fun-def by auto
        finally show ?thesis by blast
      qed
    qed
    finally show ?thesis by blast
  qed
qed
moreover have ⋀h'. h' · coeq f g = q' ⟹ h' = ?h
proof -
  fix h'
  assume h': h' · coeq f g = q'
  show h' = ?h
  proof (intro arr-eqI [of h'])
    show par: par h' ?h
      using h h' seq
      by (metis (lifting) calculation cod-comp seqE)
    show Fun h' = Fun ?h
    proof -
      have 1: Fun h' ∘ Coeq f g = Fun ?h ∘ Coeq f g
        using assms h' * Fun-coeq Fun-comp seq seqE
        by (metis (lifting))
      show ?thesis
      proof
        fix z
        show Fun h' z = Fun ?h z
        proof (cases z ∈ Set (cod-coeq f g))
          case False
          show ?thesis
            using assms False h' par Fun-def by auto
          next
          case True
          obtain x where x: x ∈ Set (cod f) ∧ Coeq f g x = z
            using assms True Coeq-surj by blast
          show ?thesis
            using True x h' 1 * Fun-comp comp-apply
            by (metis (lifting))
        qed
```

```
            qed
          qed
        qed
      qed
      ultimately show ∃!h. h · coeq f g = q' by auto
    qed
  qed

  lemma has-coequalizers:
  assumes par f g
  shows ∃ e. has-as-coequalizer f g e
  proof (cases Set (cod f) = {})
    case False
    show ?thesis
      using assms False coeq-is-coequalizer by blast
    next
    case True
    have f = g
      using assms True
      by (metis arr-eqI' comp-in-homI empty-Collect-eq in-homI)
    hence has-as-coequalizer f g (cod f)
      using assms comp-arr-dom comp-cod-arr seqE
      by (intro has-as-coequalizerI) metis+
    thus ?thesis by blast
  qed

end
```

4.10.1 Exported Notions

```
context sets-cat-with-powering
begin

  interpretation Coeq: coequalizers-in-sets-cat sml C ..

  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 ⋀x. x ∈ Set (cod (coeq f g)) ⟹ OUT (Cod-coeq f g) x ∈ Cod-coeq f g
  and ⋀y. y ∈ Cod-coeq f g ⟹ IN (Cod-coeq f g) y ∈ Set (cod (coeq f g))
  and ⋀x. x ∈ Set (cod (coeq f g)) ⟹ IN (Cod-coeq f g) (OUT (Cod-coeq f g) x) = x
  and ⋀y. y ∈ Cod-coeq f g ⟹ OUT (Cod-coeq f g) (IN (Cod-coeq f g) y) = y
```

