Category Theory with Adjunctions and Limits

Eugene W. Stark

Department of Computer Science
Stony Brook University
Stony Brook, New York 11794 USA

March 17, 2025

Abstract

This article attempts to develop a usable framework for doing category theory in Isabelle/HOL. Our point of view, which to some extent differs from that of the previous AFP articles on the subject, is to try to explore how category theory can be done efficaciously within HOL, rather than trying to match exactly the way things are done using a traditional approach. To this end, we define the notion of category in an "object-free" style, in which a category is represented by a single partial composition operation on arrows. This way of defining categories provides some advantages in the context of HOL, including the ability to avoid the use of records and the possibility of defining functors and natural transformations simply as certain functions on arrows, rather than as composite objects. We define various constructions associated with the basic notions, including: dual category, product category, functor category, discrete category, free category, functor composition, and horizontal and vertical composite of natural transformations. A "set category" locale is defined that axiomatizes the notion "category of all sets at a type and all functions between them," and a fairly extensive set of properties of set categories is derived from the locale assumptions. The notion of a set category is used to prove the Yoneda Lemma in a general setting of a category equipped with a "hom embedding," which maps arrows of the category to the "universe" of the set category. We also give a treatment of adjunctions, defining adjunctions via left and right adjoint functors, natural bijections between hom-sets, and unit and counit natural transformations, and showing the equivalence of these definitions. We also develop the theory of limits, including representations of functors, diagrams and cones, and diagonal functors. We show that right adjoint functors preserve limits, and that limits can be constructed via products and equalizers. We characterize the conditions under which limits exist in a set category. We also examine the case of limits in a functor category, ultimately culminating in a proof that the Yoneda embedding preserves limits.

Revisions made subsequent to the first version of this article added material on equivalence of categories, cartesian categories, categories with pullbacks, categories with finite limits, and cartesian closed categories. A construction was given of the category of hereditarily finite sets and functions between them, and it was shown that this category is cartesian closed.

Contents

Chapter 1

Introduction

This article attempts to develop a usable framework for doing category theory in Isabelle/HOL. Perhaps the main issue that one faces in doing this is how best to represent what is essentially a theory of a partially defined operation (composition) in HOL, which is a theory of total functions. The fact that in HOL every function is total means that a value must be given for the composition of any pair of arrows of a category, even if those arrows are not really composable. Proofs must constantly concern themselves with whether or not a particular term does or does not denote an arrow, and whether particular pairs of arrows are or are not composable. This kind of issue crops up in the most basic situations, such as trying to use associativity of composition to prove that two arrows are equal. Without some sort of systematic way of dealing with this issue, it is hard to do proofs of interesting results, because one is constantly distracted from the main line of reasoning by the necessity of proving lemmas that show that various expressions denote well-defined arrows, that various pairs of arrows are composable, *etc.*

In trying to develop category theory in this setting, one notices fairly soon that some of the problem can be solved by creating introduction rules that allow the proof assistant to automatically infer, say, that a given term denotes an arrow with a particular domain and codomain from similar properties of its proper subterms. This "upward" reasoning helps, but it goes only so far. Eventually one faces a situation in which it is desired to prove theorems whose hypotheses state that certain terms denote arrows with particular domains and codomains, but the proof requires similar lemmas about the proper subterms. Without some way of doing this "downward" reasoning, it becomes very tedious to establish the necessary lemmas.

Another issue that one faces when trying to formulate category theory within HOL is the lack of the set-theoretic universe that is usually assumed in traditional developments. Since there is no "type of all sets" in HOL, one cannot construct "the" category **Set** of *all* sets and functions between them. Instead, the best one can do is consider "a" category of all sets and functions at a particular type. Although the lack of set-theoretic universe would likely cause complications for some applications of category theory, there are many applications for which the lack of a universe is not really a hindrance. So one might well adopt a point of view that accepts *a priori* the lack of a universe and asks instead how

much of traditional category theory could be done in such a setting.

There have been two previous category theory submissions to the AFP. The first [5] is an exploratory work that develops just enough category theory to enable the statement and proof of a version of the Yoneda Lemma. The main features are: the use of records to define categories and functors, construction of a category of all subsets of a given set, where the arrows are domain set/codomain set/function triples, and the use of the category of all sets of elements of the arrow type of category C as the target for the Yoneda functor for C. The second category theory submission to the AFP [2] is somewhat more extensive in its scope, and tries to match more closely a traditional development of category theory through the use of a set-theoretic universe obtained by an axiomatic extension of HOL. Categories, functors, and natural transformations are defined as multi-component records, similarly to [5]. "The" category of sets is defined, having as its object and arrow type the type ZF, which is the axiomatically defined set-theoretic universe. Included in [2] is a more extensive development of natural transformations, vertical composition, and functor categories than is to be found in [5]. However, as in [5], the main purely category-theoretic result in [2] is the Yoneda Lemma. Beyond the use of "extensional" functions, which take on a particular default value outside of their domains of definition, neither [5] nor [2] explicitly describe a systematic approach to the problem of obtaining lemmas that establish when the various terms appearing in a proof denote well-defined arrows.

The present development differs in a number of respects from that of [5] and [2], both in style and scope. The main stylistic features of the present development are as follows:

- The notion of a category is defined in an "object-free" style, motivated by [1], Sec. 3.52-3.53, in which a category is represented by a single partial composition operation on arrows. This way of defining categories provides some advantages in the context of HOL, including the possibility of avoiding extensive use of composite objects constructed using records. (Katovsky seemed to have had some similar ideas, since he refers in [3] to a theory "PartialBinaryAlgebra" that was also motivated by [1], although this theory did not ultimately become part of his AFP article.)

- Functors and natural transformation are defined simply to be certain functions on arrows, where locale predicates are used to express the conditions that must be satisfied. This makes it possible to define functors and natural transformations easily using lambda notation without records.

- Rules for reasoning about categories, functors, and natural transformations are defined so that all "diagrammatic" hypotheses reduce to conjunctions of assertions, each of which states that a given entity is an arrow, has a particular domain or codomain, or inhabits a particular "hom-set". A system of introduction and elimination rules is established which permits both "upward" reasoning, in which such diagrammatic assertions are established for larger terms using corresponding assertions about the proper subterms, as well as "downward" reasoning, in which diagrammatic assertions about proper subterms are inferred from such assertions about a larger term, to be carried out automatically.

- Constructions on categories, functors, and natural transformations are defined using locales in a formulaic fashion. As an example, the product category construction is defined using a locale that takes two categories (given by their partial composition operations) as parameters. The partial composition operation for the product category is given by a function "*comp*" defined in the locale. Lemmas proved within the locale include the fact that *comp* indeed defines a category, as well as characterizations of the basic notions (domain, codomain, identities, composition) in terms of those of the parameter categories. For some constructions, such as the product category, it is possible and convenient to have a "transparent" arrow type, which permits reasoning about the construction without having to introduce an elaborate system of constructors, destructors, and associated rules. For other constructions, such as the functor category, it is more desirable to use an "opaque" arrow type that hides the concrete structure, and forces all reasoning to take place using a fixed set of rules.

- Rather than commit to a specific concrete construction of a category of sets and functions a "set category" locale is defined which axiomatizes the properties of the category of sets with elements at a particular type and functions between such. In keeping with the definitional approach, the axiomatization is shown consistent by exhibiting a particular interpretation for the locale, however care is taken to to ensure that any proofs making use of the interpretation depend only on the locale assumptions and not on the concrete details of the construction. The set category axioms are also shown to be categorical, in the sense that a bijection between the sets of terminal objects of two interpretations of the locale extends to an isomorphism of categories. This supports the idea that the locale axioms are an adequate characterization of the properties of a category of sets and functions and the details of a particular concrete construction can be kept hidden.

A brief synopsis of the formal mathematical content of the present development is as follows:

- Definitions are given for the notions: category, functor, and natural transformation.

- Several constructions on categories are given, including: free category, discrete category, dual category, product category, and functor category.

- Composite functor, horizontal and vertical composite of natural transformations are defined, and various properties proved.

- The notion of a "set category" is defined and a fairly extensive development of the consequences of the definition is carried out.

- Hom-functors and Yoneda functors are defined and the Yoneda Lemma is proved.

- Adjunctions are defined in several ways, including universal arrows, natural isomorphisms between hom-sets, and unit and counit natural transformations. The relationships between the definitions are established.

- The theory of limits is developed, including the notions of diagram, cone, limit cone, representable functors, products, and equalizers. It is proved that a category with products at a particular index type has limits of all diagrams at that type. The completeness properties of a set category are established. Limits in functor categories are explored, culminating in a proof that the Yoneda embedding preserves limits.

Revision Notes

The 2018 version of this development was a major revision of the original (2016) version. Although the overall organization and content remained essentially the same, the 2018 version revised the axioms used to define a category, and as a consequence many proofs required changes. The purpose of the revision was to obtain a more organized set of basic facts which, when annotated for use in automatic proof, would yield behavior more understandable than that of the original version. In particular, as I gained experience with the Isabelle simplifier, I was able to understand better how to avoid some of the vexing problems of looping simplifications that sometimes cropped up when using the original rules. The new version "feels" about as powerful as the original version, or perhaps slightly more so. However, the new version uses elimination rules in place of some things that were previously done by simplification rules, which means that from time to time it becomes necessary to provide guidance to the prover as to where the elimination rules should be invoked.

Another difference between the 2018 version of this document and the original is the introduction of some notational syntax, which I intentionally avoided in the original. An important reason for not introducing syntax in the original version was that at the time I did not have much experience with the notational features of Isabelle, and I was afraid of introducing hard-to-remove syntax that would make the development more difficult to read and write, rather than easier. (I tended to find, for example, that the proliferation of special syntax introduced in [2] made the presentation seem less readily accessible than if the syntax had been omitted.) In the 2018 revision, I introduced syntax for composition of arrows in a category, and for the notion of "an arrow inhabiting a hom-set." The notation for composition eases readability by reducing the number of required parentheses, and the notation for asserting that an arrow inhabits a particular hom-set gives these assertions a more familiar appearance; making it easier to understand them at a glance.

This document was revised again in early 2020, prior to the release of Isabelle2020. That revision incorporated the generic "concrete category" construction originally introduced in [6], and using it systematically as a uniform replacement for various constructions that were previously done in an *ad hoc* manner. These include the construction of "functor categories" of categories of functors and natural transformations, "set categories" of sets and functions, and various kinds of free categories. The awkward "abstracted category" construction, which had no interesting mathematical content but was present in the original version as a solution to a modularity problem that I no longer deem to be a significant issue, has been removed. The cumbersome "horizontal composite" locale, which was unnecessary given that in this formalization horizontal composite

is given simply by function composition, has been replaced by a single lemma that does the same job. Finally, a lemma in the original version that incorrectly advertised itself as being the "interchange law" for natural transformations, has been changed to be the correct general statement.

The current version of this document incorporates further revisions, made later in 2020 after the release of Isabelle2020. The theory "category with pullbacks", originally introduced in [6], was moved here and improved somewhat. In addition, new theories were introduced to cover additional common situations of categories with certain kinds of limits: "cartesian category", which concerns categories with binary products and a terminal object, "cartesian closed category", which additionally have exponentials, and "category with finite limits", which is shown to be the same as "category with pullbacks and terminal object". To tie things together and to verify the consistency of the locales (*e.g.* "cartesian closed category") for which concrete interpretations have not yet been given, we construct a category whose objects correspond to the hereditarily finite sets and whose arrows correspond to functions between such sets, and we show that this category is cartesian closed and has finite limits. To facilitate this development, we generalize the "set category" construction to cover some cases in which not every subset of the "universe" need determine an object. In particular, the generalized notion of "set category" covers the case in which only finite sets correspond to objects. This generalization permits us to treat the category of hereditarily finite sets as a "set category" and to apply some results previously shown about limits in such a category.

In early 2022 a construction was added, using "ZFC in HOL", of the (large) category of small sets and functions between them, and it was shown that this category is small-complete.

Chapter 2

Category

theory *Category*
imports *Main HOL−Library.FuncSet*
begin

This theory develops an "object-free" definition of category loosely following [1], Sec. 3.52-3.53. We define the notion "category" in terms of axioms that concern a single partial binary operation on a type, some of whose elements are to be regarded as the "arrows" of the category.

The nonstandard definition of category has some advantages and disadvantages. An advantage is that only one piece of data (the composition operation) is required to specify a category, so the use of records is not required to bundle up several separate objects. A related advantage is the fact that functors and natural transformations can be defined simply to be functions that satisfy certain axioms, rather than more complex composite objects. One disadvantage is that the notions of "object" and "identity arrow" are conflated, though this is easy to get used to. Perhaps a more significant disadvantage is that each arrow of a category must carry along the information about its domain and codomain. This implies, for example, that the arrows of a category of sets and functions cannot be directly identified with functions, but rather only with functions that have been equipped with their domain and codomain sets.

To represent the partiality of the composition operation of a category, we assume that the composition for a category has a unique zero element, which we call *null*, and we consider arrows to be "composable" if and only if their composite is non-null. Functors and natural transformations are required to map arrows to arrows and be "extensional" in the sense that they map non-arrows to null. This is so that equality of functors and natural transformations coincides with their extensional equality as functions in HOL. The fact that we co-opt an element of the arrow type to serve as *null* means that it is not possible to define a category whose arrows exhaust the elements of a given type. This presents a disadvantage in some situations. For example, we cannot construct a discrete category whose arrows are directly identified with the set of *all* elements of a given type $'a$; instead, we must pass to a larger type (such as $'a$ *option*) so that there is an element available for use as *null*. The presence of *null*, however, is crucial to our being able to

define a system of introduction and elimination rules that can be applied automatically to establish that a given expression denotes an arrow. Without *null*, we would be able to define an introduction rule to infer, say, that the composition of composable arrows is composable, but not an elimination rule to infer that arrows are composable from the fact that their composite is an arrow. Having the ability to do both is critical to the usability of the theory.

A *partial magma* is a partial binary operation *OP* defined on the set of elements at a type $'a$. As discussed above, we assume the existence of a unique element *null* of type $'a$ that is a zero for *OP*, and we use *null* to represent "undefined".

locale *partial-magma* =
fixes *OP* :: $'a \Rightarrow 'a \Rightarrow 'a$
assumes *ex-un-null*: $\exists!n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$
begin

definition *null* :: $'a$
where $null = (THE\ n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n)$

lemma *null-eqI*:
assumes $\bigwedge t.\ OP\ n\ t = n \wedge OP\ t\ n = n$
shows $n = null$
using *assms null-def ex-un-null the1-equality* [*of* $\lambda n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$]
by *auto*

lemma *null-is-zero* [*simp*]:
shows $OP\ null\ t = null$ **and** $OP\ t\ null = null$
using *null-def ex-un-null theI'* [*of* $\lambda n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$]
by *auto*

end

2.1 Partial Composition

A *partial composition* is formally the same thing as a partial magma, except that we think of the operation as an operation of "composition", and we regard elements f and g of type $'a$ as *composable* if their composition is non-null.

type-synonym $'a\ comp = 'a \Rightarrow 'a \Rightarrow 'a$

locale *partial-composition* =
partial-magma C
for $C :: 'a\ comp$ (**infixr** $\langle\cdot\rangle$ *55*)
begin

An *identity* is a self-composable element a such that composition of any other element f with a on either the left or the right results in f whenever the composition is defined.

definition *ide*
where $ide\ a \equiv a \cdot a \neq null \wedge$

$$(\forall f.\ (f \cdot a \neq null \longrightarrow f \cdot a = f) \wedge (a \cdot f \neq null \longrightarrow a \cdot f = f))$$

A *domain* of an element f is an identity a for which composition of f with a on the right is defined. The notion *codomain* is defined similarly, using composition on the left. Note that, although these definitions are completely dual, the choice of terminology implies that we will think of composition as being written in traditional order, as opposed to diagram order. It is pretty much essential to do it this way, to maintain compatibility with the notation for function application once we start working with functors and natural transformations.

definition *domains*
where $domains\ f \equiv \{a.\ ide\ a \wedge f \cdot a \neq null\}$

definition *codomains*
where $codomains\ f \equiv \{b.\ ide\ b \wedge b \cdot f \neq null\}$

lemma *domains-null*:
shows $domains\ null = \{\}$
by (*simp add*: *domains-def*)

lemma *codomains-null*:
shows $codomains\ null = \{\}$
by (*simp add*: *codomains-def*)

lemma *self-domain-iff-ide*:
shows $a \in domains\ a \longleftrightarrow ide\ a$
using *ide-def domains-def* **by** *auto*

lemma *self-codomain-iff-ide*:
shows $a \in codomains\ a \longleftrightarrow ide\ a$
using *ide-def codomains-def* **by** *auto*

An element f is an *arrow* if either it has a domain or it has a codomain. In an arbitrary partial magma it is possible for f to have one but not the other, but the *category* locale will include assumptions to rule this out.

definition *arr*
where $arr\ f \equiv domains\ f \neq \{\} \vee codomains\ f \neq \{\}$

lemma *not-arr-null* [*simp*]:
shows $\neg\ arr\ null$
by (*simp add*: *arr-def domains-null codomains-null*)

Using the notions of domain and codomain, we can define *homs*. The predicate *in-hom f a b* expresses "f is an arrow from a to b," and the term *hom a b* denotes the set of all such arrows. It is convenient to have both of these, though passing back and forth sometimes involves extra work. We choose *in-hom* as the more fundamental notion.

definition *in-hom* (‹«- : - → -»›)
where $«f : a \to b» \equiv a \in domains\ f \wedge b \in codomains\ f$

abbreviation *hom*
where *hom a b* ≡ {*f*. «*f* : *a* → *b*»}

lemma *arrI*:
assumes «*f* : *a* → *b*»
shows *arr f*
using *assms arr-def in-hom-def* **by** *auto*

lemma *ide-in-hom* [*intro*]:
shows *ide a* ⟷ «*a* : *a* → *a*»
using *self-domain-iff-ide self-codomain-iff-ide in-hom-def ide-def* **by** *fastforce*

Arrows *f g* for which the composite *g* · *f* is defined are *sequential*.

abbreviation *seq*
where *seq g f* ≡ *arr* (*g* · *f*)

lemma *comp-arr-ide*:
assumes *ide a* **and** *seq f a*
shows *f* · *a* = *f*
using *assms ide-in-hom ide-def not-arr-null* **by** *metis*

lemma *comp-ide-arr*:
assumes *ide b* **and** *seq b f*
shows *b* · *f* = *f*
using *assms ide-in-hom ide-def not-arr-null* **by** *metis*

The *domain* of an arrow *f* is an element chosen arbitrarily from the set of domains of *f* and the *codomain* of *f* is an element chosen arbitrarily from the set of codomains.

definition *dom*
where *dom f* = (*if domains f* ≠ {} *then* (*SOME a*. *a* ∈ *domains f*) *else null*)

definition *cod*
where *cod f* = (*if codomains f* ≠ {} *then* (*SOME b*. *b* ∈ *codomains f*) *else null*)

lemma *dom-null* [*simp*]:
shows *dom null* = *null*
by (*simp add*: *dom-def domains-null*)

lemma *cod-null* [*simp*]:
shows *cod null* = *null*
by (*simp add*: *cod-def codomains-null*)

lemma *dom-in-domains*:
assumes *domains f* ≠ {}
shows *dom f* ∈ *domains f*
using *assms dom-def someI* [*of* λ*a*. *a* ∈ *domains f*] **by** *auto*

lemma *cod-in-codomains*:
assumes *codomains f* ≠ {}

shows *cod f* ∈ *codomains f*
using *assms cod-def someI* [*of* λ*b*. *b* ∈ *codomains f*] **by** *auto*

end

2.2 Categories

A *category* is defined to be a partial magma whose composition satisfies an extensionality condition, an associativity condition, and the requirement that every arrow have both a domain and a codomain. The associativity condition involves four "matching conditions" (*match-1*, *match-2*, *match-3*, and *match-4*) which constrain the domain of definition of the composition, and a fifth condition (*comp-assoc′*) which states that the results of the two ways of composing three elements are equal. In the presence of the *comp-assoc′* axiom *match-4* can be derived from *match-3* and vice versa.

locale *category* = *partial-composition* +
assumes *ext*: *g* · *f* ≠ *null* ⟹ *seq g f*
and *has-domain-iff-has-codomain*: *domains f* ≠ {} ⟷ *codomains f* ≠ {}
and *match-1*: ⟦ *seq h g*; *seq* (*h* · *g*) *f* ⟧ ⟹ *seq g f*
and *match-2*: ⟦ *seq h* (*g* · *f*); *seq g f* ⟧ ⟹ *seq h g*
and *match-3*: ⟦ *seq g f*; *seq h g* ⟧ ⟹ *seq* (*h* · *g*) *f*
and *comp-assoc′*: ⟦ *seq g f*; *seq h g* ⟧ ⟹ (*h* · *g*) · *f* = *h* · *g* · *f*
begin

Associativity of composition holds unconditionally. This was not the case in previous, weaker versions of this theory, and I did not notice this for some time after updating to the current axioms. It is obviously an advantage that no additional hypotheses have to be verified in order to apply associativity, but a disadvantage is that this fact is now "too readily applicable," so that if it is made a default simplification it tends to get in the way of applying other simplifications that we would also like to be able to apply automatically. So, it now seems best not to make this fact a default simplification, but rather to invoke it explicitly where it is required.

lemma *comp-assoc*:
shows (*h* · *g*) · *f* = *h* · *g* · *f*
by (*metis comp-assoc′ ex-un-null ext match-1 match-2*)

lemma *match-4*:
assumes *seq g f* **and** *seq h g*
shows *seq h* (*g* · *f*)
using *assms match-3 comp-assoc* **by** *auto*

lemma *domains-comp*:
assumes *seq g f*
shows *domains* (*g* · *f*) = *domains f*
proof −
have *domains* (*g* · *f*) = {*a*. *ide a* ∧ *seq* (*g* · *f*) *a*}
using *domains-def ext* **by** *auto*
also have ... = {*a*. *ide a* ∧ *seq f a*}

```
    using assms ide-def match-1 match-3 by meson
  also have ... = domains f
    using domains-def ext by auto
  finally show ?thesis by blast
qed

lemma codomains-comp:
assumes seq g f
shows codomains (g · f) = codomains g
proof −
  have codomains (g · f) = {b. ide b ∧ seq b (g · f)}
    using codomains-def ext by auto
  also have ... = {b. ide b ∧ seq b g}
    using assms ide-def match-2 match-4 by meson
  also have ... = codomains g
    using codomains-def ext by auto
  finally show ?thesis by blast
qed

lemma has-domain-iff-arr:
shows domains f ≠ {} ⟷ arr f
  by (simp add: arr-def has-domain-iff-has-codomain)

lemma has-codomain-iff-arr:
shows codomains f ≠ {} ⟷ arr f
  using has-domain-iff-arr has-domain-iff-has-codomain by auto
```

A consequence of the category axioms is that domains and codomains, if they exist, are unique.

```
lemma domain-unique:
assumes a ∈ domains f and a' ∈ domains f
shows a = a'
proof −
  have ide a ∧ seq f a ∧ ide a' ∧ seq f a'
    using assms domains-def ext by force
  thus ?thesis
    using match-1 ide-def not-arr-null by metis
qed

lemma codomain-unique:
assumes b ∈ codomains f and b' ∈ codomains f
shows b = b'
proof −
  have ide b ∧ seq b f ∧ ide b' ∧ seq b' f
    using assms codomains-def ext by force
  thus ?thesis
    using match-2 ide-def not-arr-null by metis
qed
```

lemma *domains-simp*:
assumes *arr f*
shows *domains f* = {*dom f*}
using *assms dom-in-domains has-domain-iff-arr domain-unique* **by** *auto*

lemma *codomains-simp*:
assumes *arr f*
shows *codomains f* = {*cod f*}
using *assms cod-in-codomains has-codomain-iff-arr codomain-unique* **by** *auto*

lemma *domains-char*:
shows *domains f* = (*if arr f then* {*dom f*} *else* {})
using *dom-in-domains has-domain-iff-arr domain-unique* **by** *auto*

lemma *codomains-char*:
shows *codomains f* = (*if arr f then* {*cod f*} *else* {})
using *cod-in-codomains has-codomain-iff-arr codomain-unique* **by** *auto*

A consequence of the following lemma is that the notion *arr* is redundant, given *in-hom*, *dom*, and *cod*. However, I have retained it because I have not been able to find a set of usefully powerful simplification rules expressed only in terms of *in-hom* that does not result in looping in many situations.

lemma *arr-iff-in-hom*:
shows *arr f* ⟷ «*f* : *dom f* → *cod f*»
using *cod-in-codomains dom-in-domains has-domain-iff-arr has-codomain-iff-arr in-hom-def*
by *auto*

lemma *in-homI* [*intro*]:
assumes *arr f* **and** *dom f* = *a* **and** *cod f* = *b*
shows «*f* : *a* → *b*»
using *assms cod-in-codomains dom-in-domains has-domain-iff-arr has-codomain-iff-arr in-hom-def*
by *auto*

lemma *in-homE* [*elim*]:
assumes «*f* : *a* → *b*»
and *arr f* ⟹ *dom f* = *a* ⟹ *cod f* = *b* ⟹ *T*
shows *T*
using *assms in-hom-def domains-char codomains-char has-domain-iff-arr*
by (*metis empty-iff singleton-iff*)

To obtain the "only if" direction in the next two results and in similar results later for composition and the application of functors and natural transformations, is the reason for assuming the existence of *null* as a special element of the arrow type, as opposed to, say, using option types to represent partiality. The presence of *null* allows us not only to make the "upward" inference that the domain of an arrow is again an arrow, but also to make the "downward" inference that if *dom f* is an arrow then so is *f*. Similarly, we will be able to infer not only that if f and g are composable arrows then $g \cdot f$ is an arrow, but also that if $g \cdot f$ is an arrow then f and g are composable arrows. These inferences

allow most necessary facts about what terms denote arrows to be deduced automatically from minimal assumptions. Typically all that is required is to assume or establish that certain terms denote arrows in particular homs at the point where those terms are first introduced, and then similar facts about related terms can be derived automatically. Without this feature, nearly every proof would involve many tedious additional steps to establish that each of the terms appearing in the proof (including all its subterms) in fact denote arrows.