```
    using assms Coeq.ide-cod-coeq by auto

  lemma coeq-is-coequalizer:
  assumes par f g and Set (cod f) ≠ {}
  shows has-as-coequalizer f g (coeq f g)
    using assms Coeq.coeq-is-coequalizer by blast
```

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 ⊆ 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'
    using assms Coeq.equivcl-props [of r A]
    unfolding equivcl-def by auto

  lemma Fun-coeq:
  assumes par f g
  shows Fun (coeq f g) = (λy. if y ∈ Set (cod f)
                               then IN (Cod-coeq f g)
                                      (equivcl (Set (cod f))
                                         ((λx. (f · x, g · x)) ‘ Set (dom f)) “ {y})
                           else null)
    using assms Coeq.Fun-coeq Coeq.Coeq-def
    unfolding equivcl-def by auto

  lemma has-coequalizers:
  assumes par f g
  shows ∃ e. has-as-coequalizer f g e
   using assms Coeq.has-coequalizers by blast

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 −

  have y ∈ Set b ∨ y = null
    using assms popular-value-in-range [of F y] by blast
  hence y ≠ null ⟹ {x. F x = y} ⊆ Set a
    using assms by blast
  thus y = null
    using assms smaller-than-small small-Set by auto
qed

lemma memb-Exp-imp-small-function:
assumes ide a and ide b and F ∈ Exp a b
shows small-function F
proof
  show small (range F)
  proof −
    have range F ⊆ Set b ∪ {null}
      using assms by blast
    moreover have small ...
      using assms small-Set by auto
    ultimately show ?thesis
      using smaller-than-small by blast
  qed
  show at-most-one-popular-value F
    using assms memb-Exp-popular-value Uniq-def
    by (metis (no-types, lifting))
qed

lemma small-Exp:
assumes ide a and ide b
shows small (Exp a b)
proof −
  show ?thesis
  proof (cases small (UNIV :: 'U set))
    case False
   have Exp a b ⊆ {F. small-function F ∧ SF-Dom F ⊆ Set a ∧ range F ⊆ Set b ∪ {null}}
```

```
    proof
      fix F
      assume F: F ∈ Exp a b
      have small-function F
        using assms F memb-Exp-imp-small-function [of a b F] by blast
      moreover have SF-Dom F ⊆ Set a
      proof –
        have popular-value F null
        proof –

          have ⋀F y. F ∈ Exp a b ⟹ popular-value F y ⟹ y = null
            using assms memb-Exp-popular-value by meson
          moreover have ∃ y. popular-value F y
            by (metis (no-types, lifting) HOL.ext False assms(1,2) ex-popular-value-iff
                F memb-Exp-imp-small-function)
          ultimately show ?thesis
            using F by blast
        qed
        thus ?thesis
          using F by auto
      qed
      moreover have range F ⊆ Set b ∪ {null}
        using F by blast
      ultimately
      show F ∈ {F. small-function F ∧ SF-Dom F ⊆ Set a ∧ range F ⊆ Set b ∪ {null}}
        by blast
    qed
    thus ?thesis
      using False small-funcset [of Set a Set b ∪ {null}]
            small-Set assms(1,2) smaller-than-small
      by fastforce
    next
    case True
   have Exp a b ⊆ {F. small-function F ∧ SF-Dom F ⊆ UNIV ∧ range F ⊆ Set b ∪ {null}}
      using assms memb-Exp-imp-small-function by auto
    thus ?thesis
      using True small-funcset [of UNIV Set b ∪ {null}]
            small-Set assms(1,2) smaller-than-small
      by (metis (mono-tags, lifting) subset-UNIV)
  qed
qed

lemma embeds-Exp:
assumes ide a and ide b
shows embeds (Exp a b)
proof –
  have is-embedding-of some-embedding-of-small-functions (Exp a b)
  proof –
    have Exp a b ⊆ SEF
```

```
      unfolding EF-def
      using assms memb-Exp-imp-small-function by blast
    thus ?thesis
      using assms some-embedding-of-small-functions-is-embedding memb-Exp-popular-value
      by (meson image-mono inj-on-subset subset-trans)
  qed
  thus ?thesis by blast
qed

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 −
  have small (Exp a b)
    using assms small-Exp by blast
  moreover have embeds (Exp a b)
    using assms embeds-Exp by blast
  ultimately show ide (exp a b) and bij-betw (OUT (Exp a b)) (Set (exp a b)) (Exp a b)
    unfolding exp-def
    using assms ide-mkide bij-OUT by blast+
  thus bij-betw (IN (Exp a b)) (Exp a b) (Set (exp a b))
    using bij-betw-inv-into exp-def by fastforce
qed

abbreviation Eval
where Eval b c ≡ (λfx. if fx ∈ Set (prod (exp b c) b)
                       then OUT (Exp b c)
                              (Fun (pr1 (exp b c) b) fx)
                              (Fun (pr0 (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 (unfold eval-def, intro mkarr-in-hom)
  show ide c by fact
  show ide (prod (exp b c) b)
    using assms ide-exp ide-prod by auto
  show Eval b c ∈ Hom (prod (exp b c) b) c
  proof
    show Eval b c ∈ Set (prod (exp b c) b) → Set c
    proof
      fix fx
      assume fx: fx ∈ Set (prod (exp b c) b)
      have Eval b c fx = OUT (Exp b c) (Fun (pr_1 (exp b c) b) fx)
                             (Fun (pr_0 (exp b c) b) fx)
        using fx by simp
      moreover have ... ∈ Set c
      proof -
        have OUT (Exp b c) (Fun (pr_1 (exp b c) b) fx) ∈ Exp b c
        proof -
          have Fun (pr_1 (exp b c) b) fx ∈ Set (exp b c)
            using assms fx Fun-def
            by (simp add: comp-in-homI ide-exp(1))
          thus ?thesis
            using assms(1,2) bij-betwE ide-exp(2) by blast
        qed
        moreover have Fun (pr_0 (exp b c) b) fx ∈ Set b
          using assms(1,2) fx ide-exp(1) Fun-def by auto
        ultimately show ?thesis by blast
      qed
      ultimately show Eval b c fx ∈ Set c by auto
    qed
    show Eval b c ∈ {F. ∀x. x ∉ Set (prod (exp b c) b) ⟶ F x = null}
      by simp
  qed
qed

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
  using assms eval-in-hom by blast+

lemma Fun-eval:
assumes ide b and ide c
shows Fun (eval b c) = Eval b c
  using assms eval-def Fun-mkarr [of prod (exp b c) b c Eval b c]
  by (metis arrI eval-in-hom)

definition Curry
where Curry a b c ≡ λf. if «f : prod a b → c»
                        then mkarr a (exp b c)
                               (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)
                        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) =
        (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 −
  have ⋀x. x ∈ Set a ⟹
             IN (Exp b c) (λy. if y ∈ Set b then C f (tuple x y) else null)
               ∈ Set (exp b c)
  proof −
    fix x
    assume x: x ∈ Set a
    have (λy. if y ∈ Set b then C f (tuple x y) else null) ∈ Exp b c
    proof −
      have ⋀y. y ∈ Set b ⟹ C f (tuple x y) ∈ Set c
        using assms x by auto
      thus ?thesis by simp
    qed
    thus IN (Exp b c) (λy. if y ∈ Set b then C f (tuple x y) else null)
            ∈ Set (exp b c)
      using assms bij-betwE ide-exp
      by (metis (no-types, lifting))
  qed
  thus «Curry a b c f : a → exp b c»
    unfolding Curry-def
    using assms ide-exp
    by (simp, intro mkarr-in-hom, auto)
  show 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)
    using ‹«Curry a b c f : a → exp b c»› arrI assms(4) Curry-def app-mkarr
    by auto
qed

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
  using assms Curry-in-hom by blast+

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)
  using assms Curry-in-hom(2) by blast
```

```
interpretation elementary-category-with-terminal-object C ‹1^?› some-terminator
  using extends-to-elementary-category-with-terminal-object by blast

lemma is-category-with-terminal-object:
shows elementary-category-with-terminal-object C 1^? some-terminator
and category-with-terminal-object C
  ..

interpretation elementary-cartesian-closed-category
                 C pr_0 pr_1 ‹1^?› some-terminator exp eval Curry
proof
  show ⋀b c. ⟦ide b; ide c⟧ ⟹ «eval b c : prod (exp b c) b → c»
    using eval-in-hom by blast
  show ⋀b c. ⟦ide b; ide c⟧ ⟹ ide (exp b c)
    using ide-exp by blast
  show ⋀a b c g. ⟦ide a; ide b; ide c; «g : prod a b → c»⟧
          ⟹ «Curry a b c g : a → exp b c»
    using Curry-in-hom by simp
  show ⋀a b c g. ⟦ide a; ide b; ide c; «g : prod a b → c»⟧
          ⟹ C (eval b c) (prod (Curry a b c g) b) = g
  proof −
    fix a b c g
    assume a: ide a and b: ide b and c: ide c and g: «g : prod a b → c»
    show eval b c · prod (Curry a b c g) b = g
    proof (intro arr-eqI [of - g])
      show par: par (C (eval b c) (prod (Curry a b c g) b)) g
        using a b c g by auto
      show Fun (eval b c · prod (Curry a b c g) b) = Fun g
      proof
        fix x
        show Fun (eval b c · prod (Curry a b c g) b) x = Fun g x
        proof (cases x ∈ Set (prod a b))
          case False
          show ?thesis
            using False Fun-def
            by (metis g in-homE par)
          next
          case True
          have Fun (C (eval b c) (prod (Curry a b c g) b)) x =
                Fun (eval b c) (Fun (prod (Curry a b c g) b) x)
            using True a b c g Fun-comp par comp-assoc by auto
          also have ... =  (λfx. 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)
                            ((if x ∈ Set (prod a b)
                              then tuple
                                    (Fun (Curry a b c g) (pr_1 a b · x))
```

```
                    (Fun b (pr0 a b · x))
                else null))
proof -
  have Fun (eval b c) = (λfx. if fx ∈ Set (prod (exp b c) b)
                            then OUT (Exp b c) (Fun (pr1 (exp b c) b) fx)
                                   (Fun (pr0 (exp b c) b) fx)
                            else null)
    using b c Fun-eval by simp
  moreover have Fun (prod (Curry a b c g) b) =
              (λx. if x ∈ Set (prod a b)
                   then tuple
                          (Fun (Curry a b c g) (pr1 a b · x))
                          (Fun b (pr0 a b · x))
                   else null)
    using a b c g Fun-prod [of Curry a b c g a exp b c b b b] Curry-in-hom
    by (meson ide-in-hom)
  ultimately show ?thesis by simp
qed
also have ... = OUT (Exp b c)
                  (Fun (pr1 (exp b c) b)
                     (tuple
                        (Fun (Curry a b c g) (C (pr1 a b) x))
                        (Fun b (C (pr0 a b) x))))
                  (Fun (pr0 (exp b c) b)
                     (tuple
                        (Fun (Curry a b c g) (C (pr1 a b) x))
                        (Fun b (C (pr0 a b) x))))
 proof -
   have tuple
          (Fun (Curry a b c g) (C (pr1 a b) x))
          (Fun b (C (pr0 a b) x))
             ∈ Set (prod (exp b c) b)
     using a b c g True Fun-def by auto
   thus ?thesis
     using True by presburger
 qed
also have ... = OUT (Exp b c)
                  (pr1 (exp b c) b ·
                     tuple
                       (Fun (Curry a b c g) (C (pr1 a b) x))
                       (Fun b (C (pr0 a b) x)))
                  (pr0 (exp b c) b ·
                     tuple
                       (Fun (Curry a b c g) (C (pr1 a b) x))
                       (Fun b (C (pr0 a b) x)))
proof -
  have tuple
         (Fun (Curry a b c g) (C (pr1 a b) x))
         (Fun b (C (pr0 a b) x))
```

```
             ∈ Set (prod (exp b c) b)
    using a b c g True Fun-def by auto
  moreover have Set (prod (exp b c) b) = Set (dom (pr_1 (exp b c) b))
    using b c
    by (simp add: ide-exp(1))
  moreover have Set (prod (exp b c) b) = Set (dom (pr_0 (exp b c) b))
    using b c
    by (simp add: ide-exp(1))
  ultimately show ?thesis
    unfolding Fun-def
    using a b c g True by auto
qed
also have ... = OUT (Exp b c)
                  (Fun (Curry a b c g) (C (pr_1 a b) x))
                  (Fun b (C (pr_0 a b) x))
  unfolding Fun-def
  using True a b c g by auto
also have ... = OUT (Exp b c)
                   (Fun (Curry a b c g) (C (pr_1 a b) x))
                   (C (pr_0 a b) x)
proof -
  have C (pr_0 a b) x ∈ Set b
    using True a b by blast
  thus ?thesis
    using b Fun-ide [of b]
    by presburger
qed
also have ... = OUT (Exp b c)
                  ((λx. if x ∈ Set a
                        then IN (Exp b c)
                           (λy. if y ∈ Set b then g · tuple x y else null)
                        else null)
                      (C (pr_1 a b) x))
                  (C (pr_0 a b) x)
  using a b c g Fun-Curry [of a b c g] by simp
also have ... = OUT (Exp b c)
                  (IN (Exp b c)
                     (λy. if y ∈ Set b then g · tuple (pr_1 a b · x) y else null))
                     (pr_0 a b · x)
  using True a b c g by auto
also have ... = (λy. if y ∈ Set b then g · tuple (pr_1 a b · x) y else null)
                   (pr_0 a b · x)
proof -
  have (λy. if y ∈ Set b then g · tuple (pr_1 a b · x) y else null) ∈ Hom b c
  proof
    show (λy. if y ∈ Set b then g · tuple (pr_1 a b · x) y else null) ∈ Set b → Set c
    proof
      fix y
      assume y: y ∈ Set b
```

```
              show (if y ∈ Set b then g · tuple ($pr_1$ a b · x) y else null) ∈ Set c
                using True a b c g y by auto
            qed
            show (λy. if y ∈ Set b then g · tuple ($pr_1$ a b · x) y else null)
                     ∈ {F. ∀x. x ∉ Set b ⟶ F x = null}
              by auto
          qed
          thus ?thesis
            using a b c g small-Exp [of b c] embeds-Exp [of b c] ide-exp(1) [of b c]
                  OUT-IN
                    [of Exp b c
                        λy. if y ∈ Set b then g · tuple ($pr_1$ a b · x) y else null]
            by auto
        qed
        also have ... = g · tuple ($pr_1$ a b · x) ($pr_0$ a b · x)
          using True a b c g by auto
        also have ... = g · tuple ($pr_1$ a b) ($pr_0$ a b) · x
          using True a b c g comp-tuple-arr
          by (metis CollectD in-homE pr-simps(2) span-pr)
        also have ... = g · x
          using True a b tuple-pr comp-cod-arr by fastforce
        also have ... = Fun g x
          using True g Fun-def by auto
        finally show ?thesis by blast
      qed
    qed
  qed
qed
show ⋀a b c h. ⟦ide a; ide b; ide c; «h : a → exp b c»⟧
         ⟹ Curry a b c (C (eval b c) (prod h b)) = h
proof -
  fix a b c h
  assume a: ide a and b: ide b and c: ide c and h: «h : a → exp b c»
  show Curry a b c (C (eval b c) (prod h b)) = h
  proof (intro arr-eqI [of - h])
    show par: par (Curry a b c (C (eval b c) (prod h b))) h
      using a b c h Curry-def Curry-simps(1) by auto
    show Fun (Curry a b c (C (eval b c) (prod h b))) = Fun h
    proof
      fix x
      show Fun (Curry a b c (C (eval b c) (prod h b))) x = Fun h x
      proof (cases x ∈ Set a)
        case False
        show ?thesis
          using False a b c h
          by (metis Fun-def in-homE par)
        next
        case True
        have OUT (Exp b c) (Fun (Curry a b c (C (eval b c) (prod h b))) x) =
      OUT (Exp b c)
        (IN (Exp b c)
           (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null))
  using True a b c h Fun-Curry [of a b c C (eval b c) (prod h b)]
        eval-in-hom [of b c]
  by auto
also have ... = (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null)
proof –
  have (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null) ∈ Hom b c
  proof
    show (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null)
            ∈ Set b → Set c
    proof
      fix y
      assume y: y ∈ Set b
      show (if y ∈ Set b then (eval b c · prod h b) · tuple x y else null) ∈ Set c
        using True a b c h y ide-in-hom by auto
    qed
    show (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null)
            ∈ {F. ∀x. x ∉ Set b ⟶ F x = null}
      by simp
  qed
  thus ?thesis
    using True a b c h small-Exp [of b c] embeds-Exp ide-exp [of b c]
          OUT-IN
            [of Exp b c
               λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null]
    by auto
qed
also have ... = OUT (Exp b c) (Fun h x)
proof
  fix y
  show ... y = OUT (Exp b c) (Fun h x) y
  proof (cases y ∈ Set b)
    assume y: y ∉ Set b
    have «Fun h x : 1? → mkide (Exp b c)»
      using True b c h
      by (metis Fun-arr[of h a cod h] arr-iff-in-hom[of h · x]
          dom-comp[of h x] cod-comp[of h x] exp-def[of b c]
          in-homE[of h a exp b c] in-homE[of x 1? a]
          mem-Collect-eq[of x λuub. «uub : 1? → a»] seqI[of x h])
    thus ?thesis
      using True b c h y OUT-elem-of [of Exp b c Fun h x] small-Exp [of b c]
            embeds-Exp [of b c] ide-exp [of b c]
      by auto
    next
    assume y: y ∈ Set b
    have (λy. if y ∈ Set b then (eval b c · prod h b) · tuple x y else null) y =
          (eval b c · prod h b) · tuple x y
      using y by simp
    also have ... = eval b c · (prod h b · tuple x y)
      using comp-assoc by simp
    also have ... = eval b c · tuple (h · x) (b · y)
      using True b c h y prod-tuple
      by (metis comp-cod-arr in-homE mem-Collect-eq seqI)
    also have ... = eval b c · tuple (h · x) y
      using b y
      by (metis comp-cod-arr in-homE mem-Collect-eq)
    also have ... = Fun (eval b c) (tuple (h · x) y)
      using True b c h y Fun-def [of eval b c tuple (h · x) y] by auto
    also have ... = (λfx. 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)
                    (tuple (h · x) y)
      using b c Fun-eval [of b c] by presburger
    also have ... = OUT (Exp b c) (Fun (pr_1 (exp b c) b) (tuple (h · x) y))
                   (Fun (pr_0 (exp b c) b) (tuple (h · x) y))
      using True b c h y
      by (simp add: comp-in-homI tuple-in-hom)
    also have ... = OUT (Exp b c) (pr_1 (exp b c) b · tuple (h · x) y)
                   (pr_0 (exp b c) b · tuple (h · x) y)
      using True b c h y Fun-def ide-exp(1) span-pr by auto
    also have ... = OUT (Exp b c) (h · x) y
      using True b c h y
      apply auto
      by fastforce
    also have ... = OUT (Exp b c) (Fun h x) y
      using True h Fun-def by auto
    finally show (if y ∈ Set b then (eval b c · prod h b) · tuple x y else null) =
                 OUT (Exp b c) (Fun h x) y
      by blast
  qed
qed
finally have *: OUT (Exp b c) (Fun (Curry a b c (C (eval b c) (prod h b))) x) =
                OUT (Exp b c) (Fun h x)
  by simp
show Fun (Curry a b c (C (eval b c) (prod h b))) x = Fun h x
proof -
  have Fun (Curry a b c (C (eval b c) (prod h b))) x =
       IN (Exp b c) (OUT (Exp b c) (Fun (Curry a b c (C (eval b c) (prod h b))) x))
  proof -
    have Fun (Curry a b c (eval b c · prod h b)) x ∈ Set (mkide (Exp b c))
    proof -
      have «Curry a b c (eval b c · prod h b) : a → exp b c»
        using a b c h par
              Curry-in-hom [of a b c C (eval b c) (prod h b)]
        by (metis arr-iff-in-hom in-homE)
```

```
                  hence Fun (Curry a b c (eval b c · prod h b)) ∈ Set a → Set (exp b c)
                    using Fun-in-Hom [of Curry a b c (eval b c · prod h b) a exp b c]
                    by blast
                  thus ?thesis
                    using True exp-def by auto
                qed
                thus ?thesis
                  using True a b c h small-Exp embeds-Exp
                        IN-OUT [of Exp b c Fun (Curry a b c (C (eval b c) (prod h b))) x]
                  by presburger
              qed
              also have ... = IN (Exp b c) (OUT (Exp b c) (Fun h x))
                using ∗ by simp
              also have ... = Fun h x
              proof −
                have Fun h x ∈ Set (mkide (Exp b c))
                  using True b c h Fun-def exp-def by auto
                thus ?thesis
                  using True b c h small-Exp embeds-Exp
                        IN-OUT [of Exp b c Fun h x]
                  by presburger
              qed
              finally show ?thesis by blast
            qed
          qed
        qed
      qed
    qed
  qed

  lemma is-elementary-cartesian-closed-category:
  shows elementary-cartesian-closed-category C pr0 pr1 1? some-terminator exp eval Curry
    ..

  lemma is-cartesian-closed-category:
  shows cartesian-closed-category C
    ..

end
```

4.11.1 Exported Notions

```
context sets-cat-with-tupling
begin

  sublocale sets-cat-with-pairing ..

  interpretation Expos: exponentials-in-sets-cat sml C ..
```

```
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)
  using assms Expos.ide-exp by blast

lemma exp-comparison-map-props:
assumes ide a and ide b
shows OUT (Exp a b) ∈ Set (exp a b) → Exp a b
and IN (Exp a b) ∈ Exp a b → Set (exp a b)
and ⋀x. x ∈ Set (exp a b) ⟹ IN (Exp a b) (OUT (Exp a b) x) = x
and ⋀y. y ∈ Exp a b ⟹ 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 −
  show OUT (Exp a b) ∈ Set (exp a b) → Exp a b
    using assms Expos.ide-exp(2) [of a b] bij-betw-def bij-betw-imp-funcset
    by simp
  thus IN (Exp a b) ∈ Exp a b → Set (exp a b)
    using assms Expos.exp-def
  by (metis (no-types, lifting) HOL.ext Expos.ide-exp(2) bij-betw-imp-funcset bij-betw-inv-into)
  show ⋀x. x ∈ Set (exp a b) ⟹ IN (Exp a b) (OUT (Exp a b) x) = x
    using assms
   by (metis (no-types, lifting) HOL.ext Expos.exp-def Expos.ide-exp(2) bij-betw-inv-into-left)
  show ⋀y. y ∈ Exp a b ⟹ OUT (Exp a b) (IN (Exp a b) y) = y
    using assms
  by (metis (no-types, lifting) HOL.ext Expos.exp-def Expos.ide-exp(2) bij-betw-inv-into-right)
  show bij-betw (OUT (Exp a b)) (Set (exp a b)) (Exp a b)
    using assms Expos.exponentials-in-sets-cat-axioms exponentials-in-sets-cat.ide-exp(2)
    by fastforce
  show bij-betw (IN (Exp a b)) (Exp a b) (Set (exp a b))
    using assms Expos.exponentials-in-sets-cat-axioms exponentials-in-sets-cat.ide-exp(3)
    by fastforce
qed

abbreviation Eval
where Eval ≡ Expos.Eval

abbreviation eval
where eval ≡ Expos.eval

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

```
  using assms Expos.eval-in-hom by blast

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
  using assms Expos.eval-simps by auto

lemma Fun-eval:
assumes ide b and ide c
shows Fun (eval b c) = Eval b c
  unfolding eval-def
  using assms Expos.Fun-eval [of b c] by simp

abbreviation Curry
where Curry ≡ Expos.Curry

lemma Curry-in-hom [intro, simp]:
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»
  using assms Expos.Curry-in-hom by auto

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
  using assms Expos.Curry-simps by auto

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)
  using assms Expos.Fun-Curry by blast

theorem is-cartesian-closed:
shows elementary-cartesian-closed-category C pr_0 pr_1 1^? some-terminator exp eval Curry
and cartesian-closed-category C
  using Expos.is-elementary-cartesian-closed-category Expos.is-cartesian-closed-category
  by auto

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
    using assms subobject-classifier-def by blast

  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
    using assms subobject-classifier-def by force

end

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

  sublocale category-with-terminal-object
    using category-axioms category-with-terminal-object.intro
          category-with-terminal-object-axioms-def has-subobject-classifier-ax
    by force
```

```
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 ≡ λy. 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»
    using assms ide-two ff-in-hom tt-in-hom χ-def mkarr-in-hom by auto

  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
    using assms χ-in-hom by blast+

  lemma Fun-χ:
  assumes «m : b → a» and mono m
  shows Fun (χ m) = Chi m
    unfolding χ-def
    using assms Fun-mkarr
    by (metis (no-types, lifting) χ-def χ-in-hom arrI)

  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 −
    have {y. y ∈ Set a ∧ χ m · y = tt} = {y. y ∈ Set a ∧ Chi m y = tt}
    proof −
      have ⋀y. y ∈ Set a ⟹ χ m · y = tt ⟷ Chi m y = tt
        by (metis Fun-χ Fun-arr χ-in-hom assms(1,2))
      thus ?thesis by blast
    qed
    moreover have bij-betw (Fun m) (Set b) {y. y ∈ Set a ∧ Chi m y = tt}
      unfolding bij-betw-def
      using assms mono-char tt-def ff-def tt-ne-ff Fun-def by auto
    ultimately show ?thesis by simp
  qed

  lemma has-subobject-classifier:
  shows subobject-classifier tt
  proof
    show «tt : 1? → 2»
      using tt-in-hom by blast
    show terminal 1?
      using terminal-some-terminal by blast
    show mono tt
      using mono-tt by blast
    fix m
    assume m: mono m
    define b where b-def: b = dom m
    define a where a-def: a = cod m
    have m: «m : b → a» ∧ mono m
      using m a-def b-def mono-implies-arr by blast
    have bij-Fun-m: bij-betw (Fun m) (Set b) {y ∈ Set a. χ m · y = tt}
      using m bij-Fun-mono by presburger
    have ∃!χ. «χ : a → 2» ∧ has-as-pullback tt χ t?[b] m
    proof −
      have 1: «χ m : a → 2»
        using m χ-in-hom by blast
      moreover have 2: has-as-pullback tt (χ m) t?[b] m
      proof
        show cs: commutative-square tt (χ m) t?[b] m
        proof
          show cospan tt (χ m)
            by (metis (lifting) χ-in-hom arr-iff-in-hom m in-homE mono-char tt-simps(1,3))
          show span: span t?[b] m
            using m by auto
          show dom tt = cod t?[b]
            using m by auto
          show tt · t?[b] = χ m · m
          proof (intro arr-eqI)
            show par: par (tt · t?[b]) (χ m · m)
              using m ‹span t?[b] m› a-def b-def by auto
            show Fun (tt · t?[b]) = Fun (χ m · m)
            proof
              fix x
              show Fun (tt · t?[b]) x = Fun (χ m · m) x
              proof (cases x ∈ Set b)
                case False
                show ?thesis
                  using False par m Fun-def by auto
                next
                case True
          have Fun (tt · t?[b]) x = Fun tt (Fun t?[b] x)
            using Fun-comp par by auto
          also have ... = (λx. if x ∈ Set 1? then tt else null)
                          (if x ∈ Set b then 1? else null)
            using Fun-some-terminator Fun-tt span b-def ide-dom by auto
          also have ... = tt
            using True ide-in-hom ide-some-terminal by auto
          also have ... = (λx. if x ∈ Set a then tt else null) (Fun m x)
            using m True Fun-def
            by (metis CollectD CollectI in-homE comp-in-homI)
          also have ... = Chi m (Fun m x)
            using app-mkarr m Fun-def by auto
          also have ... = Fun (χ m) (Fun m x)
            using m Fun-χ [of m b a] by simp
          also have ... = Fun (χ m · m) x
            by (metis comp-eq-dest-lhs par Fun-comp)
          finally show ?thesis by blast
        qed
      qed
    qed
  qed
  show ⋀h k. commutative-square tt (χ m) h k ⟹ ∃!l. t?[b] · l = h ∧ m · l = k
  proof −
    fix h k
    assume hk: commutative-square tt (χ m) h k
    have inj-m: inj-on (Fun m) (Set b)
      using m mono-char by blast
    have kx: ⋀x. x ∈ Set (dom h) ⟹ k · x ∈ {y ∈ Set a. χ m · y = tt}
    proof −
      fix x
      assume x: x ∈ Set (dom h)
      have χ m · k · x  = tt · h · x
        using hk comp-assoc
        by (metis (no-types, lifting) commutative-squareE)
      hence χ m · k · x  = tt
        by (metis (lifting) IntI Int-Collect comp-arr-dom comp-in-homI' in-homE
            commutative-squareE hk ide-some-terminal ide-in-hom some-trm-eqI
            tt-simps(2) x)
      thus k · x ∈ {y ∈ Set a. χ m · y = tt}
        using hk comp-assoc
        by (metis (mono-tags, lifting) 1 dom-comp in-homE in-homI mem-Collect-eq
            seqE tt-simps(1,2))
    qed
    let ?l = mkarr (dom h) b
                (λx. if x ∈ Set (dom h) then inv-into (Set b) (Fun m) (k · x) else null)
    have l: «?l : dom h → b»
    proof (intro mkarr-in-hom)
      show ide (dom h)
        using hk ide-dom by blast
    show ide b
      using m by auto
    show (λx. if x ∈ Set (dom h) then inv-into (Set b) (Fun m) (k · x) else null)
            ∈ Hom (dom h) b
    proof
      show (λx. if x ∈ Set (dom h) then inv-into (Set b) (Fun m) (k · x) else null)
              ∈ Set (dom h) → Set b
      proof
        fix x
        assume x: x ∈ Set (dom h)
        have inv-into (Set b) (Fun m) (k · x) ∈ Set b ∧
                        Fun m (inv-into (Set b) (Fun m) (k · x)) = k · x
          using x bij-Fun-m kx
          by (meson bij-betw-apply bij-betw-inv-into bij-betw-inv-into-right)
        thus (if x ∈ Set (dom h) then inv-into (Set b) (Fun m) (k · x) else null)
                ∈ Set b
          using x by presburger
      qed
      show (λx. if x ∈ Set (dom h) then inv-into (Set b) (Fun m) (k · x) else null)
              ∈ {F. ∀x. x ∉ Set (dom h) ⟶ F x = null}
        by auto
    qed
  qed
  have t^?[b] · ?l = h
    by (metis (lifting) commutative-square-def comp-cod-arr
        elementary-category-with-terminal-object.trm-naturality
        elementary-category-with-terminal-object.trm-one
        extends-to-elementary-category-with-terminal-object hk in-homE l
        tt-simps(2))
  moreover have m · ?l = k
  proof (intro arr-eqI)
    show par: par (m · ?l) k
      by (metis (no-types, lifting) HOL.ext χ-simps(2) m cod-comp dom-comp seqI'
          commutative-squareE hk in-homE l)
    show Fun (m · ?l) = Fun k
    proof
      fix x
      show Fun (m · ?l) x = Fun k x
      proof (cases x ∈ Set (dom h))
        case False
        show ?thesis
          using False par commutative-square-def Fun-def by auto
        next
        case True
        have Fun (m · ?l) x = Fun m (Fun ?l x)
          using True Fun-comp CollectI m comp-in-homI in-homE l comp-assoc par
          by fastforce
        also have ... = Fun m (inv-into (Set b) (Fun m) (k · x))
          using True m app-mkarr l by auto
```