lemma *arr-dom-iff-arr*:
shows *arr* (*dom f*) ⟷ *arr f*
using *dom-def dom-in-domains has-domain-iff-arr self-domain-iff-ide domains-def*
by *fastforce*

lemma *arr-cod-iff-arr*:
shows *arr* (*cod f*) ⟷ *arr f*
using *cod-def cod-in-codomains has-codomain-iff-arr self-codomain-iff-ide codomains-def*
by *fastforce*

lemma *arr-dom* [*simp*]:
assumes *arr f*
shows *arr* (*dom f*)
using *assms arr-dom-iff-arr* **by** *simp*

lemma *arr-cod* [*simp*]:
assumes *arr f*
shows *arr* (*cod f*)
using *assms arr-cod-iff-arr* **by** *simp*

lemma *seqI* [*simp*]:
assumes *arr f* **and** *arr g* **and** *dom g* = *cod f*
shows *seq g f*
proof −
have *ide* (*cod f*) ∧ *seq* (*cod f*) *f*
using *assms*(*1*) *has-codomain-iff-arr codomains-def cod-in-codomains ext* **by** *blast*
moreover have *ide* (*cod f*) ∧ *seq g* (*cod f*)
using *assms*(*2*−*3*) *domains-def domains-simp ext* **by** *fastforce*
ultimately show *?thesis*
using *match-4 ide-def ext* **by** *metis*
qed

This version of *seqI* is useful as an introduction rule, but not as useful as a simplification, because it requires finding the intermediary term *b*. Sometimes *auto* is able to do this, but other times it is more expedient just to invoke this rule and fill in the missing terms manually, especially when dealing with a chain of compositions.

lemma *seqI'* [*intro*]:
assumes «*f* : *a* → *b*» **and** «*g* : *b* → *c*»
shows *seq g f*
using *assms* **by** *fastforce*

lemma *compatible-iff-seq*:
shows *domains g* ∩ *codomains f* ≠ {} ⟷ *seq g f*
proof
show *domains g* ∩ *codomains f* ≠ {} ⟹ *seq g f*
using *cod-in-codomains dom-in-domains empty-iff has-domain-iff-arr has-codomain-iff-arr domain-unique codomain-unique*
by (*metis Int-emptyI seqI*)
show *seq g f* ⟹ *domains g* ∩ *codomains f* ≠ {}
proof −
assume *gf*: *seq g f*
have *1*: *cod f* ∈ *codomains f*
using *gf has-domain-iff-arr domains-comp cod-in-codomains codomains-simp* **by** *blast*
have *ide* (*cod f*) ∧ *seq* (*cod f*) *f*
using *1 codomains-def ext* **by** *auto*
hence *seq g* (*cod f*)
using *gf has-domain-iff-arr match-2 domains-null ide-def* **by** *metis*
thus *?thesis*
using *domains-def 1 codomains-def* **by** *auto*
qed
qed

The following is another example of a crucial "downward" rule that would not be possible without a reserved *null* value.

lemma *seqE* [*elim*]:
assumes *seq g f*
and *arr f* ⟹ *arr g* ⟹ *dom g* = *cod f* ⟹ *T*
shows *T*
using *assms cod-in-codomains compatible-iff-seq has-domain-iff-arr has-codomain-iff-arr domains-comp codomains-comp domains-char codomain-unique*
by (*metis Int-emptyI singletonD*)

lemma *comp-in-homI* [*intro*]:
assumes «*f* : *a* → *b*» **and** «*g* : *b* → *c*»
shows «*g* · *f* : *a* → *c*»
using *assms*(*1*−*2*) *codomains-comp domains-comp in-hom-def seqI'* **by** *auto*

lemma *comp-in-homI'* [*simp*]:
assumes *arr f* **and** *arr g* **and** *dom f* = *a* **and** *cod g* = *c* **and** *dom g* = *cod f*
shows «*g* · *f* : *a* → *c*»
using *assms* **by** *auto*

lemma *comp-in-homE* [*elim*]:
assumes «*g* · *f* : *a* → *c*»
obtains *b* **where** «*f* : *a* → *b*» **and** «*g* : *b* → *c*»
using *assms in-hom-def domains-comp codomains-comp*
by (*metis arrI in-homI seqE*)

The next two rules are useful as simplifications, but they slow down the simplifier too much to use them by default. So it is necessary to guess when they are needed and cite them explicitly. This is usually not too difficult.

lemma *comp-arr-dom*:
assumes *arr f* **and** *dom f* = *a*
shows *f* · *a* = *f*
using *assms dom-in-domains has-domain-iff-arr domains-def ide-def* **by** *auto*

lemma *comp-cod-arr*:
assumes *arr f* **and** *cod f* = *b*
shows *b* · *f* = *f*
using *assms cod-in-codomains has-codomain-iff-arr ide-def codomains-def* **by** *auto*

lemma *ide-char*:
shows *ide a* ⟷ *arr a* ∧ *dom a* = *a* ∧ *cod a* = *a*
using *ide-in-hom* **by** *auto*

In some contexts, this rule causes the simplifier to loop, but it is too useful not to have as a default simplification. In cases where it is a problem, usually a method like *blast* or *force* will succeed if this rule is cited explicitly.

lemma *ideD* [*simp*]:
assumes *ide a*
shows *arr a* **and** *dom a* = *a* **and** *cod a* = *a*
using *assms ide-char* **by** *auto*

lemma *ide-dom* [*simp*]:
assumes *arr f*
shows *ide* (*dom f*)
using *assms dom-in-domains has-domain-iff-arr domains-def* **by** *auto*

lemma *ide-cod* [*simp*]:
assumes *arr f*
shows *ide* (*cod f*)
using *assms cod-in-codomains has-codomain-iff-arr codomains-def* **by** *auto*

lemma *dom-eqI*:
assumes *ide a* **and** *seq f a*
shows *dom f* = *a*
using *assms cod-in-codomains codomain-unique ide-char*
by (*metis seqE*)

lemma *cod-eqI*:
assumes *ide b* **and** *seq b f*
shows *cod f* = *b*
using *assms dom-in-domains domain-unique ide-char*
by (*metis seqE*)

lemma *dom-eqI′*:
assumes *a* ∈ *domains f*
shows *a* = *dom f*
using *assms dom-in-domains domain-unique* **by** *blast*

lemma *cod-eqI'*:
assumes $a \in \mathit{codomains}\ f$
shows $a = \mathit{cod}\ f$
using *assms cod-in-codomains codomain-unique* **by** *blast*

lemma *ide-char'*:
shows $\mathit{ide}\ a \longleftrightarrow \mathit{arr}\ a \wedge (\mathit{dom}\ a = a \vee \mathit{cod}\ a = a)$
using *ide-dom ide-cod ide-char* **by** *metis*

lemma *dom-dom*:
shows $\mathit{dom}\ (\mathit{dom}\ f) = \mathit{dom}\ f$
by (*metis dom-null domains-char ideD*(2) *ide-dom dom-def*)

lemma *cod-cod*:
shows $\mathit{cod}\ (\mathit{cod}\ f) = \mathit{cod}\ f$
by (*metis arr-cod-iff-arr cod-def has-codomain-iff-arr ideD*(3) *ide-cod*)

lemma *dom-cod*:
shows $\mathit{dom}\ (\mathit{cod}\ f) = \mathit{cod}\ f$
by (*metis arr-cod-iff-arr cod-def has-codomain-iff-arr has-domain-iff-arr ideD*(2)
ide-cod dom-def)

lemma *cod-dom*:
shows $\mathit{cod}\ (\mathit{dom}\ f) = \mathit{dom}\ f$
by (*metis cod-null has-domain-iff-arr ideD*(3) *ide-dom dom-def*)

lemma *dom-comp* [*simp*]:
assumes *seq g f*
shows $\mathit{dom}\ (g \cdot f) = \mathit{dom}\ f$
using *assms* **by** (*simp add*: *dom-def domains-comp*)

lemma *cod-comp* [*simp*]:
assumes *seq g f*
shows $\mathit{cod}\ (g \cdot f) = \mathit{cod}\ g$
using *assms* **by** (*simp add*: *cod-def codomains-comp*)

lemma *comp-ide-self* [*simp*]:
assumes *ide a*
shows $a \cdot a = a$
using *assms comp-arr-ide arrI* **by** *auto*

lemma *ide-compE* [*elim*]:
assumes $\mathit{ide}\ (g \cdot f)$
and $\mathit{seq}\ g\ f \Longrightarrow \mathit{seq}\ f\ g \Longrightarrow g \cdot f = \mathit{dom}\ f \Longrightarrow g \cdot f = \mathit{cod}\ g \Longrightarrow T$
shows *T*
using *assms dom-comp cod-comp ide-char ide-in-hom*
by (*metis seqE seqI*)

The next two results are sometimes useful for performing manipulations at the head of a chain of composed arrows. I have adopted the convention that such chains are canon-

ically represented in right-associated form. This makes it easy to perform manipulations at the "tail" of a chain, but more difficult to perform them at the "head". These results take care of the rote manipulations using associativity that are needed to either permute or combine arrows at the head of a chain.

lemma *comp-permute*:
assumes $f \cdot g = k \cdot l$ **and** *seq f g* **and** *seq g h*
shows $f \cdot g \cdot h = k \cdot l \cdot h$
using *assms* **by** (*metis comp-assoc*)

lemma *comp-reduce*:
assumes $f \cdot g = k$ **and** *seq f g* **and** *seq g h*
shows $f \cdot g \cdot h = k \cdot h$
using *assms comp-assoc* **by** *auto*

Here we define some common configurations of arrows. These are defined as abbreviations, because we want all "diagrammatic" assumptions in a theorem to reduce readily to a conjunction of assertions of the basic forms *arr f*, $dom\ f = X$, $cod\ f = Y$, and «$f : a \rightarrow b$».

abbreviation *endo*
where *endo f* ≡ *seq f f*

abbreviation *antipar*
where *antipar f g* ≡ *seq g f* ∧ *seq f g*

abbreviation *span*
where *span f g* ≡ *arr f* ∧ *arr g* ∧ $dom\ f = dom\ g$

abbreviation *cospan*
where *cospan f g* ≡ *arr f* ∧ *arr g* ∧ $cod\ f = cod\ g$

abbreviation *par*
where *par f g* ≡ *arr f* ∧ *arr g* ∧ $dom\ f = dom\ g$ ∧ $cod\ f = cod\ g$

end

end

Chapter 3

EpiMonoIso

theory *EpiMonoIso*
imports *Category*
begin

This theory defines and develops properties of epimorphisms, monomorphisms, isomorphisms, sections, and retractions.

context *category*
begin

definition *epi*
where *epi f* = (*arr f* ∧ *inj-on* (λ*g*. *g* · *f*) {*g*. *seq g f*})

definition *mono*
where *mono f* = (*arr f* ∧ *inj-on* (λ*g*. *f* · *g*) {*g*. *seq f g*})

lemma *epiI* [*intro*]:
assumes *arr f* **and** ⋀*g g'*. ⟦*seq g f*; *seq g' f*; *g* · *f* = *g'* · *f*⟧ ⟹ *g* = *g'*
shows *epi f*
using *assms epi-def inj-on-def* **by** *blast*

lemma *epi-implies-arr*:
assumes *epi f*
shows *arr f*
using *assms epi-def* **by** *auto*

lemma *epi-cancel*:
assumes *epi f*
and *seq g f* **and** *g* · *f* = *g'* · *f*
shows *g* = *g'*
using *assms* **unfolding** *epi-def inj-on-def* **by** *auto*

lemma *monoI* [*intro*]:
assumes *arr g* **and** ⋀*f f'*. ⟦*seq g f*; *g* · *f* = *g* · *f'*⟧ ⟹ *f* = *f'*
shows *mono g*
using *assms mono-def inj-on-def* **by** *blast*

lemma *mono-implies-arr*:
assumes *mono f*
shows *arr f*
using *assms mono-def* **by** *auto*

lemma *mono-cancel*:
assumes *mono g*
and *seq g f* **and** $g \cdot f = g \cdot f'$
shows $f' = f$
using *assms* **unfolding** *mono-def inj-on-def* **by** *auto*

definition *inverse-arrows*
where *inverse-arrows f g* $\equiv$ *ide* $(g \cdot f) \wedge$ *ide* $(f \cdot g)$

lemma *inverse-arrowsI* [*intro*]:
assumes *ide* $(g \cdot f)$ **and** *ide* $(f \cdot g)$
shows *inverse-arrows f g*
using *assms inverse-arrows-def* **by** *blast*

lemma *inverse-arrowsE* [*elim*]:
assumes *inverse-arrows f g*
and $[\![$ *ide* $(g \cdot f)$; *ide* $(f \cdot g)$ $]\!] \Longrightarrow T$
shows T
using *assms inverse-arrows-def* **by** *blast*

lemma *inverse-arrows-sym*:
shows *inverse-arrows f g* $\longleftrightarrow$ *inverse-arrows g f*
using *inverse-arrows-def* **by** *auto*

lemma *ide-self-inverse*:
assumes *ide a*
shows *inverse-arrows a a*
using *assms* **by** *auto*

lemma *inverse-arrow-unique*:
assumes *inverse-arrows f g* **and** *inverse-arrows f g'*
shows $g = g'$
using *assms* **apply** (*elim inverse-arrowsE*)
by (*metis comp-cod-arr ide-compE comp-assoc seqE*)

lemma *inverse-arrows-compose*:
assumes *seq g f* **and** *inverse-arrows f f'* **and** *inverse-arrows g g'*
shows *inverse-arrows* $(g \cdot f)$ $(f' \cdot g')$
using *assms* **apply** (*elim inverse-arrowsE*, *intro inverse-arrowsI*)
apply (*metis seqE comp-arr-dom ide-compE comp-assoc*)
by (*metis seqE comp-arr-dom ide-compE comp-assoc*)

definition *section*

where *section f* ≡ ∃ *g*. *ide* (*g* · *f*)

lemma *sectionI* [*intro*]:
assumes *ide* (*g* · *f*)
shows *section f*
  using *assms section-def* **by** *auto*

lemma *sectionE* [*elim*]:
assumes *section f*
obtains *g* **where** *ide* (*g* · *f*)
  using *assms section-def* **by** *blast*

definition *retraction*
where *retraction g* ≡ ∃ *f*. *ide* (*g* · *f*)

lemma *retractionI* [*intro*]:
assumes *ide* (*g* · *f*)
shows *retraction g*
  using *assms retraction-def* **by** *auto*

lemma *retractionE* [*elim*]:
assumes *retraction g*
obtains *f* **where** *ide* (*g* · *f*)
  using *assms retraction-def* **by** *blast*

lemma *section-is-mono*:
assumes *section g*
shows *mono g*
proof
  show *arr g* **using** *assms section-def* **by** *blast*
  from *assms* **obtain** *h* **where** *h*: *ide* (*h* · *g*) **by** *blast*
  have *hg*: *seq h g* **using** *h* **by** *auto*
  thus ⋀*f f'*. ⟦*seq g f*; *g* · *f* = *g* · *f'*⟧ ⟹ *f* = *f'*
    using *hg h ide-compE seqE comp-assoc comp-cod-arr* **by** *metis*
qed

lemma *retraction-is-epi*:
assumes *retraction g*
shows *epi g*
proof
  show *arr g* **using** *assms retraction-def* **by** *blast*
  from *assms* **obtain** *f* **where** *f*: *ide* (*g* · *f*) **by** *blast*
  have *gf*: *seq g f* **using** *f* **by** *auto*
  thus ⋀*h h'*. ⟦*seq h g*; *seq h' g*; *h* · *g* = *h'* · *g*⟧ ⟹ *h* = *h'*
    using *gf f ide-compE seqE comp-assoc comp-arr-dom* **by** *metis*
qed

lemma *section-retraction-compose*:
assumes *ide* (*e* · *m*) **and** *ide* (*e'* · *m'*) **and** *seq m' m*

shows *ide* $((e \cdot e') \cdot (m' \cdot m))$
using *assms seqI seqE ide-compE comp-assoc comp-arr-dom* **by** *metis*

lemma *sections-compose* [*intro*]:
assumes *section m* **and** *section m'* **and** *seq m' m*
shows *section* $(m' \cdot m)$
using *assms section-def section-retraction-compose* **by** *metis*

lemma *retractions-compose* [*intro*]:
assumes *retraction e* **and** *retraction e'* **and** *seq e' e*
shows *retraction* $(e' \cdot e)$
proof −
from *assms*(*1*−*2*) **obtain** *m m'*
where ∗: *ide* $(e \cdot m) \wedge$ *ide* $(e' \cdot m')$
using *retraction-def* **by** *auto*
hence *seq m m'*
using *assms*(*3*) **by** (*metis seqE seqI ide-compE*)
with ∗ **show** *?thesis*
using *section-retraction-compose retractionI* **by** *blast*
qed

lemma *monos-compose* [*intro*]:
assumes *mono m* **and** *mono m'* **and** *seq m' m*
shows *mono* $(m' \cdot m)$
proof −
have *inj-on* $(\lambda f.\ (m' \cdot m) \cdot f)$ $\{f.\ seq\ (m' \cdot m)\ f\}$
unfolding *inj-on-def*
using *assms*
by (*metis CollectD seqE mono-cancel comp-assoc*)
thus *?thesis* **using** *assms*(*3*) *mono-def* **by** *force*
qed

lemma *epis-compose* [*intro*]:
assumes *epi e* **and** *epi e'* **and** *seq e' e*
shows *epi* $(e' \cdot e)$
proof −
have *inj-on* $(\lambda g.\ g \cdot (e' \cdot e))$ $\{g.\ seq\ g\ (e' \cdot e)\}$
unfolding *inj-on-def*
using *assms* **by** (*metis CollectD epi-cancel match-2 comp-assoc*)
thus *?thesis* **using** *assms*(*3*) *epi-def* **by** *force*
qed

definition *iso*
where *iso f* $\equiv \exists g.$ *inverse-arrows f g*

lemma *isoI* [*intro*]:
assumes *inverse-arrows f g*
shows *iso f*
using *assms iso-def* **by** *auto*

lemma *isoE* [*elim*]:
assumes *iso f*
obtains *g* **where** *inverse-arrows f g*
using *assms iso-def* **by** *blast*

lemma *ide-is-iso* [*simp*]:
assumes *ide a*
shows *iso a*
using *assms ide-self-inverse* **by** *auto*

lemma *iso-is-arr*:
assumes *iso f*
shows *arr f*
using *assms* **by** *blast*

lemma *iso-is-section*:
assumes *iso f*
shows *section f*
using *assms inverse-arrows-def* **by** *blast*

lemma *iso-is-retraction*:
assumes *iso f*
shows *retraction f*
using *assms inverse-arrows-def* **by** *blast*

lemma *iso-iff-mono-and-retraction*:
shows *iso f* $\longleftrightarrow$ *mono f* $\wedge$ *retraction f*
proof
show *iso f* $\Longrightarrow$ *mono f* $\wedge$ *retraction f*
by (*simp add*: *iso-is-retraction iso-is-section section-is-mono*)
show *mono f* $\wedge$ *retraction f* $\Longrightarrow$ *iso f*
proof −
assume *f*: *mono f* $\wedge$ *retraction f*
from *f* **obtain** *g* **where** *g*: *ide* (*f* · *g*) **by** *blast*
have *inverse-arrows f g*
using *f g comp-arr-dom comp-cod-arr comp-assoc inverse-arrowsI*
by (*metis ide-char*′ *ide-compE mono-cancel mono-implies-arr*)
thus *iso f* **by** *auto*
qed
qed

lemma *iso-iff-section-and-epi*:
shows *iso f* $\longleftrightarrow$ *section f* $\wedge$ *epi f*
proof
show *iso f* $\Longrightarrow$ *section f* $\wedge$ *epi f*
by (*simp add*: *iso-is-retraction iso-is-section retraction-is-epi*)
show *section f* $\wedge$ *epi f* $\Longrightarrow$ *iso f*
proof −

```
    assume f: section f ∧ epi f
    from f obtain g where g: ide (g · f) by blast
    have inverse-arrows f g
      using f g comp-arr-dom comp-cod-arr epi-implies-arr
            comp-assoc ide-compE inverse-arrowsI epi-cancel ide-char'
      by metis
    thus iso f by auto
  qed
qed

lemma iso-iff-section-and-retraction:
shows iso f ⟷ section f ∧ retraction f
  using iso-is-retraction iso-is-section iso-iff-mono-and-retraction section-is-mono
  by auto

lemma isos-compose [intro]:
assumes iso f and iso f' and seq f' f
shows iso (f' · f)
proof −
  from assms(1) obtain g where g: inverse-arrows f g by blast
  from assms(2) obtain g' where g': inverse-arrows f' g' by blast
  have inverse-arrows (f' · f) (g · g')
    using assms g g inverse-arrowsI inverse-arrowsE section-retraction-compose
    by (simp add: g' inverse-arrows-compose)
  thus ?thesis using iso-def by auto
qed

lemma iso-cancel-left:
assumes iso f and f · g = f · g' and seq f g
shows g = g'
  using assms iso-is-section section-is-mono mono-cancel by metis

lemma iso-cancel-right:
assumes iso g and f · g = f' · g and seq f g and iso g
shows f = f'
  using assms iso-is-retraction retraction-is-epi epi-cancel by metis

definition isomorphic
where isomorphic a a' = (∃f. «f : a → a'» ∧ iso f)

lemma isomorphicI [intro]:
assumes iso f
shows isomorphic (dom f) (cod f)
  using assms isomorphic-def iso-is-arr by blast

lemma isomorphicE [elim]:
assumes isomorphic a a'
obtains f where «f : a → a'» ∧ iso f
  using assms isomorphic-def by meson
```

definition *iso-in-hom* (‹«- : - ≅ -»›)
where *iso-in-hom f a b* ≡ «*f* : *a* → *b*» ∧ *iso f*

lemma *iso-in-homI* [*intro*]:
assumes «*f* : *a* → *b*» **and** *iso f*
shows «*f* : *a* ≅ *b*»
  using *assms iso-in-hom-def* **by** *simp*

lemma *iso-in-homE* [*elim*]:
assumes «*f* : *a* ≅ *b*»
and ⟦«*f* : *a* → *b*»; *iso f*⟧ ⟹ *T*
shows *T*
  using *assms iso-in-hom-def* **by** *simp*

lemma *isomorphicI'*:
assumes «*f* : *a* ≅ *b*»
shows *isomorphic a b*
  using *assms iso-in-hom-def isomorphic-def* **by** *auto*

lemma *ide-iso-in-hom*:
assumes *ide a*
shows «*a* : *a* ≅ *a*»
  using *assms* **by** *fastforce*

lemma *comp-iso-in-hom* [*intro*]:
assumes «*f* : *a* ≅ *b*» **and** «*g* : *b* ≅ *c*»
shows «*g* · *f* : *a* ≅ *c*»
  using *assms iso-in-hom-def* **by** *auto*

definition *inv*
where *inv f* = (*SOME g*. *inverse-arrows f g*)

lemma *inv-is-inverse*:
assumes *iso f*
shows *inverse-arrows f* (*inv f*)
  using *assms inv-def someI* [*of inverse-arrows f*] **by** *auto*

lemma *iso-inv-iso* [*intro*, *simp*]:
assumes *iso f*
shows *iso* (*inv f*)
  using *assms inv-is-inverse inverse-arrows-sym* **by** *blast*

lemma *inverse-unique*:
assumes *inverse-arrows f g*
shows *inv f* = *g*
  using *assms inv-is-inverse inverse-arrow-unique isoI* **by** *auto*

lemma *inv-ide* [*simp*]:

assumes *ide a*
shows *inv a = a*
using *assms* **by** (*simp add*: *inverse-arrowsI inverse-unique*)

lemma *inv-inv* [*simp*]:
assumes *iso f*
shows *inv* (*inv f*) = *f*
using *assms inverse-arrows-sym inverse-unique* **by** *blast*

lemma *comp-arr-inv*:
assumes *inverse-arrows f g*
shows *f* · *g* = *dom g*
using *assms* **by** *auto*

lemma *comp-inv-arr*:
assumes *inverse-arrows f g*
shows *g* · *f* = *dom f*
using *assms* **by** *auto*

lemma *comp-arr-inv′*:
assumes *iso f*
shows *f* · *inv f* = *cod f*
using *assms inv-is-inverse* **by** *blast*

lemma *comp-inv-arr′*:
assumes *iso f*
shows *inv f* · *f* = *dom f*
using *assms inv-is-inverse* **by** *blast*

lemma *inv-in-hom* [*simp*]:
assumes *iso f* **and** «*f* : *a* → *b*»
shows «*inv f* : *b* → *a*»
using *assms inv-is-inverse seqE inverse-arrowsE*
by (*metis ide-compE in-homE in-homI*)

lemma *arr-inv* [*simp*]:
assumes *iso f*
shows *arr* (*inv f*)
using *assms inv-in-hom* **by** *blast*

lemma *dom-inv* [*simp*]:
assumes *iso f*
shows *dom* (*inv f*) = *cod f*
using *assms inv-in-hom* **by** *blast*

lemma *cod-inv* [*simp*]:
assumes *iso f*
shows *cod* (*inv f*) = *dom f*
using *assms inv-in-hom* **by** *blast*

lemma *inv-comp*:
assumes *iso f* **and** *iso g* **and** *seq g f*
shows *inv* $(g \cdot f) = inv\ f \cdot inv\ g$
using *assms inv-is-inverse inverse-unique inverse-arrows-compose inverse-arrows-def*
by *meson*

lemma *isomorphic-reflexive*:
assumes *ide f*
shows *isomorphic f f*
unfolding *isomorphic-def*
using *assms ide-is-iso ide-in-hom* **by** *blast*

lemma *isomorphic-symmetric*:
assumes *isomorphic f g*
shows *isomorphic g f*
using *assms inv-in-hom* **by** *blast*

lemma *isomorphic-transitive* [*trans*]:
assumes *isomorphic f g* **and** *isomorphic g h*
shows *isomorphic f h*
using *assms isomorphic-def isos-compose* **by** *auto*

A section or retraction of an isomorphism is in fact an inverse.

lemma *section-retraction-of-iso*:
assumes *iso f*
shows *ide* $(g \cdot f) \Longrightarrow$ *inverse-arrows f g*
and *ide* $(f \cdot g) \Longrightarrow$ *inverse-arrows f g*
proof −
show *ide* $(g \cdot f) \Longrightarrow$ *inverse-arrows f g*
using *assms*
by (*metis comp-inv-arr′ epi-cancel ide-compE inv-is-inverse*
iso-iff-section-and-epi)
show *ide* $(f \cdot g) \Longrightarrow$ *inverse-arrows f g*
using *assms*
by (*metis ide-compE comp-arr-inv′ inv-is-inverse iso-iff-mono-and-retraction*
mono-cancel)
qed