```
          also have ... = k · x
            using True bij-Fun-m bij-betw-inv-into-right kx by force
          also have ... = Fun k x
            using True hk Fun-def by fastforce
          finally show ?thesis by blast
        qed
      qed
    qed
    ultimately have 1: t^?[b] · ?l = h ∧ m · ?l = k by blast
    moreover have ⋀l'. t^?[b] · l' = h ∧ m · l' = k ⟹ l' = ?l
      using m l
      by (metis (lifting) ‹m · ?l = k› seqI' mono-cancel)
    ultimately show ∃!l. t^?[b] · l = h ∧ m · l = k by auto
  qed
qed
moreover have ⋀χ'. «χ' : a → 2» ∧ has-as-pullback tt χ' t^?[b] m ⟹ χ' = χ m
proof –
  fix χ'
  assume χ': «χ' : a → 2» ∧ has-as-pullback tt χ' t^?[b] m
  show χ' = χ m
  proof (intro arr-eqI' [of χ'])
    show «χ' : a → 2»
      using χ' by simp
    show «χ m : a → 2»
      using 1 by force
    show ⋀y. «y : 1^? → a» ⟹ χ' · y = χ m · y
    proof –
      fix y
      assume y: «y : 1^? → a»
      show χ' · y = χ m · y
      proof (cases y ∈ Set a)
        case False
        show ?thesis
         using False y by blast
        next
        case True
        show ?thesis
        proof (cases y ∈ Fun m ‘ Set b)
          case True
          obtain x where x: x ∈ Set b ∧ y = Fun m x
            using True by blast
          have χ' · y = χ' · m · x
            using x y Fun-def by auto
          also have ... = tt · 1^?
            using χ' x Fun-def
            by (metis (no-types, lifting) HOL.ext Fun-some-terminator m
                commutative-square-def has-as-pullbackE ide-dom in-homE comp-assoc)
          also have ... = χ m · m · x
            using 1 2 x χ-def app-mkarr m comp-arr-dom y Fun-def by auto
```

```
              also have ... = χ m · y
                using x y Fun-def by auto
              finally show ?thesis by blast
              next
              case False
              have χ' · y = ff
              proof −
                have χ' · y = tt ⟹ False
                proof −
                  assume 3: χ' · y = tt
                  hence commutative-square tt χ' 1^? y
                    by (metis ‹«χ' : a → 2»› commutative-squareI comp-arr-dom ideD(1,2,3)
                        ide-some-terminal in-homE tt-simps(1,2,3) y)
                  hence ∃x. x ∈ Set b ∧ m · x = y ∧ t^?[b] · x = 1^?
                    using χ' has-as-pullbackE [of tt χ' t^?[b] m]
                    by (metis arr-iff-in-hom m dom-comp in-homE mem-Collect-eq seqE y)
                  thus False
                    using False χ' m Fun-def by auto
                qed
                thus ?thesis
                  using Set-two χ' y by blast
              qed
              also have ... = χ m · y
                using 1 False app-mkarr m y χ-def by auto
              finally show ?thesis by blast
            qed
          qed
        qed
      qed
    qed
    ultimately show ∃!χ. «χ : a → 2» ∧ has-as-pullback tt χ t^?[b] m
      by blast
  qed
  moreover have t^?[b] = (THE t. «t : dom m → 1^?»)
    using terminal-some-terminal the1-equality [of λt. «t : dom m → 1^?»]
    by (simp add: b-def m mono-implies-arr some-terminator-def)
  ultimately show ∃!χ. «χ : cod m → 2» ∧
                        has-as-pullback tt χ (THE t. «t : dom m → 1^?») m
    using m by auto
qed

sublocale category-with-subobject-classifier
  using has-subobject-classifier
  by unfold-locales auto

lemma is-category-with-subobject-classifier:
shows category-with-subobject-classifier C
  ..
```

```
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)
    using small-nat embeds-nat bij-OUT bij-IN by auto

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

  lemma Zero-in-Hom:
  shows Zero ∈ Hom 1? N
    using Pi-I' bij-betwE ide-nat(3) by fastforce

  definition zero
  where zero ≡ mkarr 1? N Zero

  lemma zero-in-hom [intro, simp]:
  shows «zero : 1? → N»
    using mkarr-in-hom [of 1? N] Zero-in-Hom ide-nat(1) ide-some-terminal zero-def
    by presburger

  lemma zero-simps [simp]:
  shows arr zero and dom zero = 1? and cod zero = N
    using zero-in-hom by blast+