A situation that occurs frequently is that we have a commuting triangle, but we need the triangle obtained by inverting one side that is an isomorphism. The following fact streamlines this derivation.

lemma *invert-side-of-triangle*:
assumes *arr h* **and** $f \cdot g = h$
shows *iso f* $\Longrightarrow$ *seq* $(inv\ f)\ h \wedge g = inv\ f \cdot h$
and *iso g* $\Longrightarrow$ *seq h* $(inv\ g) \wedge f = h \cdot inv\ g$
proof −
show *iso f* $\Longrightarrow$ *seq* $(inv\ f)\ h \wedge g = inv\ f \cdot h$
by (*metis assms seqE inv-is-inverse comp-cod-arr comp-inv-arr comp-assoc*)

show *iso g* $\Longrightarrow$ *seq h* (*inv g*) $\wedge$ *f* = *h* $\cdot$ *inv g*
by (*metis assms seqE inv-is-inverse comp-arr-dom comp-arr-inv dom-inv comp-assoc*)
qed

A similar situation is where we have a commuting square and we want to invert two opposite sides.

lemma *invert-opposite-sides-of-square*:
assumes *seq f g* **and** *f* $\cdot$ *g* = *h* $\cdot$ *k*
shows $[\![$ *iso f*; *iso k* $]\!] \Longrightarrow$ *seq g* (*inv k*) $\wedge$ *seq* (*inv f*) *h* $\wedge$ *g* $\cdot$ *inv k* = *inv f* $\cdot$ *h*
by (*metis assms invert-side-of-triangle comp-assoc*)

The following versions of *inv-comp* provide information needed for repeated application to a composition of more than two arrows and seem often to be more useful.

lemma *inv-comp-left*:
assumes *iso* (*g* $\cdot$ *f*) **and** *iso g*
shows *inv* (*g* $\cdot$ *f*) = *inv f* $\cdot$ *inv g* **and** *iso f*
proof −
have *1*: *inv f* = *inv* (*g* $\cdot$ *f*) $\cdot$ *g*
proof −
have *inv* (*g* $\cdot$ *f*) $\cdot$ *g* = *inv* (*g* $\cdot$ *f*) $\cdot$ *inv* (*inv g*)
using *assms* **by** *simp*
also have ... = *inv* (*inv g* $\cdot$ *g* $\cdot$ *f*)
using *assms inv-comp iso-is-arr* **by** *simp*
also have ... = *inv* ((*inv g* $\cdot$ *g*) $\cdot$ *f*)
using *comp-assoc* **by** *simp*
also have ... = *inv f*
using *assms comp-ide-arr invert-side-of-triangle*(*1*) *iso-is-arr comp-assoc*
by *metis*
finally show *?thesis* **by** *simp*
qed
show *inv* (*g* $\cdot$ *f*) = *inv f* $\cdot$ *inv g*
using *assms 1 comp-arr-dom comp-assoc*
by (*metis arr-inv cod-comp dom-inv invert-side-of-triangle*(*2*) *iso-is-arr seqI*)
show *iso f*
using *assms 1 comp-assoc inv-is-inverse*
by (*metis arr-inv invert-side-of-triangle*(*1*) *inv-inv iso-inv-iso isos-compose*)
qed

lemma *inv-comp-right*:
assumes *iso* (*g* $\cdot$ *f*) **and** *iso f*
shows *inv* (*g* $\cdot$ *f*) = *inv f* $\cdot$ *inv g* **and** *iso g*
proof −
have *1*: *inv g* = *f* $\cdot$ *inv* (*g* $\cdot$ *f*)
proof −
have *f* $\cdot$ *inv* (*g* $\cdot$ *f*) = *inv* (*inv f*) $\cdot$ *inv* (*g* $\cdot$ *f*)
using *assms* **by** *simp*
also have ... = *inv* ((*g* $\cdot$ *f*) $\cdot$ *inv f*)
using *assms inv-comp iso-is-arr* **by** *simp*
also have ... = *inv* (*g* $\cdot$ *f* $\cdot$ *inv f*)

```
        using comp-assoc by simp
      also have ... = inv g
        using assms comp-arr-dom invert-side-of-triangle(2) iso-is-arr comp-assoc
        by metis
      finally show ?thesis by simp
    qed
    show inv (g · f) = inv f · inv g
      using assms 1 comp-cod-arr comp-assoc
      by (metis arr-inv cod-inv dom-comp seqI invert-side-of-triangle(1) iso-is-arr)
    show iso g
      using assms 1 comp-assoc inv-is-inverse
      by (metis arr-inv invert-side-of-triangle(2) inv-inv iso-inv-iso isos-compose)
  qed

 end

end
```

Chapter 4

DualCategory

theory *DualCategory*
imports *EpiMonoIso*
begin

The locale defined here constructs the dual (opposite) of a category. The arrows of the dual category are directly identified with the arrows of the given category and simplification rules are introduced that automatically eliminate notions defined for the dual category in favor of the corresponding notions on the original category. This makes it easy to use the dual of a category in the same context as the category itself, without having to worry about whether an arrow belongs to the category or its dual.

locale *dual-category* =
  C: *category C*
for *C* :: *'a comp* (**infixr** ‹·› *55*)
begin

  definition *comp* (**infixr** ‹·^{op}› *55*)
  where *g* ·^{op} *f* ≡ *f* · *g*

  lemma *comp-char* [*simp*]:
  shows *g* ·^{op} *f* = *f* · *g*
    using *comp-def* **by** *auto*

  interpretation *partial-composition comp*
    apply *unfold-locales* **using** *comp-def C.ex-un-null* **by** *metis*

  notation *in-hom* (‹«- : - ← -»›)

  lemma *null-char* [*simp*]:
  shows *null* = *C.null*
    by (*metis C.null-is-zero*(*2*) *null-is-zero*(*2*) *comp-def*)

  lemma *ide-char* [*simp*]:
  shows *ide a* ⟷ *C.ide a*
    unfolding *ide-def C.ide-def* **by** *auto*

```
  lemma domains-char:
  shows domains f = C.codomains f
    using C.codomains-def domains-def ide-char by auto

  lemma codomains-char:
  shows codomains f = C.domains f
    using C.domains-def codomains-def ide-char by auto

  interpretation category comp
    using C.has-domain-iff-arr C.has-codomain-iff-arr domains-char codomains-char null-char
          comp-def C.match-4 C.ext arr-def C.comp-assoc
    apply (unfold-locales, auto)
    using C.match-2 by metis

  lemma is-category:
  shows category comp ..

end

sublocale dual-category ⊆ category comp
  using is-category by auto

context dual-category
begin

  lemma dom-char [simp]:
  shows dom f = C.cod f
    by (simp add: C.cod-def dom-def domains-char)

  lemma cod-char [simp]:
  shows cod f = C.dom f
    by (simp add: C.dom-def cod-def codomains-char)

  lemma arr-char [simp]:
  shows arr f ⟷ C.arr f
    using C.has-codomain-iff-arr has-domain-iff-arr domains-char by auto

  lemma hom-char [simp]:
  shows in-hom f b a ⟷ C.in-hom f a b
    by force

  lemma seq-char [simp]:
  shows seq g f = C.seq f g
    by simp

  lemma iso-char [simp]:
  shows iso f ⟷ C.iso f
    using C.iso-iff-section-and-retraction iso-iff-section-and-retraction
```

retraction-def section-def
by *auto*

end

end

Chapter 5

Concrete Categories

In this section we define a locale *concrete-category*, which provides a uniform (and more traditional) way to construct a category from specified sets of objects and arrows, with specified identity objects and composition of arrows. We prove that the identities and arrows of the constructed category are appropriately in bijective correspondence with the given sets and that domains, codomains, and composition in the constructed category are as expected according to this correspondence. In the later theory *Functor*, once we have defined functors and isomorphisms of categories, we will show a stronger property of this construction: if C is any category, then C is isomorphic to the concrete category formed from it in the obvious way by taking the identities of C as objects, the set of arrows of C as arrows, the identities of C as identity objects, and defining composition of arrows using the composition of C. Thus no information about C is lost by extracting its objects, arrows, identities, and composition and rebuilding it as a concrete category. We note, however, that we do not assume that the composition function given as parameter to the concrete category construction is "extensional", so in general it will contain incidental information about composition of non-composable arrows, and this information is not preserved by the concrete category construction.

theory *ConcreteCategory*
imports *Category*
begin

locale *concrete-category* =
fixes *Obj* :: *'o set*
and *Hom* :: *'o* ⇒ *'o* ⇒ *'a set*
and *Id* :: *'o* ⇒ *'a*
and *Comp* :: *'o* ⇒ *'o* ⇒ *'o* ⇒ *'a* ⇒ *'a* ⇒*'a*
assumes *Id-in-Hom*: *A* ∈ *Obj* ⟹ *Id A* ∈ *Hom A A*
and *Comp-in-Hom*: ⟦ *A* ∈ *Obj*; *B* ∈ *Obj*; *C* ∈ *Obj*; *f* ∈ *Hom A B*; *g* ∈ *Hom B C* ⟧
⟹ *Comp C B A g f* ∈ *Hom A C*
and *Comp-Hom-Id*: ⟦ *A* ∈ *Obj*; *B* ∈ *Obj*; *f* ∈ *Hom A B* ⟧ ⟹ *Comp B A A f* (*Id A*) = *f*
and *Comp-Id-Hom*: ⟦ *A* ∈ *Obj*; *B* ∈ *Obj*; *f* ∈ *Hom A B* ⟧ ⟹ *Comp B B A* (*Id B*) *f* = *f*
and *Comp-assoc*: ⟦ *A* ∈ *Obj*; *B* ∈ *Obj*; *C* ∈ *Obj*; *D* ∈ *Obj*;
f ∈ *Hom A B*; *g* ∈ *Hom B C*; *h* ∈ *Hom C D* ⟧ ⟹

```
                Comp D C A h (Comp C B A g f) = Comp D B A (Comp D C B h g) f
begin

  datatype ('oo, 'aa) arr =
    Null
  | MkArr 'oo 'oo 'aa

  abbreviation MkIde :: 'o ⇒ ('o, 'a) arr
  where MkIde A ≡ MkArr A A (Id A)

  fun Dom :: ('o, 'a) arr ⇒ 'o
  where Dom (MkArr A - -) = A
      | Dom - = undefined

  fun Cod
  where Cod (MkArr - B -) = B
      | Cod - = undefined

  fun Map
  where Map (MkArr - - F) = F
      | Map - = undefined

  abbreviation Arr
  where Arr f ≡ f ≠ Null ∧ Dom f ∈ Obj ∧ Cod f ∈ Obj ∧ Map f ∈ Hom (Dom f) (Cod f)

  abbreviation Ide
  where Ide a ≡ a ≠ Null ∧ Dom a ∈ Obj ∧ Cod a = Dom a ∧ Map a = Id (Dom a)


  definition COMP :: ('o, 'a) arr comp
  where COMP g f ≡ if Arr f ∧ Arr g ∧ Dom g = Cod f then
                     MkArr (Dom f) (Cod g) (Comp (Cod g) (Dom g) (Dom f) (Map g) (Map f))
                   else
                     Null

  interpretation partial-composition COMP
    using COMP-def by (unfold-locales, metis)

  lemma null-char:
  shows null = Null
  proof −
    let ?P = λn. ∀f. COMP n f = n ∧ COMP f n = n
    have Null = null
      using COMP-def null-def the1-equality [of ?P]
      by (metis (no-types, lifting))
    thus ?thesis by simp
  qed

  lemma ide-char_CC:
```

```
shows ide f ⟷ Ide f
proof
  assume f: Ide f
  show ide f
  proof −
    have COMP f f ≠ null
      using f COMP-def null-char Id-in-Hom by auto
    moreover have ∀ g. (COMP g f ≠ null ⟶ COMP g f = g) ∧
                       (COMP f g ≠ null ⟶ COMP f g = g)
    proof (intro allI conjI)
      fix g
      show COMP g f ≠ null ⟶ COMP g f = g
        using f COMP-def null-char Comp-Hom-Id Id-in-Hom
        by (cases g, auto)
      show COMP f g ≠ null ⟶ COMP f g = g
        using f COMP-def null-char Comp-Id-Hom Id-in-Hom
        by (cases g, auto)
    qed
    ultimately show ?thesis
      using ide-def by blast
  qed
  next
  assume f: ide f
  have 1: Arr f ∧ Dom f = Cod f
    using f ide-def COMP-def null-char by metis
  moreover have Map f = Id (Dom f)
  proof −
    let ?g = MkIde (Dom f)
    have g: Arr f ∧ Arr ?g ∧ Dom ?g = Cod f
      using 1 Id-in-Hom
      by (intro conjI, simp-all)
    have COMP ?g f = MkArr (Dom f) (Dom f) (Map f)
      using g COMP-def Comp-Id-Hom by auto
    moreover have COMP ?g f = ?g
    proof −
      have COMP ?g f ≠ null
        using g 1 COMP-def null-char by simp
      thus ?thesis
        using f ide-def by blast
    qed
    ultimately show ?thesis by simp
  qed
  ultimately show Ide f by auto
qed

lemma ide-MkIde [simp]:
assumes A ∈ Obj
shows ide (MkIde A)
  using assms ide-char_CC Id-in-Hom by simp
```

lemma *in-domains-char*:
shows $a \in domains\ f \longleftrightarrow Arr\ f \wedge a = MkIde\ (Dom\ f)$
proof
  assume *a*: $a \in domains\ f$
  have *Ide a*
    using *a domains-def ide-char*$_{CC}$ *COMP-def null-char* **by** *auto*
  moreover have $Arr\ f \wedge Dom\ f = Cod\ a$
  proof −
    have $COMP\ f\ a \neq null$
      using *a domains-def* **by** *simp*
    thus *?thesis*
      using *a domains-def COMP-def* [*of f a*] *null-char* **by** *metis*
  qed
  ultimately show $Arr\ f \wedge a = MkIde\ (Dom\ f)$
    by (*cases a*, *auto*)
  next
  assume *a*: $Arr\ f \wedge a = MkIde\ (Dom\ f)$
  show $a \in domains\ f$
    using *a Id-in-Hom COMP-def null-char domains-def* **by** *auto*
qed

lemma *in-codomains-char*:
shows $b \in codomains\ f \longleftrightarrow Arr\ f \wedge b = MkIde\ (Cod\ f)$
proof
  assume *b*: $b \in codomains\ f$
  have *Ide b*
    using *b codomains-def ide-char*$_{CC}$ *COMP-def null-char* **by** *auto*
  moreover have $Arr\ f \wedge Dom\ b = Cod\ f$
  proof −
    have $COMP\ b\ f \neq null$
      using *b codomains-def* **by** *simp*
    thus *?thesis*
      using *b codomains-def COMP-def* [*of b f*] *null-char* **by** *metis*
  qed
  ultimately show $Arr\ f \wedge b = MkIde\ (Cod\ f)$
    by (*cases b*, *auto*)
  next
  assume *b*: $Arr\ f \wedge b = MkIde\ (Cod\ f)$
  show $b \in codomains\ f$
    using *b Id-in-Hom COMP-def null-char codomains-def* **by** *auto*
qed

lemma *arr-char*:
shows $arr\ f \longleftrightarrow Arr\ f$
  using *arr-def in-domains-char in-codomains-char* **by** *auto*

lemma *arrI*$_{CC}$:
assumes $f \neq Null$ **and** $Dom\ f \in Obj\ Cod\ f \in Obj\ Map\ f \in Hom\ (Dom\ f)\ (Cod\ f)$

shows *arr f*
using *assms arr-char* **by** *blast*

lemma *arrE*:
assumes *arr f*
and ⟦ *f* ≠ *Null*; *Dom f* ∈ *Obj*; *Cod f* ∈ *Obj*; *Map f* ∈ *Hom* (*Dom f*) (*Cod f*) ⟧ ⟹ *T*
shows *T*
using *assms arr-char* **by** *simp*

lemma *arr-MkArr* [*simp*]:
assumes *A* ∈ *Obj* **and** *B* ∈ *Obj* **and** *f* ∈ *Hom A B*
shows *arr* (*MkArr A B f*)
using *assms arr-char* **by** *simp*

lemma *MkArr-Map*:
assumes *arr f*
shows *MkArr* (*Dom f*) (*Cod f*) (*Map f*) = *f*
using *assms arr-char* **by** (*cases f*, *auto*)

lemma *Arr-comp*:
assumes *arr f* **and** *arr g* **and** *Dom g* = *Cod f*
shows *Arr* (*COMP g f*)
unfolding *COMP-def*
using *assms arr-char Comp-in-Hom* **by** *simp*

lemma *Dom-comp* [*simp*]:
assumes *arr f* **and** *arr g* **and** *Dom g* = *Cod f*
shows *Dom* (*COMP g f*) = *Dom f*
unfolding *COMP-def*
using *assms arr-char* **by** *simp*

lemma *Cod-comp* [*simp*]:
assumes *arr f* **and** *arr g* **and** *Dom g* = *Cod f*
shows *Cod* (*COMP g f*) = *Cod g*
unfolding *COMP-def*
using *assms arr-char* **by** *simp*

lemma *Map-comp* [*simp*]:
assumes *arr f* **and** *arr g* **and** *Dom g* = *Cod f*
shows *Map* (*COMP g f*) = *Comp* (*Cod g*) (*Dom g*) (*Dom f*) (*Map g*) (*Map f*)
unfolding *COMP-def*
using *assms arr-char* **by** *simp*

lemma *seq-char*:
shows *seq g f* ⟷ *arr f* ∧ *arr g* ∧ *Dom g* = *Cod f*
using *arr-char not-arr-null null-char COMP-def Arr-comp* **by** *metis*

interpretation *category COMP*
proof

show $\bigwedge g\ f.\ COMP\ g\ f \neq null \Longrightarrow seq\ g\ f$
using *arr-char COMP-def null-char Comp-in-Hom* **by** *auto*
show *1*: $\bigwedge f.\ (domains\ f \neq \{\}) = (codomains\ f \neq \{\})$
using *in-domains-char in-codomains-char* **by** *auto*
show $\bigwedge f\ g\ h.\ seq\ h\ g \Longrightarrow seq\ (COMP\ h\ g)\ f \Longrightarrow seq\ g\ f$
by (*auto simp add*: *seq-char*)
show $\bigwedge f\ g\ h.\ seq\ h\ (COMP\ g\ f) \Longrightarrow seq\ g\ f \Longrightarrow seq\ h\ g$
using *seq-char COMP-def Comp-in-Hom* **by** (*metis Cod-comp*)
show $\bigwedge f\ g\ h.\ seq\ g\ f \Longrightarrow seq\ h\ g \Longrightarrow seq\ (COMP\ h\ g)\ f$
using *Comp-in-Hom*
by (*auto simp add*: *COMP-def seq-char*)
show $\bigwedge g\ f\ h.\ seq\ g\ f \Longrightarrow seq\ h\ g \Longrightarrow COMP\ (COMP\ h\ g)\ f = COMP\ h\ (COMP\ g\ f)$
using *seq-char COMP-def Comp-assoc Comp-in-Hom Dom-comp Cod-comp Map-comp*
by *auto*
qed

proposition *is-category*:
shows *category COMP*
..

Functions *Dom*, *Cod*, and *Map* establish a correspondence between the arrows of the constructed category and the elements of the originally given parameters *Obj* and *Hom*.

lemma *Dom-in-Obj*:
assumes *arr f*
shows *Dom f* ∈ *Obj*
using *assms arr-char* **by** *simp*

lemma *Cod-in-Obj*:
assumes *arr f*
shows *Cod f* ∈ *Obj*
using *assms arr-char* **by** *simp*

lemma *Map-in-Hom*:
assumes *arr f*
shows *Map f* ∈ *Hom* (*Dom f*) (*Cod f*)
using *assms arr-char* **by** *simp*

lemma *MkArr-in-hom* [*intro*]:
assumes *A* ∈ *Obj* **and** *B* ∈ *Obj* **and** *f* ∈ *Hom A B* **and** *a* = *MkIde A* **and** *b* = *MkIde B*
shows *in-hom* (*MkArr A B f*) *a b*
using *assms arr-char ide-MkIde*
by (*simp add*: *in-codomains-char in-domains-char in-hom-def*)

The next few results show that domains, codomains, and composition in the constructed category are as expected according to the just-given correspondence.

lemma *dom-char*:
shows *dom f* = (*if arr f then MkIde* (*Dom f*) *else null*)
using *dom-def in-domains-char dom-in-domains has-domain-iff-arr* **by** *auto*

lemma *cod-char*:
shows *cod f* = (*if arr f then MkIde* (*Cod f*) *else null*)
using *cod-def in-codomains-char cod-in-codomains has-codomain-iff-arr* **by** *auto*

lemma *comp-char*:
shows *COMP g f* = (*if seq g f then*
MkArr (*Dom f*) (*Cod g*) (*Comp* (*Cod g*) (*Dom g*) (*Dom f*) (*Map g*) (*Map f*))
else
null)
using *COMP-def seq-char arr-char null-char* **by** *auto*

lemma *in-hom-char*:
shows *in-hom f a b* ⟷ *arr f* ∧ *ide a* ∧ *ide b* ∧ *Dom f* = *Dom a* ∧ *Cod f* = *Dom b*
proof
show *in-hom f a b* ⟹ *arr f* ∧ *ide a* ∧ *ide b* ∧ *Dom f* = *Dom a* ∧ *Cod f* = *Dom b*
using *arr-char dom-char cod-char* **by** *auto*
show *arr f* ∧ *ide a* ∧ *ide b* ∧ *Dom f* = *Dom a* ∧ *Cod f* = *Dom b* ⟹ *in-hom f a b*
using *arr-char dom-char cod-char ide-char Id-in-Hom MkArr-Map in-homI* **by** *metis*
qed

lemma *Dom-dom* [*simp*]:
assumes *arr f*
shows *Dom* (*dom f*) = *Dom f*
using *assms MkArr-Map dom-char* **by** *simp*

lemma *Cod-dom* [*simp*]:
assumes *arr f*
shows *Cod* (*dom f*) = *Dom f*
using *assms MkArr-Map dom-char* **by** *simp*

lemma *Dom-cod* [*simp*]:
assumes *arr f*
shows *Dom* (*cod f*) = *Cod f*
using *assms MkArr-Map cod-char* **by** *simp*

lemma *Cod-cod* [*simp*]:
assumes *arr f*
shows *Cod* (*cod f*) = *Cod f*
using *assms MkArr-Map cod-char* **by** *simp*

lemma *Map-dom* [*simp*]:
assumes *arr f*
shows *Map* (*dom f*) = *Id* (*Dom f*)
using *assms MkArr-Map dom-char* **by** *simp*

lemma *Map-cod* [*simp*]:
assumes *arr f*
shows *Map* (*cod f*) = *Id* (*Cod f*)
using *assms MkArr-Map cod-char* **by** *simp*

lemma *Map-ide*:
assumes *ide a*
shows *Map a* = *Id* (*Dom a*) **and** *Map a* = *Id* (*Cod a*)
using *assms ide-char dom-char* [*of a*] *Map-dom Map-cod ideD*(*1*) **by** *metis*+

lemma *MkIde-Dom*:
assumes *arr a*
shows *MkIde* (*Dom a*) = *dom a*
using *assms arr-char dom-char* **by** (*cases a*, *auto*)

lemma *MkIde-Cod*:
assumes *arr a*
shows *MkIde* (*Cod a*) = *cod a*
using *assms arr-char cod-char* **by** (*cases a*, *auto*)

lemma *MkIde-Dom′* [*simp*]:
assumes *ide a*
shows *MkIde* (*Dom a*) = *a*
using *assms MkIde-Dom* **by** *simp*

lemma *MkIde-Cod′* [*simp*]:
assumes *ide a*
shows *MkIde* (*Cod a*) = *a*
using *assms MkIde-Cod* **by** *simp*

lemma *dom-MkArr* [*simp*]:
assumes *arr* (*MkArr A B F*)
shows *dom* (*MkArr A B F*) = *MkIde A*
using *assms dom-char* **by** *simp*

lemma *cod-MkArr* [*simp*]:
assumes *arr* (*MkArr A B F*)
shows *cod* (*MkArr A B F*) = *MkIde B*
using *assms cod-char* **by** *simp*

lemma *comp-MkArr* [*simp*]:
assumes *arr* (*MkArr A B F*) **and** *arr* (*MkArr B C G*)
shows *COMP* (*MkArr B C G*) (*MkArr A B F*) = *MkArr A C* (*Comp C B A G F*)
using *assms comp-char* [*of MkArr B C G MkArr A B F*] **by** *simp*

The set *Obj* of "objects" given as a parameter is in bijective correspondence (via function *MkIde*) with the set of identities of the resulting category.

proposition *bij-betw-ide-Obj*:
shows *MkIde* ∈ *Obj* → *Collect ide*
and *Dom* ∈ *Collect ide* → *Obj*
and *A* ∈ *Obj* ⟹ *Dom* (*MkIde A*) = *A*
and *a* ∈ *Collect ide* ⟹ *MkIde* (*Dom a*) = *a*

and *bij-betw Dom* (*Collect ide*) *Obj*
proof −
show *MkIde* ∈ *Obj* → *Collect ide*
using *ide-MkIde* **by** *simp*
moreover show *Dom* ∈ *Collect ide* → *Obj*
using *arr-char ideD*(*1*) **by** *simp*
moreover show ⋀*A*. *A* ∈ *Obj* ⟹ *Dom* (*MkIde A*) = *A*
by *simp*
moreover show ⋀*a*. *a* ∈ *Collect ide* ⟹ *MkIde* (*Dom a*) = *a*
using *MkIde-Dom* **by** *simp*
ultimately show *bij-betw Dom* (*Collect ide*) *Obj*
using *bij-betwI* **by** *blast*
qed

For each pair of identities *a* and *b*, the set *Hom* (*Dom a*) (*Dom b*) is in bijective correspondence (via function *MkArr* (*Dom a*) (*Dom b*)) with the "hom-set" *hom a b* of the resulting category.

proposition *bij-betw-hom-Hom*:
assumes *ide a* **and** *ide b*
shows *Map* ∈ *hom a b* → *Hom* (*Dom a*) (*Dom b*)
and *MkArr* (*Dom a*) (*Dom b*) ∈ *Hom* (*Dom a*) (*Dom b*) → *hom a b*
and ⋀*f*. *f* ∈ *hom a b* ⟹ *MkArr* (*Dom a*) (*Dom b*) (*Map f*) = *f*
and ⋀*F*. *F* ∈ *Hom* (*Dom a*) (*Dom b*) ⟹ *Map* (*MkArr* (*Dom a*) (*Dom b*) *F*) = *F*
and *bij-betw Map* (*hom a b*) (*Hom* (*Dom a*) (*Dom b*))
proof −
show *Map* ∈ *hom a b* → *Hom* (*Dom a*) (*Dom b*)
using *Map-in-Hom cod-char dom-char in-hom-char* **by** *fastforce*
moreover show *MkArr* (*Dom a*) (*Dom b*) ∈ *Hom* (*Dom a*) (*Dom b*) → *hom a b*
using *assms Dom-in-Obj MkArr-in-hom* [*of Dom a Dom b*] **by** *simp*
moreover show ⋀*f*. *f* ∈ *hom a b* ⟹ *MkArr* (*Dom a*) (*Dom b*) (*Map f*) = *f*
using *MkArr-Map* **by** *auto*
moreover show ⋀*F*. *F* ∈ *Hom* (*Dom a*) (*Dom b*)
⟹ *Map* (*MkArr* (*Dom a*) (*Dom b*) *F*) = *F*
by *simp*
ultimately show *bij-betw Map* (*hom a b*) (*Hom* (*Dom a*) (*Dom b*))
using *bij-betwI* **by** *blast*
qed

lemma *arr-eqI*:
assumes *arr t* **and** *arr t′* **and** *Dom t* = *Dom t′* **and** *Cod t* = *Cod t′* **and** *Map t* = *Map t′*
shows *t* = *t′*
using *assms MkArr-Map* **by** *metis*

end

sublocale *concrete-category* ⊆ *category COMP*
using *is-category* **by** *auto*

end

Chapter 6

InitialTerminal

theory *InitialTerminal*
imports *EpiMonoIso*
begin

This theory defines the notions of initial and terminal object in a category and establishes some properties of these notions, including that when they exist they are unique up to isomorphism.

context *category*
begin

definition *initial*
where *initial a* ≡ *ide a* ∧ (∀ *b*. *ide b* ⟶ (∃!*f*. «*f* : *a* → *b*»))

definition *terminal*
where *terminal b* ≡ *ide b* ∧ (∀ *a*. *ide a* ⟶ (∃!*f*. «*f* : *a* → *b*»))

abbreviation *initial-arr*
where *initial-arr f* ≡ *arr f* ∧ *initial* (*dom f*)

abbreviation *terminal-arr*
where *terminal-arr f* ≡ *arr f* ∧ *terminal* (*cod f*)

abbreviation *point*
where *point f* ≡ *arr f* ∧ *terminal* (*dom f*)

lemma *initial-arr-unique*:
assumes *par f f′* **and** *initial-arr f* **and** *initial-arr f′*
shows *f* = *f′*
using *assms in-homI initial-def ide-cod* **by** *blast*

lemma *initialI* [*intro*]:
assumes *ide a* **and** ⋀*b*. *ide b* ⟹ ∃!*f*. «*f* : *a* → *b*»
shows *initial a*
using *assms initial-def* **by** *auto*

lemma *initialE* [*elim*]:
assumes *initial a* **and** *ide b*
obtains *f* **where** «*f* : *a* → *b*» **and** ⋀*f'*. «*f'* : *a* → *b*» ⟹ *f'* = *f*
using *assms initial-def initial-arr-unique* **by** *meson*

lemma *terminal-arr-unique*:
assumes *par f f'* **and** *terminal-arr f* **and** *terminal-arr f'*
shows *f* = *f'*
using *assms in-homI terminal-def ide-dom* **by** *blast*

lemma *terminalI* [*intro*]:
assumes *ide b* **and** ⋀*a*. *ide a* ⟹ ∃!*f*. «*f* : *a* → *b*»
shows *terminal b*
using *assms terminal-def* **by** *auto*

lemma *terminalE* [*elim*]:
assumes *terminal b* **and** *ide a*
obtains *f* **where** «*f* : *a* → *b*» **and** ⋀*f'*. «*f'* : *a* → *b*» ⟹ *f'* = *f*
using *assms terminal-def terminal-arr-unique* **by** *meson*

lemma *terminal-objs-isomorphic*:
assumes *terminal a* **and** *terminal b*
shows *isomorphic a b*
proof −
from *assms* **obtain** *f* **where** *f*: «*f* : *a* → *b*»
using *terminal-def* **by** *meson*
from *assms* **obtain** *g* **where** *g*: «*g* : *b* → *a*»
using *terminal-def* **by** *meson*
have *iso f*
using *assms f g*
by (*metis arr-iff-in-hom cod-comp retractionI sectionI seqI' terminal-def*
dom-comp in-homE iso-iff-section-and-retraction ide-in-hom)
thus *?thesis* **using** *f* **by** *auto*
qed

lemma *isomorphic-to-terminal-is-terminal*:
assumes *terminal a* **and** *isomorphic a a'*
shows *terminal a'*
proof
show *ide a'*
using *assms terminal-def isomorphic-def* **by** *auto*
obtain *h h'* **where** *h*: «*h* : *a* → *a'*» ∧ *inverse-arrows h h'*
using *assms isomorphic-def* **by** *auto*
fix *b*
assume *b*: *ide b*
obtain *t* **where** *t*: «*t* : *b* → *a*»
using *assms b* **by** *blast*
show ∃!*f*. «*f* : *b* → *a'*»

```
  proof
    show «h · t : b → a'»
      using h t by blast
    show ⋀f. «f : b → a'» ⟹ f = h · t
    proof −
      fix f
      assume f: «f : b → a'»
      have «h' · f : b → a»
        by (metis f h inv-in-hom inverse-unique comp-in-homI isoI)
      hence h · h' · f = h · t
        using assms f h terminal-def
        by (metis t terminal-arr-unique in-homE)
      thus f = h · t
        by (metis f h comp-arr-inv' comp-cod-arr inverse-unique in-homE isoI comp-assoc)
    qed
  qed
qed

lemma initial-objs-isomorphic:
assumes initial a and initial b
shows isomorphic a b
proof −
  from assms obtain f where f: «f : a → b» using initial-def by auto
  from assms obtain g where g: «g : b → a» using initial-def by auto
  have iso f
    using assms f g
    by (metis (no-types, lifting) arr-iff-in-hom cod-comp in-homE initial-def
        retractionI sectionI dom-comp iso-iff-section-and-retraction ide-in-hom seqI')
  thus ?thesis
    using f by auto
qed

lemma isomorphic-to-initial-is-initial:
assumes initial a and isomorphic a a'
shows initial a'
proof
  show ide a'
    using assms initial-def isomorphic-def by auto
  obtain h h' where h: «h : a' → a» ∧ inverse-arrows h h'
    using assms isomorphic-def
    by (meson isoE isomorphic-symmetric)
  fix b
  assume b: ide b
  obtain i where i: «i : a → b»
    using assms b by blast
  show ∃!f. «f : a' → b»
  proof
    show «i · h : a' → b»
      using h i by blast
```

```
      show ⋀f. «f : a' → b» ⟹ f = i · h
      proof −
        fix f
        assume f: «f : a' → b»
        have «f · h' : a → b»
          by (metis f h inv-in-hom inverse-unique comp-in-homI isoI)
        hence f · h' · h = i · h
          using assms f h initial-def comp-assoc
          by (metis i initial-arr-unique in-homE)
        thus f = i · h
          by (metis f h comp-inv-arr' comp-arr-dom inverse-unique in-homE isoI)
      qed
    qed

    lemma point-is-mono:
    assumes point f
    shows mono f
    proof −
      have ide (cod f) using assms by auto
      from this obtain t where t: «t: cod f → dom f»
        using assms terminal-def by blast
      thus ?thesis
        using assms terminal-def monoI
        by (metis seqE in-homI dom-comp ide-dom terminal-def)
    qed

  end

end
```

Chapter 7

Functor

theory *Functor*
imports *Category ConcreteCategory DualCategory InitialTerminal*
begin

One advantage of the "object-free" definition of category is that a functor from category A to category B is simply a function from the type of arrows of A to the type of arrows of B that satisfies certain conditions: namely, that arrows are mapped to arrows, non-arrows are mapped to *null*, and domains, codomains, and composition of arrows are preserved.

locale *functor* =
  A: *category* A +
  B: *category* B
for A :: $'a$ *comp* (**infixr** $\langle\cdot_A\rangle$ *55*)
and B :: $'b$ *comp* (**infixr** $\langle\cdot_B\rangle$ *55*)
and F :: $'a \Rightarrow 'b$ +
assumes *extensionality*: $\neg A.arr\ f \Longrightarrow F\ f = B.null$
and *preserves-arr*: $A.arr\ f \Longrightarrow B.arr\ (F\ f)$
and *preserves-dom* [*iff*]: $A.arr\ f \Longrightarrow B.dom\ (F\ f) = F\ (A.dom\ f)$
and *preserves-cod* [*iff*]: $A.arr\ f \Longrightarrow B.cod\ (F\ f) = F\ (A.cod\ f)$
and *preserves-comp* [*iff*]: $A.seq\ g\ f \Longrightarrow F\ (g\ \cdot_A\ f) = F\ g\ \cdot_B\ F\ f$
begin

  notation *A.in-hom* ($\langle\langle$- : - $\rightarrow_A$ -$\rangle\rangle$)
  notation *B.in-hom* ($\langle\langle$- : - $\rightarrow_B$ -$\rangle\rangle$)

  lemma *preserves-hom* [*intro*]:
  assumes $\langle\langle f : a \rightarrow_A b\rangle\rangle$
  shows $\langle\langle F\ f : F\ a \rightarrow_B F\ b\rangle\rangle$
    using *assms B.in-homI*
    by (*metis A.in-homE preserves-arr preserves-cod preserves-dom*)

The following, which is made possible through the presence of *null*, allows us to infer that the subterm f denotes an arrow if the term $F\ f$ denotes an arrow. This is very useful, because otherwise doing anything with f would require a separate proof that it is

an arrow by some other means.

lemma *preserves-reflects-arr* [*iff*]:
shows *B.arr* (*F f*) ⟷ *A.arr f*
using *preserves-arr extensionality B.not-arr-null* **by** *metis*

lemma *preserves-seq* [*intro*]:
assumes *A.seq g f*
shows *B.seq* (*F g*) (*F f*)
using *assms* **by** *auto*

lemma *preserves-ide* [*simp*]:
assumes *A.ide a*
shows *B.ide* (*F a*)
using *assms A.ide-in-hom B.ide-in-hom* **by** *auto*

lemma *preserves-iso* [*simp*]:
assumes *A.iso f*
shows *B.iso* (*F f*)
using *assms A.inverse-arrowsE*
apply (*elim A.isoE A.inverse-arrowsE A.seqE A.ide-compE*)
by (*metis A.arr-dom-iff-arr B.ide-dom B.inverse-arrows-def B.isoI preserves-arr*
preserves-comp preserves-dom)

lemma *preserves-isomorphic*:
assumes *A.isomorphic a b*
shows *B.isomorphic* (*F a*) (*F b*)
by (*meson A.isomorphic-def B.isomorphic-def assms preserves-hom preserves-iso*)

lemma *preserves-section-retraction*:
assumes *A.ide* (*A e m*)
shows *B.ide* (*B* (*F e*) (*F m*))
using *assms* **by** (*metis A.ide-compE preserves-comp preserves-ide*)

lemma *preserves-section*:
assumes *A.section m*
shows *B.section* (*F m*)
using *assms preserves-section-retraction* **by** *blast*

lemma *preserves-retraction*:
assumes *A.retraction e*
shows *B.retraction* (*F e*)
using *assms preserves-section-retraction* **by** *blast*

lemma *preserves-inverse-arrows*:
assumes *A.inverse-arrows f g*
shows *B.inverse-arrows* (*F f*) (*F g*)
using *assms A.inverse-arrows-def B.inverse-arrows-def preserves-section-retraction*
by *simp*

```
  lemma preserves-inv:
  assumes A.iso f
  shows F (A.inv f) = B.inv (F f)
    using assms preserves-inverse-arrows A.inv-is-inverse B.inv-is-inverse
          B.inverse-arrow-unique
    by blast

  lemma preserves-iso-in-hom [intro]:
  assumes A.iso-in-hom f a b
  shows B.iso-in-hom (F f) (F a) (F b)
    using assms preserves-hom preserves-iso by blast

end

locale endofunctor =
  functor A A F
for A :: 'a comp      (infixr ‹·› 55)
and F :: 'a ⇒ 'a

locale faithful-functor = functor A B F
for A :: 'a comp
and B :: 'b comp
and F :: 'a ⇒ 'b +
assumes is-faithful: ⟦ A.par f f'; F f = F f' ⟧ ⟹ f = f'
begin

  lemma locally-reflects-ide:
  assumes «f : a →A a» and B.ide (F f)
  shows A.ide f
    using assms is-faithful
    by (metis A.arr-dom-iff-arr A.cod-dom A.dom-dom A.in-homE B.comp-ide-self
        B.ide-self-inverse B.comp-arr-inv A.ide-cod preserves-dom)

end

locale full-functor = functor A B F
for A :: 'a comp
and B :: 'b comp
and F :: 'a ⇒ 'b +
assumes is-full: ⟦ A.ide a; A.ide a'; «g : F a' →B F a» ⟧ ⟹ ∃f. «f : a' →A a» ∧ F f = g

locale fully-faithful-functor =
  faithful-functor A B F +
  full-functor A B F
for A :: 'a comp
and B :: 'b comp
and F :: 'a ⇒ 'b
begin
```

```
  lemma reflects-iso:
  assumes «f : a' →_A a» and B.iso (F f)
  shows A.iso f
  proof −
    from assms obtain g' where g': B.inverse-arrows (F f) g' by blast
    have 1: «g' : F a →_B F a'»
      using assms g' by (metis B.inv-in-hom B.inverse-unique preserves-hom)
    from this obtain g where g: «g : a →_A a'» ∧ F g = g'
      using assms(1) is-full by (metis A.arrI A.ide-cod A.ide-dom A.in-homE)
    have A.inverse-arrows f g
      using assms 1 g g' A.inverse-arrowsI
      by (metis A.arr-iff-in-hom A.dom-comp A.in-homE A.seqI' B.inverse-arrowsE
          A.cod-comp locally-reflects-ide preserves-comp)
    thus ?thesis by auto
  qed

  lemma reflects-isomorphic:
  assumes A.ide f and A.ide f' and B.isomorphic (F f) (F f')
  shows A.isomorphic f f'
    by (metis A.isomorphic-def B.isomorphicE assms(1−3) is-full reflects-iso)

end

locale embedding-functor = functor A B F
for A :: 'a comp
and B :: 'b comp
and F :: 'a ⇒ 'b +
assumes is-embedding: ⟦ A.arr f; A.arr f'; F f = F f' ⟧ ⟹ f = f'

sublocale embedding-functor ⊆ faithful-functor
  using is-embedding by (unfold-locales, blast)

context embedding-functor
begin

  lemma reflects-ide:
  assumes B.ide (F f)
  shows A.ide f
    using assms is-embedding A.ide-in-hom B.ide-in-hom
    by (metis A.in-homE B.in-homE A.ide-cod preserves-cod preserves-reflects-arr)

end

locale full-embedding-functor =
  embedding-functor A B F +
  full-functor A B F
for A :: 'a comp
and B :: 'b comp
and F :: 'a ⇒ 'b
```

locale *essentially-surjective-functor* = *functor* +
assumes *essentially-surjective*: $\bigwedge b.\ B.ide\ b \Longrightarrow \exists a.\ A.ide\ a \wedge B.isomorphic\ (F\ a)\ b$

locale *constant-functor* =
  A: *category A* +
  B: *category B*
for *A* :: *'a comp*
and *B* :: *'b comp*
and *b* :: *'b* +
assumes *value-is-ide*: *B.ide b*
begin

  definition *map*
  where *map f* = (*if A.arr f then b else B.null*)

  lemma *map-simp* [*simp*]:
  assumes *A.arr f*
  shows *map f* = *b*
    using *assms map-def* **by** *auto*

  lemma *is-functor*:
  shows *functor A B map*
    using *map-def value-is-ide* **by** (*unfold-locales*, *auto*)

end

sublocale *constant-functor* ⊆ *functor A B map*
  using *is-functor* **by** *auto*

locale *identity-functor* =
  C: *category C*
  for *C* :: *'a comp*
begin

  definition *map* :: *'a* ⇒ *'a*
  where *map f* = (*if C.arr f then f else C.null*)

  lemma *map-simp* [*simp*]:
  assumes *C.arr f*
  shows *map f* = *f*
    using *assms map-def* **by** *simp*

  sublocale *functor C C map*
    using *C.arr-dom-iff-arr C.arr-cod-iff-arr*
    by (*unfold-locales*; *auto simp add*: *map-def*)

  lemma *is-functor*:
  shows *functor C C map*

..

sublocale *fully-faithful-functor C C map*
using *C.arrI* **by** *unfold-locales auto*

lemma *is-fully-faithful*:
shows *fully-faithful-functor C C map*
..

end

It is convenient to have an easy way to obtain from a category the identity functor on that category. The following declaration causes the definitions and facts from the *identity-functor* locale to be inherited by the *category* locale, including the function *map* on arrows that represents the identity functor. This makes it generally unnecessary to give explicit interpretations of *identity-functor*.

sublocale *category* $\subseteq$ *identity-functor C* **..**

Composition of functors coincides with function composition, thanks to the magic of *null*.

lemma *functor-comp*:
assumes *functor A B F* **and** *functor B C G*
shows *functor A C* (*G o F*)
proof −
interpret *F*: *functor A B F* **using** *assms*(*1*) **by** *auto*
interpret *G*: *functor B C G* **using** *assms*(*2*) **by** *auto*
show *functor A C* (*G o F*)
using *F.preserves-arr F.extensionality G.extensionality* **by** (*unfold-locales*, *auto*)
qed

locale *composite-functor* =
F: *functor A B F* +
G: *functor B C G*
for *A* :: *'a comp*
and *B* :: *'b comp*
and *C* :: *'c comp*
and *F* :: *'a* ⇒ *'b*
and *G* :: *'b* ⇒ *'c*
begin

abbreviation *map*
where *map* ≡ *G o F*

sublocale *functor A C* ‹*G o F*›
using *functor-comp F.functor-axioms G.functor-axioms* **by** *blast*

lemma *is-functor*:
shows *functor A C* (*G o F*)
..

end

lemma *comp-functor-identity* [*simp*]:
assumes *functor A B F*
shows *F o identity-functor.map A = F*
proof
  interpret *functor A B F* **using** *assms* **by** *blast*
  show $\bigwedge x$. *(F o A.map) x = F x*
    using *A.map-def extensionality* **by** *simp*
qed

lemma *comp-identity-functor* [*simp*]:
assumes *functor A B F*
shows *identity-functor.map B o F = F*
proof
  interpret *functor A B F* **using** *assms* **by** *blast*
  show $\bigwedge x$. *(B.map o F) x = F x*
    using *B.map-def* **by** (*metis comp-apply extensionality preserves-arr*)
qed

lemma *faithful-functors-compose*:
assumes *faithful-functor A B F* **and** *faithful-functor B C G*
shows *faithful-functor A C (G o F)*
proof −
  interpret *F*: *faithful-functor A B F*
    using *assms*(*1*) **by** *simp*
  interpret *G*: *faithful-functor B C G*
    using *assms*(*2*) **by** *simp*
  interpret *composite-functor A B C F G* **..**
  show *faithful-functor A C (G o F)*
  proof
    show $\bigwedge f\ f'$. $[\![$*F.A.par f f'*; *map f = map f'*$]\!] \Longrightarrow$ *f = f'*
      using *F.is-faithful G.is-faithful*
      by (*metis* (*mono-tags*, *lifting*) *F.preserves-arr F.preserves-cod F.preserves-dom o-apply*)
  qed
qed

lemma *full-functors-compose*:
assumes *full-functor A B F* **and** *full-functor B C G*
shows *full-functor A C (G o F)*
proof −
  interpret *F*: *full-functor A B F*
    using *assms*(*1*) **by** *simp*
  interpret *G*: *full-functor B C G*
    using *assms*(*2*) **by** *simp*
  interpret *composite-functor A B C F G* **..**
  show *full-functor A C (G o F)*
  proof

```
    show ⋀a a' g. ⟦F.A.ide a; F.A.ide a'; «g : map a' → map a»⟧
                  ⟹ ∃f. F.A.in-hom f a' a ∧ map f = g
      using F.is-full G.is-full
      by (metis F.preserves-ide o-apply)
  qed
qed

lemma fully-faithful-functors-compose:
assumes fully-faithful-functor A B F and fully-faithful-functor B C G
shows full-functor A C (G o F)
proof −
  interpret F: fully-faithful-functor A B F
    using assms(1) by simp
  interpret G: fully-faithful-functor B C G
    using assms(2) by simp
  interpret composite-functor A B C F G ..
  interpret faithful-functor A C ‹G o F›
    using F.faithful-functor-axioms G.faithful-functor-axioms faithful-functors-compose
    by blast
  interpret full-functor A C ‹G o F›
    using F.full-functor-axioms G.full-functor-axioms full-functors-compose
    by blast
  show full-functor A C (G o F) ..
qed

lemma embedding-functors-compose:
assumes embedding-functor A B F and embedding-functor B C G
shows embedding-functor A C (G o F)
proof −
  interpret F: embedding-functor A B F
    using assms(1) by simp
  interpret G: embedding-functor B C G
    using assms(2) by simp
  interpret composite-functor A B C F G ..
  show embedding-functor A C (G o F)
  proof
    show ⋀f f'. ⟦F.A.arr f; F.A.arr f'; map f = map f'⟧ ⟹ f = f'
      by (simp add: F.is-embedding G.is-embedding)
  qed
qed

lemma full-embedding-functors-compose:
assumes full-embedding-functor A B F and full-embedding-functor B C G
shows full-embedding-functor A C (G o F)
proof −
  interpret F: full-embedding-functor A B F
    using assms(1) by simp
  interpret G: full-embedding-functor B C G
    using assms(2) by simp
```

```
  interpret composite-functor A B C F G ..
  interpret embedding-functor A C ‹G o F›
  using F.embedding-functor-axioms G.embedding-functor-axioms embedding-functors-compose
    by blast
  interpret full-functor A C ‹G o F›
    using F.full-functor-axioms G.full-functor-axioms full-functors-compose
    by blast
  show full-embedding-functor A C (G o F) ..
qed

lemma essentially-surjective-functors-compose:
assumes essentially-surjective-functor A B F and essentially-surjective-functor B C G
shows essentially-surjective-functor A C (G o F)
proof −
  interpret F: essentially-surjective-functor A B F
    using assms(1) by simp
  interpret G: essentially-surjective-functor B C G
    using assms(2) by simp
  interpret composite-functor A B C F G ..
  show essentially-surjective-functor A C (G o F)
  proof
    show ⋀c. G.B.ide c ⟹ ∃ a. F.A.ide a ∧ G.B.isomorphic (map a) c
      by (metis F.essentially-surjective G.B.isomorphic-transitive
          G.essentially-surjective G.preserves-isomorphic comp-def)
  qed
qed

locale inverse-functors =
  A: category A +
  B: category B +
  F: functor B A F +
  G: functor A B G
for A :: 'a comp      (infixr ‹·A› 55)
and B :: 'b comp      (infixr ‹·B› 55)
and F :: 'b ⇒ 'a
and G :: 'a ⇒ 'b +
assumes inv: G o F = identity-functor.map B
and inv': F o G = identity-functor.map A
begin

  lemma bij-betw-arr-sets:
  shows bij-betw F (Collect B.arr) (Collect A.arr)
    using inv inv'
    apply (intro bij-betwI)
      apply auto
    using comp-eq-dest-lhs by force+

end
```

locale *isomorphic-categories* =
  A: *category A* +
  B: *category B*
for *A* :: *'a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_B$› *55*) +
assumes *iso*: ∃ *F G*. *inverse-functors A B F G*

sublocale *inverse-functors* ⊆ *isomorphic-categories A B*
  using *inverse-functors-axioms* **by** (*unfold-locales*, *auto*)

lemma *inverse-functors-sym*:
assumes *inverse-functors A B F G*
shows *inverse-functors B A G F*
proof −
  interpret *inverse-functors A B F G* **using** *assms* **by** *auto*
  show *?thesis* **using** *inv inv'* **by** (*unfold-locales*, *auto*)
qed

Inverse functors uniquely determine each other.

lemma *inverse-functor-unique*:
assumes *inverse-functors C D F G* **and** *inverse-functors C D F G'*
shows *G* = *G'*
proof −
  interpret *FG*: *inverse-functors C D F G* **using** *assms*(*1*) **by** *auto*
  interpret *FG'*: *inverse-functors C D F G'* **using** *assms*(*2*) **by** *auto*
  show *G* = *G'*
    using *FG.G.extensionality FG'.G.extensionality FG'.inv FG.inv'*
    by (*metis FG'.G.functor-axioms FG.G.functor-axioms comp-assoc comp-identity-functor*
        comp-functor-identity)
qed

lemma *inverse-functor-unique'*:
assumes *inverse-functors C D F G* **and** *inverse-functors C D F' G*
shows *F* = *F'*
  using *assms inverse-functors-sym inverse-functor-unique* **by** *blast*

locale *invertible-functor* =
  A: *category A* +
  B: *category B* +
  G: *functor A B G*
for *A* :: *'a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_B$› *55*)
and *G* :: *'a* ⇒ *'b* +
assumes *invertible*: ∃ *F*. *inverse-functors A B F G*
begin

  lemma *has-unique-inverse*:
  shows ∃!*F*. *inverse-functors A B F G*
    using *invertible inverse-functor-unique'* **by** *blast*

definition *inv*
where *inv* ≡ *THE F*. *inverse-functors A B F G*

interpretation *inverse-functors A B inv G*
using *inv-def has-unique-inverse theI'* [*of* λ*F*. *inverse-functors A B F G*]
by *simp*

lemma *inv-is-inverse*:
shows *inverse-functors A B inv G* **..**

sublocale *inverse-functors A B inv G*
using *inv-is-inverse* **by** *simp*

lemma *is-surjective-on-objects*:
shows *G* ‘ *Collect A.ide* ⊇ *Collect B.ide*
by (*metis* (*no-types*, *lifting*) *B.category-axioms B.map-simp*
CollectD CollectI F.preserves-ide category.ideD(*1*) *image-eqI*
inv o-apply subsetI)

sublocale *fully-faithful-functor A B G*
proof −
obtain *F* **where** *F*: *inverse-functors A B F G*
using *invertible* **by** *auto*
interpret *FG*: *inverse-functors A B F G*
using *F* **by** *simp*
show *fully-faithful-functor A B G*
proof
fix *f f'*
assume *par*: *A.par f f'* **and** *eq*: *G f* = *G f'*
show *f* = *f'*
using *par eq FG.inv'*
by (*metis A.map-simp comp-apply*)
next
fix *a a' g*
assume *a*: *A.ide a* **and** *a'*: *A.ide a'* **and** *g*: «*g* : *G a* →*B G a'*»
show ∃*f*. «*f* : *a* →*A a'*» ∧ *G f* = *g*
by (*metis A.ideD*(*1*) *A.map-simp B.arrI B.map-simp FG.F.preserves-hom FG.inv*
FG.inv'
a a' g o-apply)
qed
qed

lemma *is-fully-faithful*:
shows *fully-faithful-functor A B G*
..