  lemma Fun-zero:
  shows Fun zero = Zero
    using zero-def app-mkarr zero-in-hom zero-simps(2) by auto
```

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

lemma Succ-in-Hom:
shows Succ ∈ Hom N N
  using Pi-I′ bij-betwE ide-nat(3) by fastforce

definition succ
where succ ≡ mkarr N N Succ

lemma succ-in-hom [intro]:
shows «succ : N → N»
  using Succ-in-Hom ide-nat(1) succ-def by auto

lemma succ-simps [simp]:
shows arr succ and dom succ = N and cod succ = N
  using succ-in-hom by blast+

lemma Fun-succ:
shows Fun succ = Succ
  using succ-def app-mkarr succ-in-hom succ-simps(2) by auto

lemma nat-universality:
assumes «Z : 1? → a» and «S : a → a»
shows ∃!f. «f : N → a» ∧ f · zero = Z ∧ f · succ = S · f
proof –
  let ?F = λn. if n ∈ Set N then ((·) S ^^ OUT (UNIV :: nat set) n) Z else null
  have F: ?F ∈ Hom N a
  proof
    show ?F ∈ {F. ∀x. x ∉ Set (mkide (UNIV :: nat set)) ⟶ F x = null} by simp
    show ?F ∈ Set N → Set a
    proof
      have 1: ⋀k. ((·) S ^^ k) Z ∈ Set a
      proof –
        fix k
        show ((·) S ^^ k) Z ∈ Set a
          using assms by (induct k) auto
      qed
      fix n
      assume n: n ∈ Set N
      show ?F n ∈ Set a
        using n 1 by auto
    qed
  qed
  let ?f = mkarr N a ?F
  have f: «?f : N → a»
    using mkarr-in-hom F assms(2) ide-nat(1) by auto
  have «?f : N → a» ∧ ?f · zero = Z ∧ ?f · succ = S · ?f
  proof (intro conjI)
```

```
show «?f : N → a» by fact
show ?f · zero = Z
proof (intro arr-eqI)
  show par: par (?f · zero) Z
    using assms(1) f by fastforce
  show Fun (?f · zero) = Fun Z
  proof –
    have Fun (?f · zero) = Fun ?f ∘ Fun zero
      using Fun-comp par by blast
    also have ... = ?F ∘ Zero
      using Fun-mkarr Fun-zero par by fastforce
    also have ... = Fun Z
    proof
      fix x
      show (?F ∘ Zero) x = Fun Z x
      proof (cases x ∈ Set 1^?)
        case False
        show ?thesis
          using False par Fun-def by auto
        next
        case True
        have (?F ∘ Zero) x =
              ((·) S ^^ OUT (UNIV :: nat set) (IN (UNIV :: nat set) 0)) Z
          using True bij-betw-imp-surj-on ide-nat(3) by fastforce
        also have ... = ((·) S ^^ 0) Z
          using OUT-IN [of UNIV :: nat set 0 :: nat] small-nat embeds-nat
          by simp
        also have ... = Fun Z x
          using True Fun-def
          by (metis assms(1) comp-arr-dom funpow-0 ide-in-hom ide-some-terminal
              in-homE mem-Collect-eq some-trm-eqI)
        finally show ?thesis by blast
      qed
    qed
    finally show ?thesis by blast
  qed
qed
show ?f · succ = S · ?f
proof (intro arr-eqI)
  show par: par (?f · succ) (S · ?f)
    using assms(2) f by fastforce
  show Fun (?f · succ) = Fun (S · ?f)
  proof –
    have Fun (?f · succ) = Fun ?f ∘ Fun succ
      using Fun-comp par by blast
    also have ... = Fun S ∘ Fun ?f
    proof
      fix x
      show (Fun ?f ∘ Fun succ) x = (Fun S ∘ Fun ?f) x
```

```
        proof (cases x ∈ Set N)
          case False
          show ?thesis
            using False f Fun-def by auto
          next
          case True
          have (Fun ?f ∘ Fun succ) x = ?F (succ · x)
            using True f app-mkarr [of N a - succ · x] Fun-def by auto
          also have ... = ((·) S ^^ OUT UNIV (succ · x)) Z
            using True f by auto
          also have ... = ((·) S ^^ Suc (OUT UNIV x)) Z
            by (metis (no-types, lifting) Fun-def Fun-succ True UNIV-I bij-betw-def
                bij-betw-inv-into-left ide-nat(2,3) mem-Collect-eq rangeI succ-simps(2))
          also have ... = S · ((·) S ^^ OUT UNIV x) Z
            by auto
          also have ... = S · ?F x
            using True by auto
          also have ... = S · Fun ?f x
            using f by auto
          also have ... = Fun S (Fun ?f x)
            by (metis (no-types, lifting) CollectD CollectI Fun-def dom-comp in-homE
                in-homI ext null-is-zero(2) seqE)
          also have ... = (Fun S ∘ Fun ?f) x
            by simp
          finally show ?thesis by blast
        qed
      qed
      also have ... = Fun (S · ?f)
        using Fun-comp par by presburger
      finally show ?thesis by blast
    qed
  qed
qed
moreover have ⋀f'. «f' : N → a» ∧ f' · zero = Z ∧ f' · succ = S · f' ⟶ f' = ?f
proof (intro impI arr-eqI)
  fix f'
  assume f': «f' : N → a» ∧ f' · zero = Z ∧ f' · succ = S · f'
  show par: par f' ?f
    using f f' by fastforce
  have *: ⋀k. ((·) S ^^ k) Z = Fun f' (IN UNIV k)
  proof -
    fix k
    show ((·) S ^^ k) Z = Fun f' (IN UNIV k)
    proof (induct k)
      show ((·) S ^^ 0) Z = Fun f' (IN (UNIV :: nat set) 0)
        using f' app-mkarr
        unfolding zero-def
        by (metis (no-types, lifting) CollectI Fun-zero comp-arr-dom f' funpow-0
            ide-in-hom ide-some-terminal in-homE zero-in-hom Fun-def)
        fix k
        assume ind: ((·) S ^^ k) Z = Fun f′ (IN UNIV k)
        have Fun f′ (IN UNIV (Suc k)) = Fun f′ (succ · IN UNIV k)
        proof −
          have ⋀n. OUT UNIV (IN UNIV (n::nat)) = n
            by (metis (no-types) bij-betw-inv-into-right ide-nat(2) iso-tuple-UNIV-I)
          thus ?thesis
            by (metis (no-types) Fun-def Fun-succ bij-betwE ide-nat(3) iso-tuple-UNIV-I
                succ-simps(2))
        qed
        also have ... = f′ · succ · IN UNIV k
          using bij-betwE f′ ide-nat(3) Fun-def by fastforce
        also have ... = (f′ · succ) · IN UNIV k
          using comp-assoc by simp
        also have ... = S · Fun f′ (IN UNIV k)
          using f′ bij-betw-apply ide-nat(3) comp-assoc Fun-def by fastforce
        also have ... = S · ((·) S ^^ k) Z
          using ind by simp
        also have ... = ((·) S ^^ Suc k) Z
          by auto
        finally show ((·) S ^^ Suc k) Z = Fun f′ (IN UNIV (Suc k))
          by simp
      qed
    qed
    show Fun f′ = Fun ?f
    proof
      fix x
      show Fun f′ x = Fun ?f x
      proof (cases x ∈ Set N)
        case False
        show ?thesis
          using False par Fun-def by auto
        next
        case True
        have Fun ?f x = ((·) S ^^ OUT UNIV x) Z
          using True app-mkarr f par by force
        also have ... = Fun f′ (IN (UNIV :: nat set) (OUT UNIV x))
          using ∗ by simp
        also have ... = Fun f′ x
          using True IN-OUT small-nat embeds-nat by metis
        finally show ?thesis by simp
      qed
    qed
  qed
  ultimately show ?thesis by auto
qed

lemma has-natural-numbers-object:
shows ∃ a z s. «z : 1? → a» ∧ «s : a → a» ∧
```

```
        (∀ a' z' s'. «z' : 1? → a'» ∧ «s' : a' → a'» ⟶
                (∃!f. «f : a → a'» ∧ f · z = z' ∧ f · s = s' · f))
  proof –
    have «zero : 1? → nat» ∧ «succ : nat → nat» ∧
        (∀ a' z' s'. «z' : 1? → a'» ∧ «s' : a' → a'» ⟶
                (∃!f. «f : nat → a'» ∧ f · zero = z' ∧ f · succ = s' · f))
      using nat-universality by auto
    thus ?thesis by auto
  qed

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 ..

  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*›
  by *unfold-locales* (*meson finite-surj lepoll-iff*)

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*
  by (*metis eqpoll-finite-iff eqpoll-iff-finite-card local.small-def*)

```
interpretation small-finite ‹finite :: nat set ⇒ bool›
  by unfold-locales blast

lemma small-finite-finset:
shows small-finite (finite :: nat set ⇒ bool)
  ..

interpretation small-product ‹finite :: nat set ⇒ bool›
  using eqpoll-iff-finite-card by unfold-locales auto

lemma small-product-finset:
shows small-product (finite :: nat set ⇒ bool)
  ..

interpretation small-sum ‹finite :: nat set ⇒ bool›
  by unfold-locales (meson eqpoll-iff-finite-card finite-SigmaI finite-lessThan)

lemma small-sum-finset:
shows small-sum (finite :: nat set ⇒ bool)
  ..

interpretation small-powerset ‹finite :: nat set ⇒ bool›
  using eqpoll-iff-finite-card by unfold-locales blast

lemma small-powerset-finset:
shows small-powerset (finite :: nat set ⇒ bool)
  ..

interpretation small-funcset ‹finite :: nat set ⇒ bool› ..
```

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

```
lemma large-nat-finset:
shows ¬ local.small (UNIV :: nat set)
  using finset-small-iff-finite large-UNIV by blast
```

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

```
interpretation embedding ‹UNIV :: nat set› .

interpretation lifting ‹UNIV :: nat set›
  by unfold-locales blast

lemma nat-admits-lifting:
shows lifting (UNIV :: nat set)
  ..

interpretation pairing ‹UNIV :: nat set›
  by unfold-locales blast
  lemma nat-admits-pairing:
  shows pairing (UNIV :: nat set)
    ..

  interpretation powering ‹finite :: nat set ⇒ bool› ‹UNIV :: nat set›
    using inj-on-set-encode small-iff-sml
    by unfold-locales auto

  lemma nat-admits-finite-powering:
  shows powering (finite :: nat set ⇒ bool) (UNIV :: nat set)
    ..

  interpretation tupling ‹finite :: nat set ⇒ bool› ‹UNIV :: nat set› ..

  lemma nat-admits-finite-tupling:
  shows tupling (finite :: nat set ⇒ bool) (UNIV :: nat set)
    ..

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
  using nat-admits-finite-tupling by blast
```

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›
    using lepoll-small by unfold-locales blast
```

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
    by (metis eqpoll-sym local.small-def small-eqpoll small-iff)

  interpretation small-finite ‹ZFC-in-HOL.small :: V set ⇒ bool›
    by unfold-locales
```

(*meson eqpoll-sym finite-atLeastAtMost finite-imp-small small-elts small-eqpoll*)

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

interpretation *small-product* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
by *unfold-locales* (*metis eqpoll-sym small-Times small-elts small-eqpoll*)

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

interpretation *small-sum* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
by *unfold-locales* (*meson eqpoll-sym small-Sigma small-elts small-eqpoll*)

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

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 −
obtain *f v* **where** *f*: *inj-on f X* ∧ *f* ‘ *X* = *elts v*
using *assms imageE ZFC-in-HOL.small-def* **by** *meson*
obtain *f'* **where** *f'*: *inj-on f'* (*Pow X*) ∧ *f'* ‘ (*Pow X*) = *Pow* (*elts v*)
using *f image-Pow-surj inj-on-image-Pow* **by** *metis*
have *ZFC-in-HOL.small* (*f'* ‘ (*Pow X*))
using *assms f' ZFC-in-HOL.small-image-iff* [*of f' Pow X*]
by (*metis Pow-iff down elts-VPow inj-onCI inj-on-image-eqpoll-self set-injective small-eqpoll*)
moreover have *eqpoll* (*f'* ‘ (*Pow X*)) (*Pow X*)
using *f' eqpoll-sym inj-on-image-eqpoll-self* **by** *meson*
ultimately show *ZFC-in-HOL.small* (*Pow X*)
by (*metis image-iff inj-on-image-eqpoll-1 ZFC-in-HOL.small-def small-eqpoll*)
qed

interpretation *small-powerset* ‹*ZFC-in-HOL.small* :: *V set* ⇒ *bool*›
by *unfold-locales*
(*meson eqpoll-sym gcard-eqpoll small-iff ZFC-small-implies-small-powerset*)

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

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

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

interpretation small-nat ‹ZFC-in-HOL.small :: V set ⇒ bool›
proof −
  have ZFC-in-HOL.small (UNIV :: nat set)
    using small-image-nat by (metis surj-id)
  thus small-nat (ZFC-in-HOL.small :: V set ⇒ bool)
    using gcard-eqpoll by unfold-locales auto
qed

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

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

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

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

```
interpretation embedding ‹UNIV :: V set› .

interpretation lifting ‹UNIV :: V set›
proof
  let ?ι = λ None ⇒ ZFC-in-HOL.set {}
          | Some x ⇒ ZFC-in-HOL.set {x}
  have is-embedding-of ?ι ({None} ∪ Some ‘ UNIV)
  proof
    show ?ι ‘ ({None} ∪ Some ‘ UNIV) ⊆ UNIV by blast
    show inj-on ?ι ({None} ∪ Some ‘ UNIV)
    proof
      fix x y
      assume x: x ∈ {None :: V option} ∪ Some ‘ UNIV
      assume y: y ∈ {None :: V option} ∪ Some ‘ UNIV
      assume eq: ?ι x = ?ι y
      show x = y
        by (metis (no-types, lifting) elts-of-set eq insert-not-empty option.case-eq-if
            option.collapse range-constant singleton-eq-iff small-image-nat)
    qed
  qed
  thus ∃ι :: V option ⇒ V. is-embedding-of ι ({None} ∪ Some ‘ UNIV)
    by blast
qed

lemma V-admits-lifting:
  shows lifting (UNIV :: V set)
   ..