lemma *preserves-terminal*:
assumes *A.terminal a*

```
  shows B.terminal (G a)
  proof
    show 0: B.ide (G a) using assms G.preserves-ide A.terminal-def by blast
    fix b :: 'b
    assume b: B.ide b
    show ∃!g. «g : b →_B G a»
    proof
      let ?F = SOME F. inverse-functors A B F G
      from invertible have F: inverse-functors A B ?F G
        using someI-ex [of λF. inverse-functors A B F G] by fast
      interpret inverse-functors A B ?F G using F by auto
      let ?P = λf. «f : ?F b →_A a»
      have 1: ∃!f. ?P f using assms b A.terminal-def by simp
      hence 2: ?P (THE f. ?P f) by (metis (no-types, lifting) theI')
      thus «G (THE f. ?P f) : b →_B G a»
        using b apply (elim A.in-homE, intro B.in-homI, auto)
        using B.ideD(1) B.map-simp comp-def inv by metis
      hence 3: «(THE f. ?P f) : ?F b →_A a»
        using assms 2 b F by simp
      fix g :: 'b
      assume g: «g : b →_B G a»
      have ?F (G a) = a
        using assms(1) A.terminal-def inv' A.map-simp
        by (metis 0 B.ideD(1) G.preserves-reflects-arr comp-eq-dest-lhs)
      hence «?F g : ?F b →_A a»
        using assms(1) g A.terminal-def inv
        by (elim B.in-homE, auto)
      hence ?F g = (THE f. ?P f) using assms 1 3 A.terminal-def by blast
      thus g = G (THE f. ?P f)
        using inv g by (metis B.in-homE B.map-simp comp-def)
    qed
  qed

end

context full-embedding-functor
begin

  lemma is-invertible-if-surjective-on-objects:
  assumes F ‘ Collect A.ide ⊇ Collect B.ide
  shows invertible-functor A B F
  and inverse-functors A B (λy. if B.arr y then inv-into (Collect A.arr) F y else A.null) F
  proof −
    have ∗: F ‘ Collect A.ide = Collect B.ide
      using assms preserves-reflects-arr by auto
    have inj: inj-on F (Collect A.arr)
      using is-embedding inj-on-def by blast
    have inj': inj-on F (Collect A.ide)
      by (simp add: inj-on-def is-embedding)
```

```
have surj: F ‘ Collect A.arr = Collect B.arr
proof
  show F ‘ Collect A.arr ⊆ Collect B.arr
    using preserves-reflects-arr by auto
  show Collect B.arr ⊆ F ‘ Collect A.arr
  proof
    fix g
    assume g: g ∈ Collect B.arr
    let ?a = inv-into (Collect A.ide) F (B.dom g)
    let ?a' = inv-into (Collect A.ide) F (B.cod g)
    have a: A.ide ?a ∧ F ?a = B.dom g
      using ∗ g by (simp add: f-inv-into-f reflects-ide)
    have a': A.ide ?a' ∧ F ?a' = B.cod g
      using ∗ g by (simp add: f-inv-into-f reflects-ide)
    have «g : F ?a →B F ?a'»
      using g a a' by auto
    hence ∃f. «f : ?a →A ?a'» ∧ F f = g
      using a a' is-full by blast
    thus g ∈ F ‘ Collect A.arr by blast
  qed
qed
let ?G = λy. if B.arr y then inv-into (Collect A.arr) F y else A.null
show inverse-functors A B ?G F
proof
  show ⋀f. ¬ B.arr f ⟹ ?G f = A.null
    by simp
  show 1: ⋀f. B.arr f ⟹ A.arr (?G f)
    using assms inj surj inv-into-into
    by (metis (full-types) mem-Collect-eq)
  show 2: ⋀f. B.arr f ⟹ A.dom (?G f) = ?G (B.dom f)
  proof −
    fix f
    assume f: B.arr f
    have F (A.dom (?G f)) = B.dom f
    proof −
      have F (A.dom (?G f)) = B.dom (F (inv-into (Collect A.arr) F f))
        using f 1 preserves-dom by simp
      also have ... = B.dom f
        using f f-inv-into-f by (metis CollectI surj)
      finally show ?thesis by blast
    qed
    thus A.dom (?G f) = ?G (B.dom f)
      using f
      by (metis 1 A.arr-dom B.arr-dom inj inv-into-f-f mem-Collect-eq)
  qed
  show 3: ⋀f. B.arr f ⟹ A.cod (?G f) = ?G (B.cod f)
  proof −
    fix f
    assume f: B.arr f
```

```
      have F (A.cod (?G f)) = B.cod f
      proof −
        have F (A.cod (?G f)) = B.cod (F (inv-into (Collect A.arr) F f))
          using f 1 preserves-cod by simp
        also have ... = B.cod f
          using f f-inv-into-f by (metis CollectI surj)
        finally show ?thesis by blast
      qed
      thus A.cod (?G f) = ?G (B.cod f)
        using f
        by (metis 1 A.arr-cod B.arr-cod inj inv-into-f-f mem-Collect-eq)
    qed
    fix f g
    assume fg: B.seq g f
    show ?G (B g f) = A (?G g) (?G f)
      using assms fg 1 2 3 inj surj f-inv-into-f inj-on-def inv-into-into
          preserves-comp
      by (auto simp add: f-inv-into-f is-embedding)
    next
    show F ∘ ?G = B.map
      using inj surj f-inv-into-f A.not-arr-null B.map-def extensionality
      by (auto simp add: f-inv-into-f)
    show ?G ∘ F = A.map
      using inj surj A.extensionality by auto
  qed
  hence ∃ G. inverse-functors A B G F
    by blast
  thus invertible-functor A B F
    using functor-axioms functor-def invertible-functor.intro
        invertible-functor-axioms.intro
    by blast
qed

end

locale dual-functor =
  F: functor A B F +
  Aop: dual-category A +
  Bop: dual-category B
for A :: 'a comp      (infixr ‹·A› 55)
and B :: 'b comp      (infixr ‹·B› 55)
and F :: 'a ⇒ 'b
begin

  notation Aop.comp     (infixr ‹·A^op› 55)
  notation Bop.comp     (infixr ‹·B^op› 55)

  abbreviation map
  where map ≡ F
```

lemma *is-functor*:
shows *functor Aop.comp Bop.comp map*
using *F.extensionality* **by** (*unfold-locales*, *auto*)

end

sublocale *dual-functor* ⊆ *functor Aop.comp Bop.comp map*
using *is-functor* **by** *auto*

A bijection from a set S to the set of arrows of a category C induces an isomorphic copy of C having S as its set of arrows, assuming that there exists some $n \notin S$ to serve as the null.

context *category*
begin

lemma *bij-induces-invertible-functor*:
assumes *bij-betw* φ S (*Collect arr*) **and** $n \notin S$
shows $\exists C'.$ *Collect* (*partial-composition.arr* C') $= S \wedge$
invertible-functor C' C ($\lambda i.$ *if partial-composition.arr* C' i *then* φ i *else null*)
proof −
define ψ **where** $\psi = (\lambda f.$ *if arr f then inv-into S* φ f *else* $n)$
have ψ: *bij-betw* ψ (*Collect arr*) S
using *assms*(*1*) ψ-*def bij-betw-inv-into*
by (*metis* (*no-types*, *lifting*) *bij-betw-cong mem-Collect-eq*)
have φ-ψ [*simp*]: $\bigwedge f.$ *arr* $f \Longrightarrow \varphi$ (ψ f) $= f$
using *assms*(*1*) ψ ψ-*def bij-betw-inv-into-right* **by** *fastforce*
have ψ-φ [*simp*]: $\bigwedge i.$ $i \in S \Longrightarrow \psi$ (φ i) $= i$
unfolding ψ-*def*
using *assms*(*1*) ψ *bij-betw-inv-into-left* [*of* φ S *Collect arr*]
by (*metis bij-betw-def image-eqI mem-Collect-eq*)
define C' **where** $C' = (\lambda i\ j.$ *if* $i \in S \wedge j \in S \wedge$ *seq* (φ i) (φ j) *then* ψ (φ $i \cdot \varphi$ j) *else* $n)$
interpret C': *partial-composition* C'
using *assms*(*1*−*2*) C'-*def* ψ-*def*
by *unfold-locales metis*
have *null-char*: C'.*null* $= n$
using *assms*(*1*−*2*) C'-*def* ψ-*def* C'.*null-eqI* **by** *metis*
have *ide-char*: $\bigwedge i.$ C'.*ide* $i \longleftrightarrow i \in S \wedge$ *ide* (φ i)
proof
fix i
assume i: C'.*ide* i
show $i \in S \wedge$ *ide* (φ i)
proof (*unfold ide-def*, *intro conjI*)
show *1*: φ $i \cdot \varphi$ $i \neq$ *null*
using i *assms*(*1*) C'.*ide-def* C'-*def null-char* **by** *auto*
show *2*: $i \in S$
using *1 assms*(*1*) **by** (*metis* C'.*ide-def* C'-*def* i)
show $\forall f.$ ($f \cdot \varphi$ $i \neq$ *null* $\longrightarrow f \cdot \varphi$ $i = f$) $\wedge$ (φ $i \cdot f \neq$ *null* $\longrightarrow \varphi$ $i \cdot f = f$)
proof (*intro allI conjI impI*)

```
      show ⋀f. f · φ i ≠ null ⟹ f · φ i = f
      proof −
        fix f
        assume f: f · φ i ≠ null
        hence 1: arr f ∧ arr (φ i) ∧ seq f (φ i)
          by (meson seqE ext)
        have f · φ i = φ (C' (ψ f) i)
          using 1 2 C'-def null-char
          by (metis (no-types, lifting) φ-ψ ψ bij-betw-def image-eqI mem-Collect-eq)
        also have ... = f
          by (metis 1 C'.ide-def C'-def φ-ψ ψ assms(2) bij-betw-def i image-eqI
              mem-Collect-eq null-char)
        finally show f · φ i = f by simp
      qed
      show ⋀f. φ i · f ≠ null ⟹ φ i · f = f
      proof −
        fix f
        assume f: φ i · f ≠ null
        hence 1: arr f ∧ arr (φ i) ∧ seq (φ i) f
          by (meson seqE ext)
        show φ i · f = f
          using 1 2 C'-def null-char ψ
          by (metis (no-types, lifting) ‹⋀f. f · φ i ≠ null ⟹ f · φ i = f›
              ide-char' codomains-null comp-cod-arr has-codomain-iff-arr
              comp-ide-arr)
      qed
    qed
  qed
  next
  fix i
  assume i: i ∈ S ∧ ide (φ i)
  have ψ (φ i) ∈ S
    using i assms(1)
    by (metis ψ bij-betw-def ideD(1) image-eqI mem-Collect-eq)
  show C'.ide i
    using assms(2) i C'-def null-char comp-arr-ide comp-ide-arr
    apply (unfold C'.ide-def, intro conjI allI impI)
      apply auto[1]
    by force+
qed
have dom: ⋀i. i ∈ S ⟹ ψ (dom (φ i)) ∈ C'.domains i
proof −
  fix i
  assume i: i ∈ S
  have 1: C'.ide (ψ (dom (φ i)))
    by (metis φ-ψ ψ ψ-φ ψ-def arr-dom assms(2) bij-betw-def i ide-char ide-dom
        image-eqI mem-Collect-eq)
  moreover have C' i (ψ (dom (φ i))) ≠ C'.null
    by (metis C'-def φ-ψ ψ-φ ψ-def assms(2) calculation comp-arr-dom i ide-char
```

```
      null-char)
  ultimately show ψ (dom (φ i)) ∈ C'.domains i
    using C'.domains-def by simp
qed
have cod: ⋀i. i ∈ S ⟹ ψ (cod (φ i)) ∈ C'.codomains i
proof −
  fix i
  assume i: i ∈ S
  have 1: C'.ide (ψ (cod (φ i)))
    by (metis φ-ψ ψ ψ-φ ψ-def arr-cod assms(2) bij-betw-def i ide-char ide-cod
        image-eqI mem-Collect-eq)
  moreover have C' (ψ (cod (φ i))) i ≠ C'.null
    by (metis 1 C'-def φ-ψ ψ-φ ψ-def assms(2) comp-cod-arr i ide-char null-char)
  ultimately show ψ (cod (φ i)) ∈ C'.codomains i
    using C'.codomains-def by simp
qed
have arr-char: ⋀i. C'.arr i ⟷ i ∈ S
  by (metis (mono-tags, lifting) C'.arr-def C'.codomains-def C'.domains-def
    C'-def assms(2) dom mem-Collect-eq null-char C'.cod-in-codomains C'.dom-in-domains)
have seq-char: ⋀i j. C'.seq i j ⟷ i ∈ S ∧ j ∈ S ∧ seq (φ i) (φ j)
  using assms(1−2) C'-def arr-char null-char
  apply simp
  using ψ bij-betw-apply by fastforce
interpret C': category C'
proof
  show ⋀g f. C' g f ≠ C'.null ⟹ C'.seq g f
    using C'-def null-char seq-char by fastforce
  show ⋀f. (C'.domains f ≠ {}) = (C'.codomains f ≠ {})
    using dom cod null-char arr-char C'.arr-def by blast
  show ⋀h g f. ⟦C'.seq h g; C'.seq (C' h g) f⟧ ⟹ C'.seq g f
    using seq-char
    apply simp
    using C'-def by fastforce
  show ⋀h g f. ⟦C'.seq h (C' g f); C'.seq g f⟧ ⟹ C'.seq h g
    using seq-char
    apply simp
    using C'-def by fastforce
  show ⋀g f h. ⟦C'.seq g f; C'.seq h g⟧ ⟹ C'.seq (C' h g) f
    using seq-char arr-char
    apply simp
    using C'-def by auto
  show ⋀g f h. ⟦C'.seq g f; C'.seq h g⟧ ⟹ C' (C' h g) f = C' h (C' g f)
    using seq-char arr-char C'-def comp-assoc assms(2)
    apply simp by presburger
qed
have dom-char: C'.dom = (λi. if i ∈ S then ψ (dom (φ i)) else n)
  using dom arr-char null-char C'.dom-eqI' C'.arr-def C'.dom-def by metis
have cod-char: C'.cod = (λi. if i ∈ S then ψ (cod (φ i)) else n)
  using cod arr-char null-char C'.cod-eqI' C'.arr-def C'.cod-def by metis
```

```
  interpret φ: functor C' C ‹λi. if C'.arr i then φ i else null›
   using arr-char null-char dom-char cod-char seq-char φ-ψ ψ-φ ψ-def C'.not-arr-null C'-def
         C'.arr-dom C'.arr-cod
    apply unfold-locales
       apply simp-all
    by metis+
  interpret ψ: functor C C' ψ
    using ψ-def null-char arr-char
    apply unfold-locales
       apply simp
      apply (metis (no-types, lifting) ψ bij-betw-def image-eqI mem-Collect-eq)
     apply (metis (no-types, lifting) φ-ψ ψ bij-betw-def dom-char image-eqI mem-Collect-eq)
    apply (metis (no-types, lifting) φ-ψ ψ bij-betw-def cod-char image-eqI mem-Collect-eq)
    by (metis (no-types, lifting) C'-def φ-ψ ψ bij-betw-def seqE image-eqI mem-Collect-eq)
  interpret φψ: inverse-functors C' C ψ ‹λi. if C'.arr i then φ i else null›
  proof
    show ψ ∘ (λi. if C'.arr i then φ i else null) = C'.map
      by (auto simp add: C'.extensionality ψ.extensionality arr-char)
    show (λi. if C'.arr i then φ i else null) ∘ ψ = map
      by (auto simp add: extensionality)
  qed
  have invertible-functor C' C (λi. if C'.arr i then φ i else null)
    using φψ.inverse-functors-axioms by unfold-locales auto
  thus ?thesis
    using arr-char by blast
qed

corollary (in category) finite-imp-ex-iso-nat-comp:
assumes finite (Collect arr)
shows ∃ C' :: nat comp. isomorphic-categories C' C
proof −
  obtain n :: nat and φ where φ: bij-betw φ {0..<n} (Collect arr)
    using assms ex-bij-betw-nat-finite by blast
  obtain C' where C': Collect (partial-composition.arr C') = {0..<n} ∧
                      invertible-functor C' (·)
                        (λi. if partial-composition.arr C' i then φ i else null)
    using φ bij-induces-invertible-functor [of φ {0..<n}] by auto
  interpret φ: invertible-functor C' C ‹λi. if partial-composition.arr C' i then φ i else null›
    using C' by simp
  show ?thesis
    using φ.isomorphic-categories-axioms by blast
qed

end
```

We now prove the result, advertised earlier in theory *ConcreteCategory*, that any category is in fact isomorphic to the concrete category formed from it in the obvious way.

```
context category
```

```
  begin

    interpretation CC: concrete-category ‹Collect ide› hom id ‹λ- - - g f. g · f›
      using comp-arr-dom comp-cod-arr comp-assoc
      by (unfold-locales, auto)

    interpretation F: functor C CC.COMP
                      ‹λf. if arr f then CC.MkArr (dom f) (cod f) f else CC.null›
      by (unfold-locales, auto simp add: in-homI)

    interpretation G: functor CC.COMP C ‹λF. if CC.arr F then CC.Map F else null›
      using CC.Map-in-Hom CC.seq-char
      by (unfold-locales, auto)

    interpretation FG: inverse-functors C CC.COMP
                      ‹λF. if CC.arr F then CC.Map F else null›
                      ‹λf. if arr f then CC.MkArr (dom f) (cod f) f else CC.null›
    proof
      show (λF. if CC.arr F then CC.Map F else null) ∘
           (λf. if arr f then CC.MkArr (dom f) (cod f) f else CC.null) =
           map
        using CC.arr-char map-def by fastforce
      show (λf. if arr f then CC.MkArr (dom f) (cod f) f else CC.null) ∘
           (λF. if CC.arr F then CC.Map F else null) =
           CC.map
        using CC.MkArr-Map G.preserves-arr G.preserves-cod G.preserves-dom
              CC.extensionality
        by auto
    qed

    theorem is-isomorphic-to-concrete-category:
    shows isomorphic-categories C CC.COMP
      ..

  end

end
```

Chapter 8

Subcategory

In this chapter we give a construction of the subcategory of a category defined by a predicate on arrows subject to closure conditions. The arrows of the subcategory are directly identified with the arrows of the ambient category. We also define the related notions of full subcategory and inclusion functor.

theory *Subcategory*
imports *Functor*
begin

locale *subcategory* =
  C: *category C*
  for *C* :: *'a comp* (**infixr** ‹·$_C$› *55*)
  and *Arr* :: *'a* ⇒ *bool* +
  assumes *inclusion*: *Arr f* ⟹ *C.arr f*
  and *dom-closed*: *Arr f* ⟹ *Arr* (*C.dom f*)
  and *cod-closed*: *Arr f* ⟹ *Arr* (*C.cod f*)
  and *comp-closed*: ⟦ *Arr f*; *Arr g*; *C.cod f* = *C.dom g* ⟧ ⟹ *Arr* (*g* ·$_C$ *f*)
begin

  no-notation *C.in-hom* (‹«- : - → -»›)
  notation *C.in-hom* (‹«- : - →$_C$ -»›)

  definition *comp* (**infixr** ‹·› *55*)
  where *g* · *f* = (*if Arr f* ∧ *Arr g* ∧ *C.cod f* = *C.dom g then g* ·$_C$ *f else C.null*)

  interpretation *partial-composition comp*
  proof
    show ∃!*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n*
    proof
      show *1*: ∀*f*. *C.null* · *f* = *C.null* ∧ *f* · *C.null* = *C.null*
        by (*metis C.null-is-zero*(*1*) *C.ex-un-null comp-def*)
      show ⋀*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n* ⟹ *n* = *C.null*
        using *1 C.ex-un-null* **by** *metis*
    qed
  qed

lemma *null-char* [*simp*]:
shows *null* = *C.null*
proof −
  have ∀*f*. *C.null* · *f* = *C.null* ∧ *f* · *C.null* = *C.null*
    by (*metis C.null-is-zero*(*1*) *C.ex-un-null comp-def*)
  thus *?thesis* **using** *ex-un-null* **by** (*metis null-is-zero*(*2*))
qed

lemma *ideI*$_{SbC}$:
assumes *Arr a* **and** *C.ide a*
shows *ide a*
  unfolding *ide-def*
  using *assms null-char C.ide-def comp-def* **by** *auto*

lemma *Arr-iff-dom-in-domain*:
shows *Arr f* ⟷ *C.dom f* ∈ *domains f*
proof
  show *C.dom f* ∈ *domains f* ⟹ *Arr f*
    using *domains-def comp-def ide-def* **by** *fastforce*
  show *Arr f* ⟹ *C.dom f* ∈ *domains f*
  proof −
    assume *f*: *Arr f*
    have *ide* (*C.dom f*)
      using *f inclusion C.dom-in-domains C.has-domain-iff-arr C.domains-def*
          dom-closed ideI$_{SbC}$
      by *auto*
    moreover have *f* · *C.dom f* ≠ *null*
      using *f comp-def dom-closed null-char inclusion C.comp-arr-dom* **by** *force*
    ultimately show *?thesis*
      using *domains-def* **by** *simp*
  qed
qed

lemma *Arr-iff-cod-in-codomain*:
shows *Arr f* ⟷ *C.cod f* ∈ *codomains f*
proof
  show *C.cod f* ∈ *codomains f* ⟹ *Arr f*
    using *codomains-def comp-def ide-def* **by** *fastforce*
  show *Arr f* ⟹ *C.cod f* ∈ *codomains f*
  proof −
    assume *f*: *Arr f*
    have *ide* (*C.cod f*)
      using *f inclusion C.cod-in-codomains C.has-codomain-iff-arr C.codomains-def*
          cod-closed ideI$_{SbC}$
      by *auto*
    moreover have *C.cod f* · *f* ≠ *null*
      using *f comp-def cod-closed null-char inclusion C.comp-cod-arr* **by** *force*
    ultimately show *?thesis*

```
    using codomains-def by simp
  qed
qed

lemma arr-char_SbC:
shows arr f ⟷ Arr f
proof
  show Arr f ⟹ arr f
    using arr-def comp-def Arr-iff-dom-in-domain Arr-iff-cod-in-codomain by auto
  show arr f ⟹ Arr f
  proof −
    assume f: arr f
    obtain a where a: a ∈ domains f ∨ a ∈ codomains f
      using f arr-def by auto
    have f · a ≠ C.null ∨ a · f ≠ C.null
      using a domains-def codomains-def null-char by auto
    thus Arr f
      using comp-def by metis
  qed
qed

lemma arrI_SbC [intro]:
assumes Arr f
shows arr f
  using assms arr-char_SbC by simp

lemma arrE [elim]:
assumes arr f
shows Arr f
  using assms arr-char_SbC by simp

interpretation category comp
proof
  show 1: ⋀g f. g · f ≠ null ⟹ seq g f
    using comp-closed comp-def by fastforce
  show ⋀f. (domains f ≠ {}) = (codomains f ≠ {})
   using Arr-iff-cod-in-codomain Arr-iff-dom-in-domain arrE arr-def codomains-def by blast
  show ⋀h g f. ⟦seq h g; seq (h · g) f⟧ ⟹ seq g f
    by (metis (full-types) 1 C.dom-comp C.match-1 C.not-arr-null arrE inclusion comp-def)
  show ⋀h g f. ⟦seq h (g · f); seq g f⟧ ⟹ seq h g
    by (metis (full-types) 1 C.cod-comp C.match-2 C.not-arr-null arrE inclusion comp-def)
  show 2: ⋀g f h. ⟦seq g f; seq h g⟧ ⟹ seq (h · g) f
    by (metis (full-types) 1 C.dom-comp C.not-arr-null C.seqI arrE inclusion comp-def)
  show ⋀g f h. ⟦seq g f; seq h g⟧ ⟹ (h · g) · f = h · g · f
    by (metis 2 C.comp-assoc C.not-arr-null arrE C.cod-comp inclusion comp-def)
qed

theorem is-category:
shows category comp ..
```

notation *in-hom* (‹«- : - → -»›)

lemma *dom-simp*:
assumes *arr f*
shows *dom f* = *C.dom f*
by (*metis Arr-iff-dom-in-domain arrE assms dom-eqI'*)

lemma *dom-char*$_{SbC}$:
shows *dom f* = (*if arr f then C.dom f else C.null*)
using *dom-simp dom-def arr-def arr-char*$_{SbC}$ **by** *auto*

lemma *cod-simp*:
assumes *arr f*
shows *cod f* = *C.cod f*
by (*metis Arr-iff-cod-in-codomain arrE assms cod-eqI'*)

lemma *cod-char*$_{SbC}$:
shows *cod f* = (*if arr f then C.cod f else C.null*)
using *cod-simp cod-def arr-def* **by** *auto*

lemma *in-hom-char*$_{SbC}$:
shows «*f* : *a* → *b*» ⟷ *arr a* ∧ *arr b* ∧ *arr f* ∧ «*f* : *a* →$_C$ *b*»
using *inclusion arr-char*$_{SbC}$ *cod-closed dom-closed*
by (*metis C.arr-iff-in-hom C.in-homE arr-iff-in-hom cod-simp dom-simp in-homE*)

lemma *ide-char*$_{SbC}$:
shows *ide a* ⟷ *arr a* ∧ *C.ide a*
using *ide-in-hom C.ide-in-hom in-hom-char*$_{SbC}$ **by** *simp*

lemma *seq-char*$_{SbC}$:
shows *seq g f* ⟷ *arr f* ∧ *arr g* ∧ *C.seq g f*
proof
show *arr f* ∧ *arr g* ∧ *C.seq g f* ⟹ *seq g f*
using *arr-char*$_{SbC}$ *dom-char*$_{SbC}$ *cod-char*$_{SbC}$ **by** (*intro seqI*, *auto*)
show *seq g f* ⟹ *arr f* ∧ *arr g* ∧ *C.seq g f*
apply (*elim seqE*, *auto*)
using *inclusion arr-char*$_{SbC}$ *dom-simp cod-simp* **by** *auto*
qed

lemma *hom-char*:
shows *hom a b* = *C.hom a b* ∩ *Collect Arr*
proof
show *hom a b* ⊆ *C.hom a b* ∩ *Collect Arr*
using *in-hom-char*$_{SbC}$ **by** *auto*
show *C.hom a b* ∩ *Collect Arr* ⊆ *hom a b*
using *arr-char*$_{SbC}$ *dom-char*$_{SbC}$ *cod-char*$_{SbC}$ **by** *force*
qed

lemma *comp-char*:
shows $g \cdot f = (\mathit{if}\ \mathit{arr}\ f \wedge \mathit{arr}\ g \wedge C.\mathit{seq}\ g\ f\ \mathit{then}\ g \cdot_C f\ \mathit{else}\ C.\mathit{null})$
using $\mathit{arr\text{-}char}_{SbC}$ *comp-def comp-closed C.ext* **by** *force*

lemma *comp-simp*:
assumes *seq g f*
shows $g \cdot f = g \cdot_C f$
using *assms comp-char* $\mathit{seq\text{-}char}_{SbC}$ **by** *metis*

lemma *inclusion-preserves-inverse*:
assumes *inverse-arrows f g*
shows *C.inverse-arrows f g*
using *assms* $\mathit{ide\text{-}char}_{SbC}$ *comp-simp* $\mathit{arr\text{-}char}_{SbC}$
by (*intro C.inverse-arrowsI*, *auto*)

lemma $\mathit{iso\text{-}char}_{SbC}$:
shows $\mathit{iso}\ f \longleftrightarrow C.\mathit{iso}\ f \wedge \mathit{arr}\ f \wedge \mathit{arr}\ (C.\mathit{inv}\ f)$
by (*metis C.category-axioms C.cod-inv C.comp-arr-inv' C.comp-inv-arr' C.dom-inv arr-inv category.inverse-unique category.isoI category.seqI cod-simp comp-simp dom-simp ide-cod inverse-arrowsI is-category iso-is-arr iso-def inclusion-preserves-inverse*)

lemma $\mathit{inv\text{-}char}_{SbC}$:
assumes *iso f*
shows $\mathit{inv}\ f = C.\mathit{inv}\ f$
by (*metis assms C.inverse-unique inclusion-preserves-inverse isoE inverse-unique*)

lemma $\mathit{inverse\text{-}arrows\text{-}char}_{SbC}$:
shows *inverse-arrows f g* $\longleftrightarrow$ *seq f g* $\wedge$ *C.inverse-arrows f g*
by (*metis C.inverse-arrows-def comp-simp* $\mathit{ide\text{-}char}_{SbC}$ *ide-compE inverse-arrows-def*)

end

sublocale *subcategory* $\subseteq$ *category comp*
using *is-category* **by** *auto*