  interpretation pairing ‹UNIV :: V set›
  proof
   show ∃ι :: V × V ⇒ V. is-embedding-of ι (UNIV × UNIV)
    using inj-on-vpair by blast
  qed

  lemma V-admits-pairing:
  shows pairing (UNIV :: V set)
   ..

  interpretation powering ‹ZFC-in-HOL.small :: V set => bool› ‹UNIV :: V set›
  proof
   show ∃ι :: V set ⇒ V. is-embedding-of ι {X. X ⊆ UNIV ∧ local.small X}
    using inj-on-set small-iff-sml by auto
  qed

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

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

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

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

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

 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
  using V-is-universe by blast

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 ≡ UNIV :: nat set

  abbreviation HOM
  where HOM ≡ λm n. {1..m :: nat} →E {1..n :: nat}

  abbreviation Id
  where Id n ≡ λx :: nat. if x ∈ {1..n} then x else undefined

  abbreviation Comp
  where Comp - - m ≡ compose {1..m}
```

```
interpretation Fin: concrete-category OBJ HOM Id Comp
  by unfold-locales fastforce+

abbreviation comp
where comp ≡ Fin.COMP

lemma terminal-MkIde-1:
shows Fin.terminal (Fin.MkIde 1)
proof
  show 1: Fin.ide (Fin.MkIde 1)
    using Fin.ide-MkIde by blast
  show ⋀a. Fin.ide a ⟹ ∃!f. Fin.in-hom f a (Fin.MkIde 1)
  proof −
    fix a
    assume a: Fin.ide a
    let ?Ta = λx. if x ∈ {1..Fin.Dom a} then 1 else undefined
    have 2: HOM (Fin.Dom a) 1 = {?Ta}
      by (cases Fin.Dom a = 0) auto
    have Fin.hom a (Fin.MkIde 1) = {Fin.MkArr (Fin.Dom a) 1 ?Ta}
    proof
      show {Fin.MkArr (Fin.Dom a) 1 ?Ta} ⊆ Fin.hom a (Fin.MkIde 1)
        using a 1 2 Fin.bij-betw-hom-Hom [of a Fin.MkIde 1] by fastforce
      show Fin.hom a (Fin.MkIde 1) ⊆ {Fin.MkArr (Fin.Dom a) 1 ?Ta}
        using a 1 2 Fin.bij-betw-hom-Hom(1−4) [of a Fin.MkIde 1]
        by auto[1] (simp add: Pi-iff)
    qed
    thus ∃!f. Fin.in-hom f a (Fin.MkIde 1)
      by (metis (no-types, lifting) mem-Collect-eq singleton-iff)
  qed
qed

sublocale category-with-terminal-object comp
  using terminal-MkIde-1
  by unfold-locales auto

notation some-terminal  (1^?)

sublocale sets-cat-base ‹finite :: nat set ⇒ bool› comp
  by (unfold-locales) (meson finite-surj lepoll-iff)

sublocale small-finite ‹finite :: nat set ⇒ bool›
  using Universe-Interps.small-finite-finset by blast

sublocale small-powerset ‹finite :: nat set ⇒ bool›
  using small-powerset-finset by auto

lemma finite-HOM:
shows finite (HOM m n)
```

```
  by (simp add: finite-PiE)

lemma card-HOM:
shows card (HOM m n) = n ^ m
  by (simp add: card-funcsetE)

lemma terminal-char_FSC:
shows Fin.terminal a ⟷ a = Fin.MkIde 1
proof
  show a = Fin.MkIde 1 ⟹ Fin.terminal a
    using terminal-MkIde-1 by blast
  assume a: Fin.terminal a
  have a = Fin.MkIde (Fin.Dom a)
    using a Fin.terminal-def Fin.MkIde-Dom' by auto
  moreover have Fin.Dom a = 1
  proof −
    have Fin.Dom a ≠ 1 ⟹ ¬ (∃!f. Fin.in-hom f a (Fin.MkIde 1))
    proof −
      assume 1: Fin.Dom a ≠ 1
      have card (HOM 1 (Fin.Dom a)) ≠ 1
        using 1 card-HOM
        by (metis power-one-right)
      moreover have card (HOM 1 (Fin.Dom a)) = card (Fin.hom (Fin.MkIde 1) a)
        by (metis (no-types, lifting) HOL.ext Fin.Dom.simps(1) a Fin.bij-betw-hom-Hom(5)
          bij-betw-same-card terminal-MkIde-1 Fin.terminal-def)
      moreover have ⋀A. (∃!x. x ∈ A) ⟷ card A = 1
        by (metis card-1-singletonE ex-in-conv insert-iff is-singletonI' is-singleton-altdef)
      ultimately show ¬ (∃!f. Fin.in-hom f a (Fin.MkIde 1))
        by (metis (no-types, lifting) a mem-Collect-eq terminal-MkIde-1 Fin.terminal-def)
    qed
    thus ?thesis
      using a Fin.terminal-def terminal-MkIde-1 by force
  qed
  ultimately show a = Fin.MkIde 1 by auto
qed

lemma MkIde-1-eq:
shows Fin.MkIde 1 = 1^?
  using terminal-char_FSC terminal-some-terminal by presburger

lemma finite-Set:
assumes Fin.ide a
shows finite (Set a)
  by (metis assms bij-betw-finite Fin.bij-betw-hom-Hom(5) finite-HOM ide-some-terminal)

lemma card-Set:
assumes Fin.ide a
shows card (Set a) = Fin.Dom a
proof −
```

```
  have Set a = Fin.hom (Fin.MkIde 1) a
    using assms MkIde-1-eq by presburger
  moreover have eqpoll (Fin.hom (Fin.MkIde 1) a) (HOM 1 (Fin.Dom a))
    using assms Fin.bij-betw-hom-Hom(5)[of Fin.MkIde 1 a] eqpoll-def
          MkIde-1-eq ide-some-terminal
    by auto
  moreover have card (HOM 1 (Fin.Dom a)) = Fin.Dom a
    using card-HOM
    by (metis power-one-right)
  ultimately show ?thesis
    by (metis (lifting) bij-betw-same-card eqpoll-def)
qed

abbreviation mkpoint
where mkpoint n k ≡ Fin.MkArr 1 n (λx. 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)
  using assms Fin.MkArr-in-hom [of 1 n - Fin.MkIde 1 Fin.MkIde n] by fastforce

lemma valof-in-range:
assumes Fin.in-hom x 1^? a
shows valof x ∈ {1..Fin.Dom a}
  using assms Fin.arr-char [of x] Fin.dom-char Fin.cod-char
  by (metis (no-types, lifting) Fin.Dom.simps(1) MkIde-1-eq PiE-E atLeastAtMost-singleton'
    Fin.in-hom-char singletonI)

lemma valof-mkpoint:
shows valof (mkpoint n k) = k
  by force

lemma mkpoint-valof:
assumes Fin.in-hom x 1^? a
shows mkpoint (Fin.Dom a) (valof x) = x
proof (intro Fin.arr-eqI)
  show Fin.arr (mkpoint (Fin.Dom a) (valof x))
    using assms mkpoint-in-hom valof-in-range by blast
  show 1: Fin.arr x
    using assms by blast
  show 2: Fin.Dom (mkpoint (Fin.Dom a) (valof x)) = Fin.Dom x
    by (metis (lifting) Fin.Dom.simps(1) MkIde-1-eq assms Fin.in-hom-char)
  show Fin.Cod (mkpoint (Fin.Dom a) (valof x)) = Fin.Cod x
    by (metis (lifting) Fin.Cod.simps(1) MkIde-1-eq assms Fin.in-hom-char)
  show Fin.Map (mkpoint (Fin.Dom a) (valof x)) = Fin.Map x
  proof -
```

```
    have Fin.Map (mkpoint (Fin.Dom a) (valof x)) =
          (λk. if k = 1 then valof x else undefined)
      by simp
    also have ... = Fin.Map x
    proof
      fix k
      show (if k = 1 then valof x else undefined) = Fin.Map x k
        using 1 2 Fin.arr-char by auto
    qed
    finally show ?thesis by blast
  qed
qed

lemma Map-arr-eq:
assumes Fin.in-hom f a b
shows Fin.Map f = (λk. 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
  fix k
  show Fin.Map f k = ?F k
  proof (cases k ∈ {1..Fin.Dom a})
    case False
    show ?thesis using False
      by (metis (no-types, lifting) Fin.Map-in-Hom PiE-arb assms Fin.in-hom-char)
    next
    case True
    have ?F k = Fin.Map (Fun f (mkpoint (Fin.Dom a) k)) 1
      using True by simp
    also have ... = Fin.Map (comp f (mkpoint (Fin.Dom a) k)) 1
      using assms True mkpoint-in-hom [of k Fin.Dom a] MkIde-1-eq Fin.in-homE
            Fin.in-hom-char Fun-def
      by auto
    also have ... = Fin.Map f (Fin.Map (mkpoint (Fin.Dom a) k) (1 :: nat))
      using assms True mkpoint-in-hom Fin.in-hom-char Fin.Map-comp by auto
    also have ... = Fin.Map f k
      by force
    finally show ?thesis by simp
  qed
qed

sublocale sets-cat ‹finite :: nat set ⇒ bool› comp
proof
  show ⋀a. Fin.ide a ⟹ nat-tupling.small (Set a)
    using finite-Set finset-small-iff-finite by blast
  show ⋀A. ⟦nat-tupling.small A; A ⊆ Collect Fin.arr⟧ ⟹ ∃ a. Fin.ide a ∧ Set a ≈ A
    by (metis (no-types, lifting) Fin.Dom.simps(1) card-Set eqpoll-iff-card finite-Set
        finset-small-iff-finite Fin.ide-MkIde iso-tuple-UNIV-I)
```

```
show ⋀a b. ⟦Fin.ide a; Fin.ide b⟧ ⟹ inj-on Fun (Fin.hom a b)
  using Map-arr-eq Fin.in-hom-char
  by (intro inj-onI Fin.arr-eqI) auto
show ⋀a b. ⟦Fin.ide a; Fin.ide b⟧ ⟹ Hom a b ⊆ Fun ‘ Fin.hom a b
proof
  fix a b
  assume a: Fin.ide a and b: Fin.ide b
  fix F
  assume F: F ∈ Hom a b
  show F ∈ Fun ‘ Fin.hom a b
  proof
    let ?F′ = λk. if k ∈ {1..Fin.Dom a}
                  then valof (F (mkpoint (Fin.Dom a) k))
                  else undefined
    let ?f = Fin.MkArr (Fin.Dom a) (Fin.Dom b) ?F′
    show f: ?f ∈ Fin.hom a b
    proof
      show Fin.in-hom ?f a b
      proof
        show Fin.Dom a ∈ UNIV by auto
        show Fin.Dom b ∈ UNIV by auto
        show a = Fin.MkIde (Fin.Dom a)
          using a Fin.MkIde-Dom′ by presburger
        show b = Fin.MkIde (Fin.Dom b)
          using b Fin.MkIde-Dom′ by presburger
        show ?F′ ∈ HOM (Fin.Dom a) (Fin.Dom b)
        proof
          fix k
          show k ∉ {1..Fin.Dom a} ⟹ ?F′ k = undefined by auto
          show k ∈ {1..Fin.Dom a} ⟹ ?F′ k ∈ {1..Fin.Dom b}
          proof −
            assume k: k ∈ {1..Fin.Dom a}
            have ?F′ k = valof (F (mkpoint (Fin.Dom a) k))
              using k by simp
            moreover have ... ∈ {1..Fin.Dom b}
            proof −
              have F (mkpoint (Fin.Dom a) k) ∈ Fin.hom 1? b
                using a k F mkpoint-in-hom MkIde-1-eq ‹a = Fin.MkIde (Fin.Dom a)›
                by force
              thus ?thesis
                using valof-in-range by blast
            qed
            ultimately show ?thesis by auto
          qed
        qed
      qed
    qed
    show F = Fun ?f
    proof
```

```
fix x
show F x = Fun ?f x
proof (cases x ∈ Fin.hom 1^? a)
  case False
  show ?thesis
    using False F f a Fin.dom-eqI Fin.ide-in-hom Fin.seqI' Fun-def by auto
  next
  case True
  show ?thesis
  proof (intro Fin.arr-eqI)
    show 1: Fin.arr (F x)
      using F True by blast
    show 2: Fin.arr (Fun ?f x)
      using f True a Fin.dom-eqI Fin.ide-in-hom Fin.seqI' Fun-def by auto
    show Fin.Dom (F x) = Fin.Dom (Fun ?f x)
    proof −
      have Fin.Dom (F x) = Fin.Dom 1^?
        using F True
        by (metis (no-types, lifting) Int-def Pi-iff Fin.in-hom-char mem-Collect-eq)
      also have ... = Fin.Dom (Fun ?f x)
        using True f
        by (metis (no-types, lifting) 2 Fin.Dom-comp Fun-def Fin.arrE
            Fin.in-hom-char mem-Collect-eq Fin.null-char)
      finally show ?thesis by blast
    qed
    show Fin.Cod (F x) = Fin.Cod (Fun ?f x)
    proof −
      have Fin.Cod (F x) = Fin.Dom b
        using F True
        by (metis (no-types, lifting) Int-def Pi-mem Fin.in-hom-char mem-Collect-eq)
      also have ... = Fin.Cod (Fun ?f x)
        using True f 2
        by (metis (no-types, lifting) Fin.Cod.simps(1) Fin.Cod-comp Fin.arrE
            Fin.null-char Fin.seq-char Fun-def)
      finally show ?thesis by blast
    qed
    show Fin.Map (F x) = Fin.Map (Fun ?f x)
    proof
      fix k
      show Fin.Map (F x) k = Fin.Map (Fun ?f x) k
      proof −
        have k ≠ 1 ⟹ ?thesis
        proof −
          assume k: k ≠ 1
          have 1: Fin.Map (F x) k = undefined
          proof −
            have Fin.in-hom (F x) 1^? b
              using F True by blast
            thus ?thesis
```

```
                using F True k Map-arr-eq [of F x 1^? b]
                by (metis Fin.Dom.simps(1) MkIde-1-eq atLeastAtMost-iff le-antisym)
            qed
            also have ... = Fin.Map (Fun ?f x) k
            proof -
              have Fin.Map (Fun ?f x) k = Fin.Map (comp ?f x) k
                using f True Fun-def by fastforce
              also have ... = compose {1..Fin.Dom x} (Fin.Map ?f) (Fin.Map x) k
                using f True Fin.Map-comp
                by (metis (no-types, lifting) Fin.in-hom-char mem-Collect-eq)
              also have ... = undefined
              proof -
                have k ∉ {1..Fin.Dom x}
                  using True k
                  by (metis (no-types, lifting) Fin.Dom.simps(1) MkIde-1-eq
                      atLeastAtMost-singleton Fin.in-hom-char mem-Collect-eq
                      singleton-iff)
                thus ?thesis by auto
              qed
              finally show ?thesis by simp
            qed
            finally show ?thesis by simp
          qed
          moreover have k = 1 ⟹ ?thesis
          proof -
            assume k: k = 1
            have Fin.Map (Fun ?f x) k = Fin.Map (comp ?f x) k
              using 2 Fun-def Fin.arrE Fin.null-char by fastforce
            also have ... = compose {1..1} (Fin.Map ?f) (Fin.Map x) k
              using f True Fin.Map-comp
              by (metis (lifting) Fin.Dom.simps(1) IntI Int-Collect MkIde-1-eq
                  Fin.in-hom-char)
            also have ... = ?F' (Fin.Map x k)
              apply auto[1]
              by (auto simp add: k)
            also have ... = valof (F (mkpoint (Fin.Dom a) (Fin.Map x k)))
              using F True k a valof-in-range by auto
            also have ... = valof (F x)
              using F True k mkpoint-valof by force
            also have ... = Fin.Map (F x) k
              using F True k by argo
            finally show ?thesis by simp
          qed
          ultimately show ?thesis by blast
        qed
      qed
    qed
  qed
qed
```

```
    qed
  qed
qed

lemma is-sets-cat:
shows sets-cat (finite :: nat set ⇒ bool) comp
  ..

sublocale small-product ‹finite :: nat set ⇒ bool›
  using small-product-finset by blast

sublocale sets-cat-with-pairing ‹finite :: nat set ⇒ bool› comp
proof
  show ∃ι. is-embedding-of ι (Collect Fin.arr × Collect Fin.arr)
  proof -
    have ⋀A. ⟦countable A; infinite A⟧ ⟹ ∃ι. ι ‘ (A × A) ⊆ A ∧ inj-on ι (A × A)
    proof -
      fix A :: 'a set
      assume countable: countable A and infinite: infinite A
      obtain ϱ where ϱ: bij-betw ϱ (A × A) (UNIV :: nat set)
        using countable infinite countableE-infinite
        by (metis countable-SIGMA infinite-cartesian-product)
      obtain σ where σ: bij-betw σ (UNIV :: nat set) A
        using countable infinite bij-betw-from-nat-into by blast
      have (σ ∘ ϱ) ‘ (A × A) ⊆ A ∧ inj-on (σ ∘ ϱ) (A × A)
        using ϱ σ
        by (metis bij-betw-def comp-inj-on-iff equalityD2 image-comp)
      thus ∃ι. ι ‘ (A × A) ⊆ A ∧ inj-on ι (A × A) by blast
    qed
    moreover have countable (Collect Fin.arr) ∧ infinite (Collect Fin.arr)
    proof
      show countable (Collect Fin.arr)
      proof -
        have Collect Fin.arr =
              (⋃ab∈Collect Fin.ide × Collect Fin.ide. Fin.hom (fst ab) (snd ab))
        proof
          show (⋃ab∈Collect Fin.ide × Collect Fin.ide. Fin.hom (fst ab) (snd ab)) ⊆
                Collect Fin.arr
            by blast
          show Collect Fin.arr ⊆
                (⋃ab∈Collect Fin.ide × Collect Fin.ide. Fin.hom (fst ab) (snd ab))
          proof
            fix f
            assume f: f ∈ Collect Fin.arr
            have Fin.ide (Fin.dom f) ∧ Fin.ide (Fin.cod f) ∧
                  f ∈ Fin.hom (Fin.dom f) (Fin.cod f)
              using f Fin.ide-dom Fin.ide-cod by blast
            hence (Fin.dom f, Fin.cod f) ∈ Collect Fin.ide × Collect Fin.ide ∧
                  f ∈ Fin.hom (fst (Fin.dom f, Fin.cod f)) (snd (Fin.dom f, Fin.cod f))
          by auto
        thus f ∈ (⋃ab∈Collect Fin.ide × Collect Fin.ide. Fin.hom (fst ab) (snd ab))
          by blast
      qed
    qed
    moreover have countable (Collect Fin.ide × Collect Fin.ide)
      using Fin.bij-betw-ide-Obj(5) by force
    moreover have ⋀ab. ab ∈ Collect Fin.ide × Collect Fin.ide
                      ⟹ finite (Fin.hom (fst ab) (snd ab)) ∧
                         card (Fin.hom (fst ab) (snd ab)) =
                         Fin.Dom (snd ab) ^ Fin.Dom (fst ab)
      by (metis bij-betw-finite Fin.bij-betw-hom-Hom(5) bij-betw-same-card card-HOM
        finite-HOM mem-Collect-eq mem-Times-iff)
    ultimately show ?thesis
      using countable-UN countable-finite by (metis (lifting))
  qed
  show infinite (Collect Fin.arr)
  proof -
    have ⋀X. ∀n. (∃Y. Y ⊆ X ∧ card Y ≥ n) ⟹ infinite X
      by (metis card-mono not-less-eq-eq)
    moreover have ∀n. (∃ab. ab ∈ Collect Fin.ide × Collect Fin.ide ∧
                       card (Fin.hom (fst ab) (snd ab)) ≥ n)
      by (metis (no-types, lifting) HOL.ext Fin.Dom.simps(1) SigmaI card-Set
          fst-conv Fin.ide-MkIde ide-some-terminal iso-tuple-UNIV-I mem-Collect-eq
        order-refl snd-conv)
    ultimately show ?thesis
      by (metis (no-types, lifting) Fin.in-homE mem-Collect-eq subsetI)
  qed
 qed
 ultimately show ?thesis by blast
qed
qed

lemma is-sets-cat-with-pairing:
shows sets-cat-with-pairing (finite :: nat set ⇒ bool) comp
  ..