8.1 Full Subcategory

locale *full-subcategory* =
C: *category C*
for *C* :: *'a comp*
and *Ide* :: $'a \Rightarrow \mathit{bool}$ +
assumes $\mathit{inclusion}_{FSbC}$: $\mathit{Ide}\ f \Longrightarrow C.\mathit{ide}\ f$
begin

sublocale *subcategory C* $\lambda f.\ C.\mathit{arr}\ f \wedge \mathit{Ide}\ (C.\mathit{dom}\ f) \wedge \mathit{Ide}\ (C.\mathit{cod}\ f)$
by (*unfold-locales*; *simp*)

lemma *is-subcategory*:
shows *subcategory C* $(\lambda f.\ C.\mathit{arr}\ f \wedge \mathit{Ide}\ (C.\mathit{dom}\ f) \wedge \mathit{Ide}\ (C.\mathit{cod}\ f))$

..

lemma *in-hom-char*$_{FSbC}$:
shows «$f : a \to b$» $\longleftrightarrow$ *arr a* $\wedge$ *arr b* $\wedge$ «$f : a \to_C b$»
using *arr-char*$_{SbC}$ *in-hom-char*$_{SbC}$ **by** *auto*

Isomorphisms in a full subcategory are inherited from the ambient category.

lemma *iso-char*$_{FSbC}$:
shows *iso f* $\longleftrightarrow$ *arr f* $\wedge$ *C.iso f*
using *arr-char*$_{SbC}$ *iso-char*$_{SbC}$ **by** *force*

end

8.2 Inclusion Functor

If S is a subcategory of C, then there is an inclusion functor from S to C. Inclusion functors are faithful embeddings.

locale *inclusion-functor* =
C: *category C* +
S: *subcategory C Arr*
for *C* :: *'a comp*
and *Arr* :: *'a* $\Rightarrow$ *bool*
begin

interpretation *functor S.comp C S.map*
using *S.map-def S.arr-char*$_{SbC}$ *S.inclusion S.dom-char*$_{SbC}$ *S.cod-char*$_{SbC}$
S.dom-closed S.cod-closed S.comp-closed S.arr-char$_{SbC}$ *S.comp-char*
apply *unfold-locales*
apply *auto*[4]
by (*elim S.seqE*, *auto*)

lemma *is-functor*:
shows *functor S.comp C S.map* **..**

interpretation *faithful-functor S.comp C S.map*
apply *unfold-locales* **by** *simp*

lemma *is-faithful-functor*:
shows *faithful-functor S.comp C S.map* **..**

interpretation *embedding-functor S.comp C S.map*
apply *unfold-locales* **by** *simp*

lemma *is-embedding-functor*:
shows *embedding-functor S.comp C S.map* **..**

end

sublocale *inclusion-functor* $\subseteq$ *faithful-functor S.comp C S.map*
using *is-faithful-functor* **by** *auto*
sublocale *inclusion-functor* $\subseteq$ *embedding-functor S.comp C S.map*
using *is-embedding-functor* **by** *auto*

The inclusion of a full subcategory is a special case. Such functors are fully faithful.

locale *full-inclusion-functor* =
C: *category C* +
S: *full-subcategory C Ide*
for *C* :: *'a comp*
and *Ide* :: *'a* $\Rightarrow$ *bool*
begin

sublocale *inclusion-functor C* ‹$\lambda f.\ C.arr\ f \wedge Ide\ (C.dom\ f) \wedge Ide\ (C.cod\ f)$› **..**

lemma *is-inclusion-functor*:
shows *inclusion-functor C* $(\lambda f.\ C.arr\ f \wedge Ide\ (C.dom\ f) \wedge Ide\ (C.cod\ f))$
..

interpretation *full-functor S.comp C S.map*
apply *unfold-locales*
using *S.ide-in-hom*
by (*metis* (*no-types*, *lifting*) *C.in-homE S.arr-char*$_{SbC}$ *S.in-hom-char*$_{FSbC}$ *S.map-simp*)

lemma *is-full-functor*:
shows *full-functor S.comp C S.map* **..**

sublocale *full-functor S.comp C S.map*
using *is-full-functor* **by** *auto*
sublocale *fully-faithful-functor S.comp C S.map* **..**

end

end

Chapter 9

SetCategory

theory *SetCategory*
imports *Category Functor Subcategory*
begin

This theory defines a locale *set-category* that axiomatizes the notion "category of $'a$-sets and functions between them" in the context of HOL. A primary reason for doing this is to make it possible to prove results (such as the Yoneda Lemma) that use such categories without having to commit to a particular element type $'a$ and without having the results depend on the concrete details of a particular construction. The axiomatization given here is categorical, in the sense that if categories S and S' each interpret the *set-category* locale, then a bijection between the sets of terminal objects of S and S' extends to an isomorphism of S and S' as categories.

The axiomatization is based on the following idea: if, for some type $'a$, category S is the category of all $'a$-sets and functions between them, then the elements of type $'a$ are in bijective correspondence with the terminal objects of category S. In addition, if *unity* is an arbitrarily chosen terminal object of S, then for each object a, the hom-set *hom unity a* (*i.e.* the set of "points" or "global elements" of a) is in bijective correspondence with a subset of the terminal objects of S. By making a specific, but arbitrary, choice of such a correspondence, we can then associate with each object a of S a set *set a* that consists of all terminal objects t that correspond to some point x of a. Each arrow f then induces a function $Fun\ f \in set\ (dom\ f) \to set\ (cod\ f)$, defined on terminal objects of S by passing to points of *dom f*, composing with f, then passing back from points of *cod f* to terminal objects. Once we can associate a set with each object of S and a function with each arrow, we can force S to be isomorphic to the category of $'a$-sets by imposing suitable extensionality and completeness axioms.

9.1 Some Lemmas about Restriction

The development of the *set-category* locale makes heavy use of the theory *HOL−Library.FuncSet*. However, in some cases, I found that that theory did not provide results about restriction in the form that was most useful to me. I used the following

additional results in various places.

lemma *restr-eqI*:
assumes $A = A'$ **and** $\bigwedge x.\ x \in A \Longrightarrow F\ x = F'\ x$
shows *restrict* $F\ A =$ *restrict* $F'\ A'$
using *assms* **by** *force*

lemma *restr-eqE* [*elim*]:
assumes *restrict* $F\ A =$ *restrict* $F'\ A$ **and** $x \in A$
shows $F\ x = F'\ x$
using *assms restrict-def* **by** *metis*

lemma *compose-eq'* [*simp*]:
shows *compose* $A\ G\ F =$ *restrict* $(G\ o\ F)\ A$
unfolding *compose-def restrict-def* **by** *auto*

9.2 Set Categories

We first define the locale *set-category-data*, which sets out the basic data and definitions for the *set-category* locale, without imposing any conditions other than that S is a category and that *img* is a function defined on the arrow type of S. The function *img* should be thought of as a mapping that takes a point $x \in hom\ unity\ a$ to a corresponding terminal object *img x*. Eventually, assumptions will be introduced so that this is in fact the case. The set of terminal objects of the category will serve as abstract "elements" of sets; we will refer to the set of *all* terminal objects as the *universe*.

locale *set-category-data* = *category S*
for $S :: 's\ comp$ (**infixr** ‹·› *55*)
and $img :: 's \Rightarrow 's$
begin

notation *in-hom* (‹«- : - → -»›)

Call the set of all terminal objects of S the "universe".

abbreviation *Univ* :: *'s set*
where *Univ* ≡ *Collect terminal*

Choose an arbitrary element of the universe and call it *unity*.

definition *unity* :: *'s*
where *unity* = (*SOME t. terminal t*)

Each object a determines a subset *set a* of the universe, consisting of all those terminal objects t such that $t =$ *img x* for some $x \in hom\ unity\ a$.

definition *set* :: *'s* ⇒ *'s set*
where *set a* = *img* ‘ *hom unity a*

end

Next, we define a locale *set-category-given-img* that augments the *set-category-data* locale with assumptions that serve to define the notion of a set category with a chosen correspondence between points and terminal objects. The assumptions require that the universe be nonempty (so that the definition of *unity* makes sense), that the map *img* is a locally injective map taking points to terminal objects, that each terminal object t belongs to *set t*, that two objects of S are equal if they determine the same set, that two parallel arrows of S are equal if they determine the same function, and that for any objects a and b and function $F \in hom\ unity\ a \to hom\ unity\ b$ there is an arrow $f \in hom\ a\ b$ whose action under the composition of S coincides with the function F.

The parameter *setp* is a predicate that determines which subsets of the universe are to be regarded as defining objects of the category. This parameter has been introduced because most of the characteristic properties of a category of sets and functions do not depend on there being an object corresponding to *every* subset of the universe, and we intend to consider in particular the cases in which only finite subsets or only "small" subsets of the universe determine objects. Accordingly, we assume that there is an object corresponding to each subset of the universe that satisfies *setp*. It is also necessary to assume some basic regularity properties of the predicate *setp*; namely, that it holds for all subsets of the universe corresponding to objects of S, and that it respects subset and union.

locale *set-category-given-img* = *set-category-data S img*
for *S* :: *'s comp* (**infixr** ‹·› *55*)
and *img* :: *'s* ⇒ *'s*
and *setp* :: *'s set* ⇒ *bool* +
assumes *setp-imp-subset-Univ*: *setp A* ⟹ *A* ⊆ *Univ*
and *setp-set-ide*: *ide a* ⟹ *setp* (*set a*)
and *setp-respects-subset*: *A'* ⊆ *A* ⟹ *setp A* ⟹ *setp A'*
and *setp-respects-union*: ⟦ *setp A*; *setp B* ⟧ ⟹ *setp* (*A* ∪ *B*)
and *nonempty-Univ*: *Univ* ≠ {}
and *inj-img*: *ide a* ⟹ *inj-on img* (*hom unity a*)
and *stable-img*: *terminal t* ⟹ *t* ∈ *img* ‘ *hom unity t*
and *extensional-set*: ⟦ *ide a*; *ide b*; *set a* = *set b* ⟧ ⟹ *a* = *b*
and *extensional-arr*: ⟦ *par f f'*; ⋀*x*. «*x* : *unity* → *dom f*» ⟹ *f* · *x* = *f'* · *x* ⟧ ⟹ *f* = *f'*
and *set-complete*: *setp A* ⟹ ∃ *a*. *ide a* ∧ *set a* = *A*
and *fun-complete-ax*: ⟦ *ide a*; *ide b*; *F* ∈ *hom unity a* → *hom unity b* ⟧
⟹ ∃ *f*. «*f* : *a* → *b*» ∧ (∀ *x*. «*x* : *unity* → *dom f*» ⟶ *f* · *x* = *F x*)
begin

lemma *setp-singleton*:
assumes *terminal a*
shows *setp* {*a*}
using *assms*
by (*metis setp-set-ide Set.set-insert Un-upper1 insert-is-Un set-def*
setp-respects-subset stable-img terminal-def)

lemma *setp-empty*:
shows *setp* {}
using *setp-singleton setp-respects-subset nonempty-Univ* **by** *blast*

lemma *finite-imp-setp*:
assumes $A \subseteq Univ$ **and** *finite A*
shows *setp A*
using *setp-empty setp-singleton setp-respects-union*
by (*metis assms*(1−2) *finite-subset-induct insert-is-Un mem-Collect-eq*)

Each arrow $f \in hom\ a\ b$ determines a function $Fun\ f \in Univ \to Univ$, by passing from *Univ* to *hom a unity*, composing with *f*, then passing back to *Univ*.

definition *Fun* :: *'s* ⇒ *'s* ⇒ *'s*
where *Fun f* = *restrict* (*img o S f o inv-into* (*hom unity* (*dom f*)) *img*) (*set* (*dom f*))

lemma *comp-arr-point*$_{SC}$:
assumes *arr f* **and** «*x* : *unity* → *dom f*»
shows *f* · *x* = *inv-into* (*hom unity* (*cod f*)) *img* (*Fun f* (*img x*))
proof −
have «*f* · *x* : *unity* → *cod f*»
using *assms* **by** *blast*
thus *?thesis*
using *assms Fun-def inj-img set-def* **by** *simp*
qed

Parallel arrows that determine the same function are equal.

lemma *arr-eqI*$_{SC}$:
assumes *par f f'* **and** *Fun f* = *Fun f'*
shows *f* = *f'*
using *assms comp-arr-point*$_{SC}$ *extensional-arr* **by** *metis*

lemma *terminal-unity*$_{SC}$:
shows *terminal unity*
using *unity-def nonempty-Univ* **by** (*simp add*: *someI-ex*)

lemma *ide-unity* [*simp*]:
shows *ide unity*
using *terminal-unity*$_{SC}$ *terminal-def* **by** *blast*

lemma *setp-set'* [*simp*]:
assumes *ide a*
shows *setp* (*set a*)
using *assms setp-set-ide* **by** *auto*

lemma *inj-on-set*:
shows *inj-on set* (*Collect ide*)
using *extensional-set* **by** (*intro inj-onI*, *auto*)

The inverse of the map *set* is a map *mkIde* that takes each subset of the universe to an identity of *S*.

definition *mkIde* :: *'s set* ⇒ *'s*
where *mkIde A* = (*if setp A then inv-into* (*Collect ide*) *set A else null*)

lemma *mkIde-set* [*simp*]:
assumes *ide a*
shows *mkIde* (*set a*) = *a*
by (*simp add*: *assms inj-on-set mkIde-def*)

lemma *set-mkIde* [*simp*]:
assumes *setp A*
shows *set* (*mkIde A*) = *A*
using *assms mkIde-def set-complete someI-ex* [*of* λ*a*. *a* ∈ *Collect ide* ∧ *set a* = *A*]
mkIde-set
by *metis*

lemma *ide-mkIde* [*simp*]:
assumes *setp A*
shows *ide* (*mkIde A*)
using *assms mkIde-def mkIde-set set-complete* **by** *metis*

lemma *arr-mkIde* [*iff*]:
shows *arr* (*mkIde A*) ⟷ *setp A*
using *not-arr-null mkIde-def ide-mkIde* **by** *auto*

lemma *dom-mkIde* [*simp*]:
assumes *setp A*
shows *dom* (*mkIde A*) = *mkIde A*
using *assms ide-mkIde* **by** *simp*

lemma *cod-mkIde* [*simp*]:
assumes *setp A*
shows *cod* (*mkIde A*) = *mkIde A*
using *assms ide-mkIde* **by** *simp*

Each arrow *f* determines an extensional function from *set* (*dom f*) to *set* (*cod f*).

lemma *Fun-mapsto*:
assumes *arr f*
shows *Fun f* ∈ *extensional* (*set* (*dom f*)) ∩ (*set* (*dom f*) → *set* (*cod f*))
proof
show *Fun f* ∈ *extensional* (*set* (*dom f*)) **using** *Fun-def* **by** *fastforce*
show *Fun f* ∈ *set* (*dom f*) → *set* (*cod f*)
proof
fix *t*
assume *t*: *t* ∈ *set* (*dom f*)
have *Fun f t* = *img* (*f* · *inv-into* (*hom unity* (*dom f*)) *img t*)
using *assms t Fun-def comp-def* **by** *simp*
moreover have ... ∈ *set* (*cod f*)
using *assms t set-def inv-into-into* [*of t img hom unity* (*dom f*)] **by** *blast*
ultimately show *Fun f t* ∈ *set* (*cod f*) **by** *auto*
qed
qed

Identities of S correspond to restrictions of the identity function.

lemma *Fun-ide*:
assumes *ide a*
shows *Fun a = restrict* (λ*x*. *x*) (*set a*)
using *assms Fun-def inj-img set-def comp-cod-arr* **by** *fastforce*

lemma *Fun-mkIde* [*simp*]:
assumes *setp A*
shows *Fun* (*mkIde A*) = *restrict* (λ*x*. *x*) *A*
using *assms ide-mkIde set-mkIde Fun-ide* **by** *simp*

Composition in (·) corresponds to extensional function composition.

lemma *Fun-comp* [*simp*]:
assumes *seq g f*
shows *Fun* (*g* · *f*) = *restrict* (*Fun g o Fun f*) (*set* (*dom f*))
proof −
have *restrict* (*img o S* (*g* · *f*) *o* (*inv-into* (*hom unity* (*dom* (*g* · *f*))) *img*))
(*set* (*dom* (*g* · *f*)))
= *restrict* (*Fun g o Fun f*) (*set* (*dom f*))
proof −
let *?img′* = λ*a*. λ*t*. *inv-into* (*hom unity a*) *img t*
have *1*: *set* (*dom* (*g* · *f*)) = *set* (*dom f*)
using *assms* **by** *auto*
moreover have ⋀*t*. *t* ∈ *set* (*dom* (*g* · *f*)) ⟹
(*img o S* (*g* · *f*) *o ?img′* (*dom* (*g* · *f*))) *t* = (*Fun g o Fun f*) *t*
proof −
fix *t*
assume *t* ∈ *set* (*dom* (*g* · *f*))
hence *t*: *t* ∈ *set* (*dom f*) **by** (*simp add*: *1*)
have (*img o S* (*g* · *f*) *o ?img′* (*dom* (*g* · *f*))) *t* = *img* (*g* · *f* · *?img′* (*dom f*) *t*)
using *assms dom-comp comp-assoc* **by** *simp*
also have ... = *img* (*g* · *?img′* (*dom g*) (*Fun f t*))
proof −
have ⋀*a x*. *x* ∈ *hom unity a* ⟹ *?img′ a* (*img x*) = *x*
using *assms inj-img ide-cod inv-into-f-eq*
by (*metis arrI in-homE mem-Collect-eq*)
thus *?thesis*
using *assms t Fun-def set-def comp-arr-point*$_{SC}$ **by** *auto*
qed
also have ... = *Fun g* (*Fun f t*)
proof −
have *Fun f t* ∈ *img* ‘ *hom unity* (*cod f*)
using *assms t Fun-mapsto set-def* **by** *fast*
thus *?thesis*
using *assms* **by** (*auto simp add*: *set-def Fun-def*)
qed
finally show (*img o S* (*g* · *f*) *o ?img′* (*dom* (*g* · *f*))) *t* = (*Fun g o Fun f*) *t*
by *auto*
qed

```
    ultimately show ?thesis by auto
  qed
  thus ?thesis using Fun-def by auto
qed
```

The constructor *mkArr* is used to obtain an arrow given subsets A and B of the universe and a function $F \in A \to B$.

```
definition mkArr :: 's set ⇒ 's set ⇒ ('s ⇒ 's) ⇒ 's
where mkArr A B F = (if setp A ∧ setp B ∧ F ∈ A → B
                     then (THE f. f ∈ hom (mkIde A) (mkIde B) ∧ Fun f = restrict F A)
                     else null)
```

Each function $F \in set\ a \to set\ b$ determines a unique arrow $f \in hom\ a\ b$, such that $Fun\ f$ is the restriction of F to $set\ a$.

```
lemma fun-complete:
assumes ide a and ide b and F ∈ set a → set b
shows ∃!f. «f : a → b» ∧ Fun f = restrict F (set a)
proof −
  let ?P = λf. «f : a → b» ∧ Fun f = restrict F (set a)
  show ∃!f. ?P f
  proof
    have ∃f. ?P f
    proof −
      let ?F' = λx. inv-into (hom unity b) img (F (img x))
      have ?F' ∈ hom unity a → hom unity b
      proof
        fix x
        assume x: x ∈ hom unity a
        have F (img x) ∈ set b using assms(3) x set-def by auto
        thus inv-into (hom unity b) img (F (img x)) ∈ hom unity b
          using assms setp-set-ide inj-img set-def by auto
      qed
      hence ∃f. «f : a → b» ∧ (∀x. «x : unity → a» ⟶ f · x = ?F' x)
        using assms fun-complete-ax [of a b] by force
      from this obtain f where f: «f : a → b» ∧ (∀x. «x : unity → a» ⟶ f · x = ?F' x)
        by blast
      let ?img' = λa. λt. inv-into (hom unity a) img t
      have Fun f = restrict F (set a)
      proof (unfold Fun-def, intro restr-eqI)
        show set (dom f) = set a using f by auto
        show ⋀t. t ∈ set (dom f) ⟹ (img ∘ S f ∘ ?img' (dom f)) t = F t
        proof −
          fix t
          assume t: t ∈ set (dom f)
          have (img ∘ S f ∘ ?img' (dom f)) t = img (f · ?img' (dom f) t)
            by simp
          also have ... = img (?F' (?img' (dom f) t))
            by (metis f in-homE inv-into-into set-def mem-Collect-eq t)
          also have ... = img (?img' (cod f) (F t))
```

```
        using f t set-def inj-img by auto
      also have ... = F t
      proof −
        have F t ∈ set (cod f)
          using assms f t by auto
        thus ?thesis
          using f t set-def inj-img by auto
      qed
      finally show (img ∘ S f ∘ ?img' (dom f)) t = F t by auto
    qed
  qed
  thus ?thesis using f by blast
 qed
 thus F: ?P (SOME f. ?P f) using someI-ex [of ?P] by fast
 show ⋀f'. ?P f' ⟹ f' = (SOME f. ?P f)
  using F arr-eqI_SC
  by (metis (no-types, lifting) in-homE)
qed
qed

lemma mkArr-in-hom:
assumes setp A and setp B and F ∈ A → B
shows «mkArr A B F : mkIde A → mkIde B»
  using assms mkArr-def fun-complete [of mkIde A mkIde B F] ide-mkIde set-mkIde
        theI' [of λf. f ∈ hom (mkIde A) (mkIde B) ∧ Fun f = restrict F A]
        setp-imp-subset-Univ
  by simp
```