sublocale lifting ‹Collect Fin.arr›
proof
  show embeds ({None} ∪ Some ‘ Collect Fin.arr)
  proof -
    have ⋀n :: nat. Set (Fin.MkIde n) ⊆ Collect Fin.arr ∧ card (Set (Fin.MkIde n)) = n
      using card-Set Fin.ide-MkIde by fastforce
    hence 1: infinite (Collect Fin.arr)
      by (metis (lifting) Suc-n-not-le-n card-mono)
    obtain a where a: a ∈ Collect Fin.arr
      using 1 not-finite-existsD by auto
    have 2: eqpoll (Collect Fin.arr) (Collect Fin.arr − {a})
      using 1 a
```

```
        by (metis (lifting) infinite-insert-eqpoll infinite-remove insert-Diff)
      obtain f where f: f ‘ Collect Fin.arr ⊆ Collect Fin.arr − {a} ∧
                        inj-on f (Collect Fin.arr)
        using 2
        by (metis (lifting) bij-betw-def eqpoll-def subset-refl)
      let ?ι = λNone ⇒ a | Some x ⇒ f x
      have is-embedding-of ?ι ({None} ∪ Some ‘ Collect Fin.arr)
        using a f by (auto simp add: inj-on-def)
      thus ?thesis by blast
    qed
  qed

  sublocale sets-cat-with-powering ‹finite :: nat set ⇒ bool› comp
  proof
    show embeds {X. X ⊆ Collect Fin.arr ∧ nat-tupling.small X}
    proof −
      have ⋀X. infinite X ⟹ eqpoll (Fpow X) X
        using Fpow-infinite-bij-betw eqpoll-def by blast
      hence eqpoll {X. X ⊆ Collect Fin.arr ∧ nat-tupling.small X} (Collect Fin.arr)
        using infinite-univ finset-small-iff-finite Fpow-def
        by (metis (mono-tags, lifting) Collect-cong)
      thus ?thesis
        by (metis (lifting) bij-betw-def eqpoll-def subset-refl)
    qed
  qed

  lemma is-sets-cat-with-powering:
  shows sets-cat-with-powering (finite :: nat set ⇒ bool) comp
    ..

  sublocale small-sum ‹finite :: nat set ⇒ bool›
    using small-sum-finset by blast

  sublocale sets-cat-with-tupling ‹finite :: nat set ⇒ bool› comp
    by unfold-locales

  theorem is-sets-cat-with-tupling:
  shows sets-cat-with-tupling (finite :: nat set ⇒ bool) comp
    ..

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* **.**

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 .
```

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

```
sublocale smallness ‹ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool›
  using lepoll-small by unfold-locales blast

sublocale sets-cat-base ‹ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool› ZFC.comp
  using ZFC.terminal-unity_SC by unfold-locales blast

sublocale sets-cat ‹ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool› ZFC.comp
proof
  show ⋀a. ZFC.ide a ⟹ ZFC-universe.small (Set a)
    unfolding ZFC-universe.small-def
    using ZFC.ide-char_SSC ZFC.setp-def ZFC.small-hom
    by (meson eqpoll-sym small-elts small-eqpoll)
  show ⋀A. ⟦ZFC-universe.small A; A ⊆ Collect ZFC.arr⟧ ⟹ ∃ a. ZFC.ide a ∧ Set a ≈ A
  proof -
    fix A
    assume small: ZFC-universe.small A and A: A ⊆ Collect ZFC.arr
    let ?V = λf. vpair
                 (vpair (ZFC.V-of-ide (ZFC.dom f)) (ZFC.V-of-ide (ZFC.cod f)))
                 (ZFC.V-of-arr f)
    let ?A' = ZFC.UP ` ?V ` A
    have ZFC.ide (ZFC.mkIde ?A') ∧ ZFC.set (ZFC.mkIde ?A') = ?A'
      using ZFC.ide-mkIde ZFC.setp-def
      by (metis (lifting) ZFC.set-mkIde bij-betw-imp-surj-on image-mono replacement
          replete-setcat.bij-arr-of small small-iff-ZFC-small
          subset-UNIV)
    moreover have ?A' ≈ A
    proof -
      have inj ZFC.UP
        by (simp add: ZFC.inj-UP)
      moreover have inj-on ?V (Collect ZFC.arr)
      proof (intro inj-onI)
        fix f g
        assume f: f ∈ Collect ZFC.arr and g: g ∈ Collect ZFC.arr
        assume eq: ?V f = ?V g
        have ZFC.V-of-ide (ZFC.dom f) = ZFC.V-of-ide (ZFC.dom g) ∧
              ZFC.V-of-ide (ZFC.cod f) = ZFC.V-of-ide (ZFC.cod g) ∧
              ZFC.V-of-arr f = ZFC.V-of-arr g
          using f g eq by fastforce
        thus f = g
          by (metis (lifting) ZFC-set-cat.bij-betw-hom-vfun(3) ZFC-set-cat.bij-betw-ide-V(3)
              ZFC.arr-iff-in-hom f g ZFC.ide-cod ZFC.ide-dom mem-Collect-eq)
      qed
      ultimately show ?thesis
        by (metis (no-types, lifting) A eqpoll-refl inj-on-image-eqpoll-2
            subset-UNIV inj-on-subset)
    qed
    ultimately have ZFC.ide (ZFC.mkIde ?A') ∧ Set (ZFC.mkIde ?A') ≈ A
      by (metis (no-types, lifting) HOL.ext some-terminal-def ZFC.bij-betw-points-and-set
          eqpoll-def ZFC.unity-def eqpoll-trans)
    thus ∃ a. ZFC.ide a ∧ Set a ≈ A by blast
  qed
  show ⋀a b. ⟦ZFC.ide a; ZFC.ide b⟧ ⟹ inj-on Fun (ZFC.hom a b)
  proof −
    fix a b
    assume a: ZFC.ide a and b: ZFC.ide b
    show inj-on Fun (ZFC.hom a b)
    proof
      fix f g
      assume f: f ∈ ZFC.hom a b and g: g ∈ ZFC.hom a b
      assume eq: Fun f = Fun g
      show f = g
      proof (intro ZFC.arr-eqI'_SC [of f g])
        show par: ZFC.par f g
          using f g by blast
        show ⋀x. ZFC.in-hom x ZFC.unity (ZFC.dom f) ⟹ ZFC.comp f x = ZFC.comp g x
          by (metis (lifting) some-terminal-def Fun-def par eq mem-Collect-eq ZFC.unity-def)
      qed
    qed
  qed
  show ⋀a b. ⟦ZFC.ide a; ZFC.ide b⟧ ⟹ Hom a b ⊆ Fun ‘ ZFC.hom a b
  proof
    fix a b
    assume a: ZFC.ide a and b: ZFC.ide b
    fix F
    assume F: F ∈ Hom a b
    let ?f = ZFC.mkArr' a b F
    have f: ?f ∈ ZFC.hom a b
      using a b F ZFC.mkArr'-in-hom ZFC.unity-def some-terminal-def by force
    moreover have Fun ?f = F
    proof
```

```
      fix x
      show Fun ?f x = F x
      proof (cases x ∈ Set a)
        case False
        show ?thesis
        proof −
          have Fun ?f x = ZFC.null
            unfolding Fun-def
            using f False ZFC.in-homE by fastforce
          also have ... = F x
            using False a F by auto
          finally show ?thesis by blast
        qed
        next
        case True
        show ?thesis
        proof −
          have ZFC.dom ?f = a
            using f by blast
          thus ?thesis
            unfolding Fun-def
            using a b f F True ZFC.comp-point-mkArr' ZFC.unity-def some-terminal-def
            by force
        qed
      qed
    qed
    ultimately have ∃f. f ∈ ZFC.hom a b ∧ Fun f = F by blast
    thus F ∈ Fun ‘ ZFC.hom a b by blast
  qed
qed

lemma is-sets-cat:
shows sets-cat (ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool) ZFC.comp
  ..
```

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 (intro inj-onI)
  fix f g
  assume f: f ∈ Collect ZFC.arr and g: g ∈ Collect ZFC.arr
  assume eq: arr-to-V f = arr-to-V g
  have ZFC.V-of-ide (ZFC.dom f) = ZFC.V-of-ide (ZFC.dom g) ∧
        ZFC.V-of-ide (ZFC.cod f) = ZFC.V-of-ide (ZFC.cod g) ∧
        ZFC.V-of-arr f = ZFC.V-of-arr g
      using f g eq by fastforce
    thus f = g
      by (metis (lifting) ZFC-set-cat.bij-betw-hom-vfun(3) ZFC-set-cat.bij-betw-ide-V(3)
          ZFC.arr-iff-in-hom f g ZFC.ide-cod ZFC.ide-dom mem-Collect-eq)
  qed
```

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 −
    have inj-on arr-to-V (Collect ZFC.arr)
      using inj-arr-to-V by blast
    moreover have ZFC.ide-of-V ∈ UNIV → Collect ZFC.arr ∧ inj ZFC.ide-of-V
      by (metis (no-types, lifting) Pi-iff ZFC-set-cat.bij-betw-ide-V(6) bij-betw-def
          ZFC.ide-char imageI mem-Collect-eq)
    ultimately show 1: Collect ZFC.arr ≈ (UNIV :: V set)
      using Schroeder-Bernstein [of arr-to-V Collect ZFC.arr UNIV ZFC.ide-of-V ]
      by (simp add: Pi-iff eqpoll-def image-subset-iff)
    moreover have Collect ZFC.arr ∪ {ZFC.null} ≈ Collect ZFC.arr
    proof −
      have ⋀X a. infinite X ⟹ insert a X ≈ X
        by (simp add: infinite-insert-eqpoll)
      moreover have infinite (Collect ZFC.arr)
      proof −
        have ⋀X Y. X ≈ Y ⟹ infinite X ⟷ infinite Y
          using eqpoll-finite-iff by blast
        moreover have infinite (UNIV :: V set)
          using infinite-ω rev-finite-subset by blast
        ultimately show ?thesis
          using 1 by blast
      qed
      ultimately show ?thesis by fastforce
    qed
    ultimately show Collect ZFC.arr ∪ {ZFC.null} ≈ (UNIV :: V set)
      using eqpoll-trans by blast
  qed

   sublocale universe ‹ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool› ‹Collect ZFC.arr›
ZFC.null
  proof −
    interpret V: universe ‹ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool› ‹UNIV :: V set›
      using V-is-universe by blast
    show universe (ZFC-in-HOL.small :: ZFC-in-HOL.V set ⇒ bool) (Collect ZFC.arr)
      using V-is-universe eqpoll-sym V.is-respected-by-equipollence
            eqpoll-Collect-arr-V(2)
      by blast
    qed

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

    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
      ..

  end
```

Here is the final top-level interpretation.

```
  interpretation SetsCat_ZFC: ZFC-sets-cat .

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.