The "only if" direction of the next lemma can be achieved only if there exists a non-arrow element of type $'s$, which can be used as the value of *mkArr A B F* in cases where $F \notin A \to B$. Nevertheless, it is essential to have this, because without the "only if" direction, we can't derive any useful consequences from an assumption of the form *arr* (*mkArr A B F*); instead we have to obtain $F \in A \to B$ some other way. This is is usually highly inconvenient and it makes the theory very weak and almost unusable in practice. The observation that having a non-arrow value of type $'s$ solves this problem is ultimately what led me to incorporate *null* first into the definition of the *set-category* locale and then, ultimately, into the definition of the *category* locale. I believe this idea is critical to the usability of the entire development.

```
lemma arr-mkArr [iff]:
shows arr (mkArr A B F) ⟷ setp A ∧ setp B ∧ F ∈ A → B
proof
  show arr (mkArr A B F) ⟹ setp A ∧ setp B ∧ F ∈ A → B
    using mkArr-def not-arr-null ex-un-null someI-ex [of λf. ¬arr f] setp-imp-subset-Univ
    by metis
  show setp A ∧ setp B ∧ F ∈ A → B ⟹ arr (mkArr A B F)
    using mkArr-in-hom by auto
qed
```

lemma *arr-mkArrI* [*intro*]:
assumes *setp A* **and** *setp B* **and** $F \in A \to B$
shows *arr* (*mkArr A B F*)
using *assms arr-mkArr* **by** *blast*

lemma *Fun-mkArr'*:
assumes *arr* (*mkArr A B F*)
shows «*mkArr A B F* : *mkIde A* → *mkIde B*»
and *Fun* (*mkArr A B F*) = *restrict F A*
proof −
have *1*: *setp A* ∧ *setp B* ∧ $F \in A \to B$ **using** *assms* **by** *fast*
have *2*: *mkArr A B F* ∈ *hom* (*mkIde A*) (*mkIde B*) ∧
Fun (*mkArr A B F*) = *restrict F* (*set* (*mkIde A*))
proof −
have ∃!*f*. *f* ∈ *hom* (*mkIde A*) (*mkIde B*) ∧ *Fun f* = *restrict F* (*set* (*mkIde A*))
using *1 fun-complete* [*of mkIde A mkIde B F*] *ide-mkIde set-mkIde* **by** *simp*
thus *?thesis* **using** *1 mkArr-def theI' set-mkIde* **by** *simp*
qed
show «*mkArr A B F* : *mkIde A* → *mkIde B*» **using** *1 2* **by** *auto*
show *Fun* (*mkArr A B F*) = *restrict F A* **using** *1 2 set-mkIde* **by** *auto*
qed

lemma *mkArr-Fun*:
assumes *arr f*
shows *mkArr* (*set* (*dom f*)) (*set* (*cod f*)) (*Fun f*) = *f*
proof −
have *1*: *setp* (*set* (*dom f*)) ∧ *setp* (*set* (*cod f*)) ∧ *ide* (*dom f*) ∧ *ide* (*cod f*) ∧
Fun f ∈ *extensional* (*set* (*dom f*)) ∩ (*set* (*dom f*) → *set* (*cod f*))
using *Fun-mapsto assms ide-cod ide-dom setp-set'* **by** *presburger*
hence ∃!*f'*. *f'* ∈ *hom* (*dom f*) (*cod f*) ∧ *Fun f'* = *restrict* (*Fun f*) (*set* (*dom f*))
using *fun-complete* **by** *force*
moreover have *f* ∈ *hom* (*dom f*) (*cod f*) ∧ *Fun f* = *restrict* (*Fun f*) (*set* (*dom f*))
using *assms 1 extensional-restrict* **by** *force*
ultimately have *f* = (*THE f'*. *f'* ∈ *hom* (*dom f*) (*cod f*) ∧
Fun f' = *restrict* (*Fun f*) (*set* (*dom f*)))
using *theI'* [*of* λ*f'*. *f'* ∈ *hom* (*dom f*) (*cod f*) ∧ *Fun f'* = *restrict* (*Fun f*) (*set* (*dom f*))]
by *blast*
also have ... = *mkArr* (*set* (*dom f*)) (*set* (*cod f*)) (*Fun f*)
using *assms 1 mkArr-def mkIde-set* **by** *simp*
finally show *?thesis* **by** *auto*
qed

lemma *dom-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)
shows *dom* (*mkArr A B F*) = *mkIde A*
using *assms Fun-mkArr'* **by** *auto*

lemma *cod-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)

shows *cod* (*mkArr A B F*) = *mkIde B*
using *assms Fun-mkArr'* **by** *auto*

lemma *Fun-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)
shows *Fun* (*mkArr A B F*) = *restrict F A*
using *assms Fun-mkArr'* **by** *auto*

The following provides the basic technique for showing that arrows constructed using *mkArr* are equal.

lemma *mkArr-eqI* [*intro*]:
assumes *arr* (*mkArr A B F*)
and $A = A'$ **and** $B = B'$ **and** $\bigwedge x.\ x \in A \Longrightarrow F\ x = F'\ x$
shows *mkArr A B F* = *mkArr A' B' F'*
using *assms Fun-mkArr*
by (*intro arr-eqI*$_{SC}$, *auto simp add*: *Pi-iff*)

This version avoids trivial proof obligations when the domain and codomain sets are identical from the context.

lemma *mkArr-eqI'* [*intro*]:
assumes *arr* (*mkArr A B F*) **and** $\bigwedge x.\ x \in A \Longrightarrow F\ x = F'\ x$
shows *mkArr A B F* = *mkArr A B F'*
using *assms mkArr-eqI* **by** *simp*

lemma *mkArr-restrict-eq*:
assumes *arr* (*mkArr A B F*)
shows *mkArr A B* (*restrict F A*) = *mkArr A B F*
using *assms* **by** (*intro mkArr-eqI'*, *auto*)

lemma *mkArr-restrict-eq'*:
assumes *arr* (*mkArr A B* (*restrict F A*))
shows *mkArr A B* (*restrict F A*) = *mkArr A B F*
using *assms* **by** (*intro mkArr-eqI'*, *auto*)

lemma *mkIde-as-mkArr* [*simp*]:
assumes *setp A*
shows *mkArr A A* (λ*x*. *x*) = *mkIde A*
using *assms arr-mkIde dom-mkIde cod-mkIde Fun-mkIde*
by (*intro arr-eqI*$_{SC}$, *auto*)

lemma *comp-mkArr*:
assumes *arr* (*mkArr A B F*) **and** *arr* (*mkArr B C G*)
shows *mkArr B C G* · *mkArr A B F* = *mkArr A C* (*G* ∘ *F*)
proof (*intro arr-eqI*$_{SC}$)
have *1*: *seq* (*mkArr B C G*) (*mkArr A B F*) **using** *assms* **by** *force*
have *2*: *G o F* ∈ *A* → *C* **using** *assms* **by** *auto*
show *par* (*mkArr B C G* · *mkArr A B F*) (*mkArr A C* (*G* ∘ *F*))
using *assms 1 2*
by (*intro conjI*) *simp-all*

```
  show Fun (mkArr B C G · mkArr A B F) = Fun (mkArr A C (G ∘ F))
    using 1 2 by fastforce
qed
```

The locale assumption *stable-img* forces $t \in set\ t$ in case t is a terminal object. This is very convenient, as it results in the characterization of terminal objects as identities t for which $set\ t = \{t\}$. However, it is not absolutely necessary to have this. The following weaker characterization of terminal objects can be proved without the *stable-img* assumption.

```
lemma terminal-char1:
shows terminal t ⟷ ide t ∧ (∃!x. x ∈ set t)
proof −
  have terminal t ⟹ ide t ∧ (∃!x. x ∈ set t)
  proof −
    assume t: terminal t
    have ide t using t terminal-def by auto
    moreover have ∃!x. x ∈ set t
    proof −
      have ∃!x. x ∈ hom unity t
        using t terminal-unity_SC terminal-def by auto
      thus ?thesis using set-def by auto
    qed
    ultimately show ide t ∧ (∃!x. x ∈ set t) by auto
  qed
  moreover have ide t ∧ (∃!x. x ∈ set t) ⟹ terminal t
  proof −
    assume t: ide t ∧ (∃!x. x ∈ set t)
    from this obtain t' where set t = {t'} by blast
    hence t': set t = {t'} ∧ setp {t'} ∧ t = mkIde {t'}
      using t setp-set-ide mkIde-set by metis
    show terminal t
    proof
      show ide t using t by simp
      show ⋀a. ide a ⟹ ∃!f. «f : a → t»
      proof −
        fix a
        assume a: ide a
        show ∃!f. «f : a → t»
        proof
          show 1: «mkArr (set a) {t'} (λx. t') : a → t»
            using a t t' mkArr-in-hom
            by (metis Pi-I' mkIde-set setp-set-ide singletonD)
          show ⋀f. «f : a → t» ⟹ f = mkArr (set a) {t'} (λx. t')
          proof −
            fix f
            assume f: «f : a → t»
            show f = mkArr (set a) {t'} (λx. t')
            proof (intro arr-eqI_SC)
              show 1: par f (mkArr (set a) {t'} (λx. t')) using 1 f in-homE by metis
```

```
            show Fun f = Fun (mkArr (set a) {t'} (λx. t'))
            proof −
              have Fun (mkArr (set a) {t'} (λx. t')) = (λx∈set a. t')
                using 1 Fun-mkArr by simp
              also have ... = Fun f
              proof −
                have ⋀x. x ∈ set a ⟹ Fun f x = t'
                  using f t' Fun-def mkArr-Fun arr-mkArr
                  by (metis PiE in-homE singletonD)
                moreover have ⋀x. x ∉ set a ⟹ Fun f x = undefined
                  using f Fun-def by auto
                ultimately show ?thesis by auto
              qed
              finally show ?thesis by force
            qed
          qed
        qed
      qed
    qed
    ultimately show ?thesis by blast
  qed
```

As stated above, in the presence of the *stable-img* assumption we have the following stronger characterization of terminal objects.

```
  lemma terminal-char2:
  shows terminal t ⟷ ide t ∧ set t = {t}
  proof
    assume t: terminal t
    show ide t ∧ set t = {t}
    proof
      show ide t using t terminal-char1 by auto
      show set t = {t}
      proof −
        have ∃!x. x ∈ hom unity t using t terminal-def terminal-unity_SC by force
        moreover have t ∈ img ‘ hom unity t using t stable-img set-def by simp
        ultimately show ?thesis using set-def by auto
      qed
    qed
    next
    assume ide t ∧ set t = {t}
    thus terminal t using terminal-char1 by force
  qed

end
```

At last, we define the *set-category* locale by existentially quantifying out the choice of a particular *img* map. We need to know that such a map exists, but it does not matter which one we choose.

```
locale set-category = category S
for S :: 's comp      (infixr ‹·› 55)
and setp :: 's set ⇒ bool +
assumes ex-img: ∃ img. set-category-given-img S img setp
begin

  notation in-hom (‹«- : - → -»›)

  definition some-img
  where some-img = (SOME img. set-category-given-img S img setp)

  sublocale set-category-given-img S some-img setp
  proof −
    have ∃ img. set-category-given-img S img setp using ex-img by auto
    thus set-category-given-img S some-img setp
      using someI-ex [of λimg. set-category-given-img S img setp] some-img-def
      by metis
  qed

end
```

We call a set category *replete* if there is an object corresponding to every subset of the universe.

```
locale replete-set-category =
  category S +
  set-category S ‹λA. A ⊆ Collect terminal›
for S :: 's comp      (infixr ‹·› 55)
begin

  abbreviation setp
  where setp ≡ λA. A ⊆ Univ

  lemma is-set-category:
  shows set-category S (λA. A ⊆ Collect terminal)
    ..

end

context set-category
begin
```

The arbitrary choice of *img* induces a system of arrows corresponding to inclusions of subsets.

```
  definition incl :: 's ⇒ bool
  where incl f = (arr f ∧ set (dom f) ⊆ set (cod f) ∧
                  f = mkArr (set (dom f)) (set (cod f)) (λx. x))

  lemma Fun-incl:
  assumes incl f
```

```
  shows Fun f = (λx ∈ set (dom f). x)
    using assms incl-def by (metis Fun-mkArr)

  lemma ex-incl-iff-subset:
  assumes ide a and ide b
  shows (∃f. «f : a → b» ∧ incl f) ⟷ set a ⊆ set b
  proof
    show ∃f. «f : a → b» ∧ incl f ⟹ set a ⊆ set b
      using incl-def by auto
    show set a ⊆ set b ⟹ ∃f. «f : a → b» ∧ incl f
    proof
      assume 1: set a ⊆ set b
      show «mkArr (set a) (set b) (λx. x) : a → b» ∧ incl (mkArr (set a) (set b) (λx. x))
      proof
        show «mkArr (set a) (set b) (λx. x) : a → b»
          by (metis 1 assms image-ident image-subset-iff-funcset mkIde-set
              mkArr-in-hom setp-set-ide)
        thus incl (mkArr (set a) (set b) (λx. x))
          using 1 incl-def by force
      qed
    qed
  qed

end
```

9.3 Categoricity

In this section we show that the *set-category* locale completely characterizes the structure of its interpretations as categories, in the sense that for any two interpretations S and S', a *setp*-respecting bijection between the universe of S and the universe of S' extends to an isomorphism of S and S'.

```
  locale two-set-categories-bij-betw-Univ =
    S: set-category S setp +
    S': set-category S' setp'
  for S :: 's comp       (infixr ‹·› 55)
  and setp :: 's set ⇒ bool
  and S' :: 't comp      (infixr ‹·'› 55)
  and setp' :: 't set ⇒ bool
  and φ :: 's ⇒ 't +
  assumes bij-φ: bij-betw φ S.Univ S'.Univ
  and φ-respects-setp: A ⊆ S.Univ ⟹ setp' (φ ` A) ⟷ setp A
  begin

    notation S.in-hom     (‹«- : - → -»›)
    notation S'.in-hom    (‹«- : - →' -»›)

    abbreviation ψ
    where ψ ≡ inv-into S.Univ φ
```

lemma ψ-φ:
assumes $t \in S.Univ$
shows $\psi\ (\varphi\ t) = t$
using *assms bij-φ bij-betw-inv-into-left* **by** *metis*

lemma φ-ψ:
assumes $t' \in S'.Univ$
shows $\varphi\ (\psi\ t') = t'$
using *assms bij-φ bij-betw-inv-into-right* **by** *metis*

lemma ψ-*img*-φ-*img*:
assumes $A \subseteq S.Univ$
shows $\psi\ `\ \varphi\ `\ A = A$
using *assms bij-φ* **by** (*simp add*: *bij-betw-def*)

lemma φ-*img*-ψ-*img*:
assumes $A' \subseteq S'.Univ$
shows $\varphi\ `\ \psi\ `\ A' = A'$
using *assms bij-φ* **by** (*simp add*: *bij-betw-def image-inv-into-cancel*)

We define the object map Φo of a functor from S to S'.

definition Φo
where $\Phi o = (\lambda a \in Collect\ S.ide.\ S'.mkIde\ (\varphi\ `\ S.set\ a))$

lemma *set*-Φo:
assumes $S.ide\ a$
shows $S'.set\ (\Phi o\ a) = \varphi\ `\ S.set\ a$
by (*simp add*: *S.setp-imp-subset-Univ Φo-def φ-respects-setp assms*)

lemma Φo-*preserves-ide*:
assumes $S.ide\ a$
shows $S'.ide\ (\Phi o\ a)$
using *assms S'.ide-mkIde bij-φ bij-betw-def image-mono restrict-apply' S.setp-set' φ-respects-setp S.setp-imp-subset-Univ*
unfolding Φo-*def*
by *simp*

The map Φa assigns to each arrow f of S the function on the universe of S' that is the same as the function induced by f on the universe of S, up to the bijection φ between the two universes.

definition Φa
where $\Phi a = (\lambda f.\ \lambda x' \in \varphi\ `\ S.set\ (S.dom\ f).\ \varphi\ (S.Fun\ f\ (\psi\ x')))$

lemma Φa-*mapsto*:
assumes $S.arr\ f$
shows $\Phi a\ f \in S'.set\ (\Phi o\ (S.dom\ f)) \to S'.set\ (\Phi o\ (S.cod\ f))$
proof −
have $\Phi a\ f \in \varphi\ `\ S.set\ (S.dom\ f) \to \varphi\ `\ S.set\ (S.cod\ f)$

```
  proof
    fix x
    assume x: x ∈ φ ‘ S.set (S.dom f)
    have ψ x ∈ S.set (S.dom f)
      using assms x ψ-img-φ-img [of S.set (S.dom f)] S.setp-imp-subset-Univ by auto
    hence S.Fun f (ψ x) ∈ S.set (S.cod f) using assms S.Fun-mapsto by auto
    hence φ (S.Fun f (ψ x)) ∈ φ ‘ S.set (S.cod f) by simp
    thus Φa f x ∈ φ ‘ S.set (S.cod f) using x Φa-def by auto
  qed
  thus ?thesis using assms set-Φo Φo-preserves-ide by auto
qed
```

The map Φa takes composition of arrows to extensional composition of functions.

```
lemma Φa-comp:
assumes gf: S.seq g f
shows Φa (g · f) = restrict (Φa g o Φa f) (S'.set (Φo (S.dom f)))
proof −
  have Φa (g · f) = (λx' ∈ φ ‘ S.set (S.dom f). φ (S.Fun (S g f) (ψ x')))
    using gf Φa-def by auto
  also have ... = (λx' ∈ φ ‘ S.set (S.dom f).
                      φ (restrict (S.Fun g o S.Fun f) (S.set (S.dom f)) (ψ x')))
    using gf set-Φo S.Fun-comp by simp
  also have ... = restrict (Φa g o Φa f) (S'.set (Φo (S.dom f)))
  proof −
    have ⋀x'. x' ∈ φ ‘ S.set (S.dom f)
          ⟹ φ (restrict (S.Fun g o S.Fun f) (S.set (S.dom f)) (ψ x')) = Φa g (Φa f x')
    proof −
      fix x'
      assume X': x' ∈ φ ‘ S.set (S.dom f)
      hence 1: ψ x' ∈ S.set (S.dom f)
        using gf ψ-img-φ-img S.setp-imp-subset-Univ S.ide-dom S.setp-set-ide
        by blast
      hence φ (restrict (S.Fun g o S.Fun f) (S.set (S.dom f)) (ψ x'))
            = φ (S.Fun g (S.Fun f (ψ x')))
        using restrict-apply by auto
      also have ... = φ (S.Fun g (ψ (φ (S.Fun f (ψ x')))))
      proof −
        have S.Fun f (ψ x') ∈ S.set (S.cod f)
          using gf 1 S.Fun-mapsto by fast
        hence ψ (φ (S.Fun f (ψ x'))) = S.Fun f (ψ x')
          using assms bij-φ S.setp-imp-subset-Univ bij-betw-def inv-into-f-f subsetCE
                S.ide-cod S.setp-set-ide
          by (metis S.seqE)
        thus ?thesis by auto
      qed
      also have ... = Φa g (Φa f x')
      proof −
        have Φa f x' ∈ φ ‘ S.set (S.cod f)
          using gf S.ide-dom S.ide-cod X' Φa-mapsto [of f] set-Φo [of S.dom f]
```

```
            set-Φo [of S.cod f]
        by blast
      thus ?thesis using gf X' Φa-def by auto
    qed
    finally show φ (restrict (S.Fun g o S.Fun f) (S.set (S.dom f)) (ψ x')) =
                  Φa g (Φa f x')
      by auto
  qed
  thus ?thesis using assms set-Φo by fastforce
 qed
 finally show ?thesis by auto
qed
```

Finally, we use Φo and Φa to define a functor Φ.

```
definition Φ
where Φ f = (if S.arr f then
               S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod f))) (Φa f)
             else S'.null)

lemma Φ-in-hom:
assumes S.arr f
shows Φ f ∈ S'.hom (Φo (S.dom f)) (Φo (S.cod f))
proof −
 have «Φ f : S'.dom (Φ f) →' S'.cod (Φ f)»
   using assms Φ-def Φa-mapsto Φo-preserves-ide
   by (intro S'.in-homI) auto
 thus ?thesis
   using assms Φ-def Φa-mapsto Φo-preserves-ide by auto
qed

lemma Φ-ide [simp]:
assumes S.ide a
shows Φ a = Φo a
proof −
 have Φ a = S'.mkArr (S'.set (Φo a)) (S'.set (Φo a)) (λx'. x')
 proof −
   have «Φ a : Φo a →' Φo a»
     using assms Φ-in-hom S.ide-in-hom by fastforce
   moreover have Φa a = restrict (λx'. x') (S'.set (Φo a))
   proof −
     have Φa a = (λx' ∈ φ ‘ S.set a. φ (S.Fun a (ψ x')))
       using assms Φa-def restrict-apply by auto
     also have ... = (λx' ∈ S'.set (Φo a). φ (ψ x'))
     proof −
       have S.Fun a = (λx ∈ S.set a. x)
         using assms S.Fun-ide by auto
       moreover have ⋀x'. x' ∈ φ ‘ S.set a ⟹ ψ x' ∈ S.set a
         using assms bij-φ S.setp-imp-subset-Univ image-iff S.setp-set-ide
         by (metis ψ-img-φ-img)
```

```
      ultimately show ?thesis
        using assms set-Φo by auto
    qed
    also have ... = restrict (λx'. x') (S'.set (Φo a))
      using assms S'.setp-imp-subset-Univ S'.setp-set-ide Φo-preserves-ide φ-ψ
      by (meson restr-eqI subsetCE)
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis
    using assms Φ-def Φo-preserves-ide S'.mkArr-restrict-eq'
    by (metis S'.arrI S.ide-char)
qed
thus ?thesis
  using assms S'.mkIde-as-mkArr Φo-preserves-ide Φ-in-hom S'.mkIde-set
  by simp
qed

lemma set-dom-Φ:
assumes S.arr f
shows S'.set (S'.dom (Φ f)) = φ ' (S.set (S.dom f))
  using assms S.ide-dom Φ-in-hom Φ-ide set-Φo by fastforce

lemma Φ-comp:
assumes S.seq g f
shows Φ (g · f) = Φ g ·' Φ f
proof −
  have Φ (g · f) = S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod g))) (Φa (S g f))
    using Φ-def assms by auto
  also have ... = S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod g)))
                           (restrict (Φa g o Φa f) (S'.set (Φo (S.dom f))))
    using assms Φa-comp set-Φo by force
  also have ... = S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod g))) (Φa g o Φa f)
    by (metis S'.mkArr-restrict-eq' Φ-in-hom assms calculation S'.in-homE mem-Collect-eq)
  also have ... = S' (S'.mkArr (S'.set (Φo (S.dom g))) (S'.set (Φo (S.cod g))) (Φa g))
                     (S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod f))) (Φa f))
  proof −
    have S'.arr (S'.mkArr (S'.set (Φo (S.dom f))) (S'.set (Φo (S.cod f))) (Φa f))
      using assms Φa-mapsto set-Φo S.ide-dom S.ide-cod Φo-preserves-ide
            S'.arr-mkArr S'.setp-imp-subset-Univ S'.setp-set-ide S.seqE
      by metis
    moreover have S'.arr (S'.mkArr (S'.set (Φo (S.dom g))) (S'.set (Φo (S.cod g)))
                                   (Φa g))
      using assms Φa-mapsto set-Φo S.ide-dom S.ide-cod Φo-preserves-ide S'.arr-mkArr
            S'.setp-imp-subset-Univ S'.setp-set-ide S.seqE
      by metis
    ultimately show ?thesis using assms S'.comp-mkArr by auto
  qed
  also have ... = Φ g ·' Φ f using assms Φ-def by force
  finally show ?thesis by fast
```

qed

interpretation Φ: *functor S S′ Φ*
apply *unfold-locales*
using Φ-*def*
apply *simp*
using Φ-*in-hom* Φ-*comp*
by *auto*

lemma Φ-*is-functor*:
shows *functor S S′ Φ* **..**

lemma *Fun*-Φ:
assumes *S.arr f* **and** $x \in$ *S.set* (*S.dom f*)
shows *S′.Fun* (Φ *f*) (φ *x*) = Φ*a f* (φ *x*)
using *assms* Φ-*def* Φ.*preserves-arr set-*Φ*o* **by** *auto*

lemma Φ-*acts-elementwise*:
assumes *S.ide a*
shows *S′.set* (Φ *a*) = Φ ‘ *S.set a*
proof
have *0*: *S′.set* (Φ *a*) = φ ‘ *S.set a*
using *assms* Φ-*ide set-*Φ*o* **by** *simp*
have *1*: $\bigwedge x.\ x \in$ *S.set a* $\Longrightarrow$ Φ *x* = φ *x*
proof −
fix *x*
assume *x*: *x* ∈ *S.set a*
have *1*: *S.terminal x* **using** *assms x S.setp-imp-subset-Univ S.setp-set-ide* **by** *blast*
hence *2*: *S′.terminal* (φ *x*)
by (*metis CollectD CollectI bij-*φ *bij-betw-def image-iff*)
have Φ *x* = Φ*o x*
using *assms x 1* Φ-*ide S.terminal-def* **by** *auto*
also have ... = φ *x*
proof −
have Φ*o x* = *S′.mkIde* (φ ‘ *S.set x*)
using *assms 1 x* Φ*o-def S.terminal-def* **by** *auto*
moreover have *S′.mkIde* (φ ‘ *S.set x*) = φ *x*
using *assms x 1 2 S.terminal-char2 S′.terminal-char2 S′.mkIde-set bij-*φ
by (*metis* (*no-types*, *lifting*) *empty-is-image image-insert*)
ultimately show *?thesis* **by** *auto*
qed
finally show Φ *x* = φ *x* **by** *auto*
qed
show *S′.set* (Φ *a*) ⊆ Φ ‘ *S.set a* **using** *0 1* **by** *force*
show Φ ‘ *S.set a* ⊆ *S′.set* (Φ *a*) **using** *0 1* **by** *force*
qed

lemma Φ-*preserves-incl*:
assumes *S.incl m*

```
shows S'.incl (Φ m)
proof −
  have 1: S.arr m ∧ S.set (S.dom m) ⊆ S.set (S.cod m) ∧
        m = S.mkArr (S.set (S.dom m)) (S.set (S.cod m)) (λx. x)
    using assms S.incl-def by blast
  have S'.arr (Φ m) using 1 by auto
  moreover have 2: S'.set (S'.dom (Φ m)) ⊆ S'.set (S'.cod (Φ m))
    using 1 Φ.preserves-dom Φ.preserves-cod Φ-acts-elementwise by auto
  moreover have Φ m =
              S'.mkArr (S'.set (S'.dom (Φ m))) (S'.set (S'.cod (Φ m))) (λx'. x')
  proof −
    have Φ m = S'.mkArr (S'.set (Φo (S.dom m))) (S'.set (Φo (S.cod m))) (Φa m)
      using 1 Φ-def by simp
    also have ... = S'.mkArr (S'.set (S'.dom (Φ m))) (S'.set (S'.cod (Φ m))) (Φa m)
      using 1 Φ-ide by auto
    finally have 3: Φ m =
                  S'.mkArr (S'.set (S'.dom (Φ m))) (S'.set (S'.cod (Φ m))) (Φa m)
      by auto
    also have ... = S'.mkArr (S'.set (S'.dom (Φ m))) (S'.set (S'.cod (Φ m))) (λx'. x')
    proof −
      have 4: S.Fun m = restrict (λx. x) (S.set (S.dom m))
        using assms S.incl-def by (metis (full-types) S.Fun-mkArr)
      hence Φa m = restrict (λx'. x') (φ ‘ (S.set (S.dom m)))
      proof −
        have 5: ⋀x'. x' ∈ φ ‘ S.set (S.dom m) ⟹ φ (ψ x') = x'
          by (meson 1 S.ide-dom S.setp-imp-subset-Univ S.setp-set' f-inv-into-f
                    image-mono subset-eq)
        have Φa m = restrict (λx'. φ (S.Fun m (ψ x'))) (φ ‘ S.set (S.dom m))
          using Φa-def by simp
        also have ... = restrict (λx'. x') (φ ‘ S.set (S.dom m))
        proof −
          have ⋀x. x ∈ φ ‘ (S.set (S.dom m)) ⟹ φ (S.Fun m (ψ x)) = x
          proof −
            fix x
            assume x: x ∈ φ ‘ (S.set (S.dom m))
            hence ψ x ∈ S.set (S.dom m)
              using 1 S.ide-dom S.setp-imp-subset-Univ S.setp-set-ide ψ-img-φ-img image-eqI
              by metis
            thus φ (S.Fun m (ψ x)) = x using 1 4 5 x by simp
          qed
          thus ?thesis by auto
        qed
        finally show ?thesis by auto
      qed
      hence Φa m = restrict (λx'. x') (S'.set (S'.dom (Φ m)))
        using 1 set-dom-Φ by auto
      thus ?thesis
        using 2 3 ‹S'.arr (Φ m)› S'.mkArr-restrict-eq S'.ide-cod S'.ide-dom S'.incl-def
        by (metis S'.arr-mkArr image-restrict-eq image-subset-iff-funcset)
```

qed
finally show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **using** *S'.incl-def* **by** *blast*
qed

lemma *ψ-respects-sets*:
assumes $A' \subseteq S'.Univ$
shows *setp* (ψ ‘ A') $\longleftrightarrow$ *setp'* A'
using *assms φ-respects-setp φ-img-ψ-img bij-φ*
by (*metis ψ-img-φ-img bij-betw-def image-mono order-refl*)

Interchange the role of φ and ψ to obtain a functor Ψ from S' to S.

interpretation *INV*: *two-set-categories-bij-betw-Univ S' setp' S setp ψ*
using *ψ-respects-sets bij-φ bij-betw-inv-into*
by *unfold-locales auto*

abbreviation Ψo
where $\Psi o \equiv INV.\Phi o$

abbreviation Ψa
where $\Psi a \equiv INV.\Phi a$

abbreviation Ψ
where $\Psi \equiv INV.\Phi$

interpretation Ψ: *functor S' S* Ψ
using *INV.Φ-is-functor* **by** *auto*

The functors Φ and Ψ are inverses.

lemma *Fun-Ψ*:
assumes *S'.arr f'* **and** $x' \in S'.set$ (*S'.dom f'*)
shows *S.Fun* (Ψ *f'*) (ψ *x'*) = Ψa *f'* (ψ *x'*)
using *assms INV.Fun-Φ* **by** *blast*

lemma *Ψo-Φo*:
assumes *S.ide a*
shows Ψo (Φo *a*) = *a*
using *assms Φo-def INV.Φo-def ψ-img-φ-img Φo-preserves-ide set-Φo S.mkIde-set*
by (*simp add*: *S.setp-imp-subset-Univ*)

lemma *ΦΨ*:
assumes *S.arr f*
shows Ψ (Φ *f*) = *f*
proof (*intro S.arr-eqI*$_{SC}$)
show *par*: *S.par* (Ψ (Φ *f*)) *f*
using *assms Φo-preserves-ide Ψo-Φo* **by** *auto*
show *S.Fun* (Ψ (Φ *f*)) = *S.Fun f*
proof −

```
    have S.arr (Ψ (Φ f)) using assms by auto
    moreover have Ψ (Φ f) = S.mkArr (S.set (S.dom f)) (S.set (S.cod f)) (Ψa (Φ f))
      using assms INV.Φ-def Φ-in-hom Ψo-Φo by auto
    moreover have Ψa (Φ f) = (λx ∈ S.set (S.dom f). ψ (S'.Fun (Φ f) (φ x)))
    proof −
      have Ψa (Φ f) = (λx ∈ ψ ‘ S'.set (S'.dom (Φ f)). ψ (S'.Fun (Φ f) (φ x)))
      proof −
        have ⋀x. x ∈ ψ ‘ S'.set (S'.dom (Φ f)) ⟹ INV.ψ x = φ x
          using assms S.ide-dom S.setp-imp-subset-Univ Ψ.preserves-reflects-arr par bij-φ
                inv-into-inv-into-eq subsetCE INV.set-dom-Φ
          by (metis (no-types) S.setp-set')
        thus ?thesis
          using INV.Φa-def by auto
      qed
      moreover have ψ ‘ S'.set (S'.dom (Φ f)) = S.set (S.dom f)
        using assms by (metis par Ψ.preserves-reflects-arr INV.set-dom-Φ)
      ultimately show ?thesis by auto
    qed
    ultimately have 1: S.Fun (Ψ (Φ f)) = (λx ∈ S.set (S.dom f). ψ (S'.Fun (Φ f) (φ x)))
      using S'.Fun-mkArr by simp
    show ?thesis
    proof
      fix x
      have x ∉ S.set (S.dom f) ⟹ S.Fun (Ψ (Φ f)) x = S.Fun f x
        using 1 assms extensional-def S.Fun-mapsto S.Fun-def by auto
      moreover have x ∈ S.set (S.dom f) ⟹ S.Fun (Ψ (Φ f)) x = S.Fun f x
      proof −
        assume x: x ∈ S.set (S.dom f)
        have S.Fun (Ψ (Φ f)) x = ψ (φ (S.Fun f (ψ (φ x))))
          using assms x 1 Fun-Φ bij-φ Φa-def by auto
        also have ... = S.Fun f x
        proof −
          have 2: ⋀x. x ∈ S.Univ ⟹ ψ (φ x) = x
            using bij-φ bij-betw-inv-into-left by fast
          have S.Fun f (ψ (φ x)) = S.Fun f x
            using assms x 2 S.ide-dom S.setp-imp-subset-Univ
            by (metis S.setp-set' subsetD)
          moreover have S.Fun f x ∈ S.Univ
            using x assms S.Fun-mapsto S.setp-imp-subset-Univ S.setp-set' S.ide-cod
            by blast
          ultimately show ?thesis using 2 by auto
        qed
        finally show ?thesis by auto
      qed
      ultimately show S.Fun (Ψ (Φ f)) x = S.Fun f x by auto
    qed
  qed
qed
```

lemma Φo-Ψo:
assumes $S'.ide\ a'$
shows $\Phi o\ (\Psi o\ a') = a'$
using *assms* Φo-*def INV*.Φo-*def* φ-*img*-ψ-*img INV*.Φo-*preserves-ide* ψ-φ *INV*.*set*-Φo S'.*mkIde-set* S'.*setp-imp-subset-Univ*
by *force*

lemma $\Psi\Phi$:
assumes $S'.arr\ f'$
shows $\Phi\ (\Psi\ f') = f'$
proof (*intro* S'.*arr-eqI*$_{SC}$)
show *par*: $S'.par\ (\Phi\ (\Psi\ f'))\ f'$
using *assms* Φ.*preserves-ide* Ψ.*preserves-ide* Φ-*ide INV*.Φ-*ide* Φo-Ψo **by** *auto*
show $S'.Fun\ (\Phi\ (\Psi\ f')) = S'.Fun\ f'$
proof −
have $S'.arr\ (\Phi\ (\Psi\ f'))$ **using** *assms* **by** *blast*
moreover have $\Phi\ (\Psi\ f') =$
$S'.mkArr\ (S'.set\ (S'.dom\ f'))\ (S'.set\ (S'.cod\ f'))\ (\Phi a\ (\Psi\ f'))$
using *assms* Φ-*def INV*.Φ-*in-hom* Φo-Ψo **by** *simp*
moreover have $\Phi a\ (\Psi\ f') = (\lambda x' \in S'.set\ (S'.dom\ f').\ \varphi\ (S.Fun\ (\Psi\ f')\ (\psi\ x')))$
unfolding Φa-*def*
using *assms par* Ψ.*preserves-arr set-dom*-Φ **by** *metis*
ultimately have *1*: $S'.Fun\ (\Phi\ (\Psi\ f')) =$
$(\lambda x' \in S'.set\ (S'.dom\ f').\ \varphi\ (S.Fun\ (\Psi\ f')\ (\psi\ x')))$
using S'.*Fun-mkArr* **by** *simp*
show *?thesis*
proof
fix x'
have $x' \notin S'.set\ (S'.dom\ f') \Longrightarrow S'.Fun\ (\Phi\ (\Psi\ f'))\ x' = S'.Fun\ f'\ x'$
using *1 assms* S'.*Fun-mapsto extensional-def* **by** (*simp add*: S'.*Fun-def*)
moreover have $x' \in S'.set\ (S'.dom\ f') \Longrightarrow S'.Fun\ (\Phi\ (\Psi\ f'))\ x' = S'.Fun\ f'\ x'$
proof −
assume x': $x' \in S'.set\ (S'.dom\ f')$
have $S'.Fun\ (\Phi\ (\Psi\ f'))\ x' = \varphi\ (S.Fun\ (\Psi\ f')\ (\psi\ x'))$
using x' *1* **by** *auto*
also have $... = \varphi\ (\Psi a\ f'\ (\psi\ x'))$
using *Fun*-Ψ x' *assms* S'.*setp-imp-subset-Univ bij*-φ **by** *metis*
also have $... = \varphi\ (\psi\ (S'.Fun\ f'\ (\varphi\ (\psi\ x'))))$
proof −
have $\varphi\ (\Psi a\ f'\ (\psi\ x')) = \varphi\ (\psi\ (S'.Fun\ f'\ x'))$
proof −
have $x' \in S'.Univ$
by (*meson* S'.*ide-dom* S'.*setp-imp-subset-Univ* S'.*setp-set-ide assms subsetCE* x')
thus *?thesis*
by (*simp add*: *INV*.Φa-*def INV*.ψ-φ x')
qed
also have $... = \varphi\ (\psi\ (S'.Fun\ f'\ (\varphi\ (\psi\ x'))))$
using *assms* x' φ-ψ S'.*setp-imp-subset-Univ* S'.*setp-set-ide* S'.*ide-dom*
by (*metis subsetCE*)

```
        finally show ?thesis by auto
      qed
      also have ... = S'.Fun f' x'
      proof −
        have 2: ⋀x'. x' ∈ S'.Univ ⟹ φ (ψ x') = x'
          using bij-φ bij-betw-inv-into-right by fast
        have S'.Fun f' (φ (ψ x')) = S'.Fun f' x'
          using assms x' 2 S'.setp-imp-subset-Univ S'.setp-set-ide S'.ide-dom
          by (metis subsetCE)
        moreover have S'.Fun f' x' ∈ S'.Univ
          using x' assms S'.Fun-mapsto S'.setp-imp-subset-Univ S'.setp-set-ide S'.ide-cod
          by blast
        ultimately show ?thesis using 2 by auto
      qed
      finally show ?thesis by auto
    qed
    ultimately show S'.Fun (Φ (Ψ f')) x' = S'.Fun f' x' by auto
  qed
 qed

 lemma inverse-functors-Φ-Ψ:
 shows inverse-functors S S' Ψ Φ
 proof −
  interpret ΦΨ: composite-functor S S' S Φ Ψ ..
  have inv: Ψ o Φ = S.map
    using ΦΨ S.map-def ΦΨ.extensionality by auto

  interpret ΨΦ: composite-functor S' S S' Ψ Φ ..
  have inv': Φ o Ψ = S'.map
    using ΨΦ S'.map-def ΨΦ.extensionality by auto

  show ?thesis
    using inv inv' by (unfold-locales, auto)
 qed

 lemma are-isomorphic:
 shows ∃Φ. invertible-functor S S' Φ ∧ (∀m. S.incl m ⟶ S'.incl (Φ m))
 proof −
  interpret inverse-functors S S' Ψ Φ
    using inverse-functors-Φ-Ψ by auto
  have 1: inverse-functors S S' Ψ Φ ..
  interpret invertible-functor S S' Φ
    apply unfold-locales using 1 by auto
  have invertible-functor S S' Φ ..
  thus ?thesis using Φ-preserves-incl by auto
 qed

end
```

The main result: *set-category* is categorical, in the following (logical) sense: If S and S' are two "set categories", and if the sets of terminal objects of S and S' are in correspondence via a *setp*-preserving bijection, then S and S' are isomorphic as categories, via a functor that preserves inclusion maps, hence also the inclusion relation between sets.

theorem *set-category-is-categorical*:
assumes *set-category S setp* **and** *set-category S′ setp′*
and *bij-betw φ* (*set-category-data.Univ S*) (*set-category-data.Univ S′*)
and $\bigwedge A.\ A \subseteq$ *set-category-data.Univ S* $\Longrightarrow$ *setp′* (*φ* ‘ *A*) $\longleftrightarrow$ *setp A*
shows $\exists \Phi.$ *invertible-functor S S′ Φ* $\wedge$
$(\forall\, m.$ *set-category.incl S setp m* $\longrightarrow$ *set-category.incl S′ setp′* (*Φ m*))
proof −
interpret *S*: *set-category S setp* **using** *assms*(*1*) **by** *auto*
interpret *S′*: *set-category S′ setp′* **using** *assms*(*2*) **by** *auto*
interpret *two-set-categories-bij-betw-Univ S setp S′ setp′ φ*
apply (*unfold-locales*) **using** *assms*(*3*−*4*) **by** *auto*
show *?thesis* **using** *are-isomorphic* **by** *auto*
qed

9.4 Further Properties of Set Categories

In this section we further develop the consequences of the *set-category* axioms, and establish characterizations of a number of standard category-theoretic notions for a *set-category*.

context *set-category*
begin

abbreviation *Dom*
where *Dom f* ≡ *set* (*dom f*)

abbreviation *Cod*
where *Cod f* ≡ *set* (*cod f*)

9.4.1 Initial Object

The object corresponding to the empty set is an initial object.

definition *empty*
where *empty* = *mkIde* {}

lemma *initial-empty*:
shows *initial empty*
proof
show *0*: *ide empty*
using *empty-def* **by** (*simp add*: *setp-empty*)
show $\bigwedge b.$ *ide b* $\Longrightarrow \exists!f.$ «*f* : *empty* → *b*»
proof −
fix *b*
assume *b*: *ide b*

show $\exists!f.\ «f : empty \to b»$
proof
show *1*: $«mkArr\ \{\}\ (set\ b)\ (\lambda x.\ x) : empty \to b»$
using *0 b empty-def mkArr-in-hom mkIde-set setp-imp-subset-Univ arr-mkIde*
by (*metis Pi-I empty-iff ide-def mkIde-def*)
show $\bigwedge f.\ «f : empty \to b» \Longrightarrow f = mkArr\ \{\}\ (set\ b)\ (\lambda x.\ x)$
by (*metis 1 arr-mkArr empty-iff in-homE empty-def mkArr-Fun mkArr-eqI set-mkIde*)
qed
qed
qed

9.4.2 Identity Arrows

Identity arrows correspond to restrictions of the identity function.

lemma *ide-char*$_{SC}$:
assumes *arr f*
shows $ide\ f \longleftrightarrow Dom\ f = Cod\ f \wedge Fun\ f = (\lambda x \in Dom\ f.\ x)$
using *assms mkIde-as-mkArr mkArr-Fun Fun-ide in-homE ide-cod mkArr-Fun mkIde-set*
by (*metis ide-char*)

lemma *ideI*:
assumes *arr f* **and** $Dom\ f = Cod\ f$ **and** $\bigwedge x.\ x \in Dom\ f \Longrightarrow Fun\ f\ x = x$
shows *ide f*
proof −
have $Fun\ f = (\lambda x \in Dom\ f.\ x)$
using *assms Fun-def* **by** *auto*
thus *?thesis* **using** *assms ide-char*$_{SC}$ **by** *blast*
qed

9.4.3 Inclusions

lemma *ide-implies-incl*:
assumes *ide a*
shows *incl a*
by (*simp add*: *assms incl-def*)

definition *incl-in* :: $'s \Rightarrow 's \Rightarrow bool$
where $incl\text{-}in\ a\ b = (ide\ a \wedge ide\ b \wedge set\ a \subseteq set\ b)$

abbreviation *incl-of*
where $incl\text{-}of\ a\ b \equiv mkArr\ (set\ a)\ (set\ b)\ (\lambda x.\ x)$

lemma *elem-set-implies-set-eq-singleton*:
assumes $a \in set\ b$
shows $set\ a = \{a\}$
proof −
have *ide b* **using** *assms set-def* **by** *auto*
thus *?thesis* **using** *assms setp-imp-subset-Univ terminal-char2*
by (*metis setp-set' insert-subset mem-Collect-eq mk-disjoint-insert*)

qed

lemma *elem-set-implies-incl-in*:
assumes $a \in set\ b$
shows *incl-in a b*
proof −
  have *b*: *ide b* **using** *assms set-def* **by** *auto*
  hence *setp* (*set b*) **by** *simp*
  hence $a \in Univ \wedge set\ a \subseteq set\ b$
    using *setp-imp-subset-Univ assms elem-set-implies-set-eq-singleton* **by** *auto*
  hence $ide\ a \wedge set\ a \subseteq set\ b$
    using *b terminal-char1* **by** *simp*
  thus *?thesis* **using** *b incl-in-def* **by** *simp*
qed

lemma *incl-incl-of* [*simp*]:
assumes *incl-in a b*
shows *incl* (*incl-of a b*)
and «*incl-of a b* : $a \to b$»
proof −
  show «*incl-of a b* : $a \to b$»
    using *assms incl-in-def mkArr-in-hom*
    by (*metis image-ident image-subset-iff-funcset mkIde-set setp-set-ide*)
  thus *incl* (*incl-of a b*)
    using *assms incl-def incl-in-def* **by** *fastforce*
qed

There is at most one inclusion between any pair of objects.

lemma *incls-coherent*:
assumes *par f f'* **and** *incl f* **and** *incl f'*
shows $f = f'$
  using *assms incl-def fun-complete* **by** *auto*

The set of inclusions is closed under composition.

lemma *incl-comp* [*simp*]:
assumes *incl f* **and** *incl g* **and** $cod\ f = dom\ g$
shows *incl* $(g \cdot f)$
proof −
  have *1*: *seq g f* **using** *assms incl-def* **by** *auto*
  moreover have *2*: $Dom\ (g \cdot f) \subseteq Cod\ (g \cdot f)$
    using *assms 1 incl-def* **by** *auto*
  moreover have $g \cdot f = mkArr\ (Dom\ f)\ (Cod\ g)\ (restrict\ (\lambda x.\ x)\ (Dom\ f))$
  proof (*intro arr-eqI*$_{SC}$)
    have *3*: *arr* (*mkArr* (*Dom f*) (*Cod g*) ($\lambda x{\in}Dom\ f.\ x$))
      by (*metis 1 2 cod-comp dom-comp ide-cod ide-dom incl-def incl-in-def*
          incl-incl-of(*1*) *mkArr-restrict-eq*)
    show *4*: *par* $(g \cdot f)$ (*mkArr* (*Dom f*) (*Cod g*) ($\lambda x{\in}Dom\ f.\ x$))
      using *assms 1 3 mkIde-set* **by** *auto*
    show $Fun\ (g \cdot f) = Fun\ (mkArr\ (Dom\ f)\ (Cod\ g)\ (\lambda x{\in}Dom\ f.\ x))$

using *assms 3 4 Fun-comp Fun-mkArr*
by (*metis comp-cod-arr dom-cod ide-cod ide-implies-incl incl-def mkArr-restrict-eq'*)
qed
ultimately show *?thesis* **using** *incl-def* **by** *force*
qed

9.4.4 Image Factorization

The image of an arrow is the object that corresponds to the set-theoretic image of the domain set under the function induced by the arrow.

abbreviation *Img*
where *Img f* ≡ *Fun f ‘ Dom f*

definition *img*
where *img f* = *mkIde* (*Img f*)

lemma *ide-img* [*simp*]:
assumes *arr f*
shows *ide* (*img f*)
proof −
have *Fun f ‘ Dom f* ⊆ *Cod f* **using** *assms Fun-mapsto* **by** *blast*
moreover have *setp* (*Cod f*) **using** *assms* **by** *simp*
ultimately show *?thesis* **using** *img-def setp-respects-subset* **by** *auto*
qed

lemma *set-img* [*simp*]:
assumes *arr f*
shows *set* (*img f*) = *Img f*
proof −
have *1*: *Img f* ⊆ *Cod f* ∧ *setp* (*set* (*cod f*))
using *assms Fun-mapsto* **by** *auto*
hence *Fun f ‘ set* (*dom f*) ⊆ *Univ*
using *setp-imp-subset-Univ* **by** *blast*
thus *?thesis*
using *assms 1 img-def set-mkIde setp-respects-subset* **by** *auto*
qed

lemma *img-point-in-Univ*:
assumes «*x* : *unity* → *a*»
shows *img x* ∈ *Univ*
proof −
have *set* (*img x*) = {*Fun x unity*}
using *assms terminal-char2 terminal-unity*$_{SC}$ **by** *auto*
thus *img x* ∈ *Univ* **using** *assms terminal-char1* **by** *auto*
qed

lemma *incl-in-img-cod*:
assumes *arr f*
shows *incl-in* (*img f*) (*cod f*)

proof (*unfold img-def*)
have *1*: *Img f* ⊆ *Cod f* ∧ *setp* (*Cod f*)
using *assms Fun-mapsto* **by** *auto*
hence *2*: *ide* (*mkIde* (*Img f*))
using *setp-respects-subset* **by** *auto*
moreover have *ide* (*cod f*) **using** *assms* **by** *auto*
moreover have *set* (*mkIde* (*Img f*)) ⊆ *Cod f*
using *1 2* **using** *setp-respects-subset* **by** *force*
ultimately show *incl-in* (*mkIde* (*Img f*)) (*cod f*)
using *assms incl-in-def ide-cod* **by** *blast*
qed

lemma *img-point-elem-set*:
assumes «*x* : *unity* → *a*»
shows *img x* ∈ *set a*
by (*metis assms img-point-in-Univ in-homE incl-in-img-cod insert-subset*
mem-Collect-eq incl-in-def terminal-char2)

The corestriction of an arrow *f* is the arrow *corestr f* ∈ *hom* (*dom f*) (*img f*) that induces the same function on the universe as *f*.

definition *corestr*
where *corestr f* = *mkArr* (*Dom f*) (*Img f*) (*Fun f*)

lemma *corestr-in-hom*:
assumes *arr f*
shows «*corestr f* : *dom f* → *img f*»
by (*metis assms corestr-def equalityD2 ide-dom ide-img image-subset-iff-funcset*
mkIde-set set-img mkArr-in-hom setp-set-ide)

Every arrow factors as a corestriction followed by an inclusion.

lemma *img-fact*:
assumes *arr f*
shows *S* (*incl-of* (*img f*) (*cod f*)) (*corestr f*) = *f*
proof (*intro arr-eqI*$_{SC}$)
have *1*: «*corestr f* : *dom f* → *img f*»
using *assms corestr-in-hom* **by** *blast*
moreover have *2*: «*incl-of* (*img f*) (*cod f*) : *img f* → *cod f*»
using *assms incl-in-img-cod incl-incl-of* **by** *fast*
ultimately show *P*: *par* (*incl-of* (*img f*) (*cod f*) · *corestr f*) *f*
using *assms in-homE* **by** *blast*
show *Fun* (*incl-of* (*img f*) (*cod f*) · *corestr f*) = *Fun f*
by (*metis* (*no-types*, *lifting*) *1 2 Fun-comp Fun-ide Fun-mkArr P comp-cod-arr*
corestr-def ide-img in-homE mkArr-Fun)
qed

lemma *Fun-corestr*:
assumes *arr f*
shows *Fun* (*corestr f*) = *Fun f*
by (*metis Fun-mkArr arrI assms corestr-def corestr-in-hom mkArr-Fun*)

9.4.5 Points and Terminal Objects

To each element t of *set a* is associated a point *mkPoint a t* $\in$ *hom unity a*. The function induced by such a point is the constant-t function on the set $\{unity\}$.

definition *mkPoint*
where *mkPoint a t* $\equiv$ *mkArr* $\{unity\}$ (*set a*) (λ-. *t*)

lemma *mkPoint-in-hom*:
assumes *ide a* **and** *t* $\in$ *set a*
shows «*mkPoint a t* : *unity* $\to$ *a*»
  using *assms mkArr-in-hom*
  by (*metis Pi-I mkIde-set setp-set-ide terminal-char2 terminal-unity*$_{SC}$ *mkPoint-def*)

lemma *Fun-mkPoint*:
assumes *ide a* **and** *t* $\in$ *set a*
shows *Fun* (*mkPoint a t*) = (λ- $\in$ $\{unity\}$. *t*)
  using *assms mkPoint-def terminal-unity*$_{SC}$ *mkPoint-in-hom* **by** *fastforce*

For each object a the function *mkPoint a* has as its inverse the restriction of the function *img* to *hom unity a*

lemma *mkPoint-img*:
shows *img* $\in$ *hom unity a* $\to$ *set a*
and $\bigwedge x$. «*x* : *unity* $\to$ *a*» $\Longrightarrow$ *mkPoint a* (*img x*) = *x*
proof −
  show *img* $\in$ *hom unity a* $\to$ *set a*
    using *img-point-elem-set* **by** *simp*
  show $\bigwedge x$. «*x* : *unity* $\to$ *a*» $\Longrightarrow$ *mkPoint a* (*img x*) = *x*
  proof −
    fix *x*
    assume *x*: «*x* : *unity* $\to$ *a*»
    show *mkPoint a* (*img x*) = *x*
    proof (*intro arr-eqI*$_{SC}$)
      have *0*: *img x* $\in$ *set a*
        using *x img-point-elem-set* **by** *metis*
      hence *1*: *mkPoint a* (*img x*) $\in$ *hom unity a*
        using *x mkPoint-in-hom* **by** *force*
      thus *2*: *par* (*mkPoint a* (*img x*)) *x*
        using *x* **by** *fastforce*
      have *Fun* (*mkPoint a* (*img x*)) = (λ- $\in$ $\{unity\}$. *img x*)
        using *1 mkPoint-def* **by** *auto*
      also have ... = *Fun x*
        by (*metis 0 Fun-corestr calculation elem-set-implies-set-eq-singleton*
            ide-cod ide-unity in-homE mem-Collect-eq Fun-mkPoint corestr-in-hom
            img-point-in-Univ mkPoint-in-hom singletonI terminalE x)
      finally show *Fun* (*mkPoint a* (*img x*)) = *Fun x* **by** *auto*
    qed
  qed
qed

lemma *img-mkPoint*:
assumes *ide a*
shows *mkPoint a ∈ set a → hom unity a*
and $\bigwedge t.\ t \in set\ a \Longrightarrow img\ (mkPoint\ a\ t) = t$
proof −
  show *mkPoint a ∈ set a → hom unity a*
    using *assms(1) mkPoint-in-hom* **by** *simp*
  show $\bigwedge t.\ t \in set\ a \Longrightarrow img\ (mkPoint\ a\ t) = t$
    proof −
    fix *t*
    assume *t*: *t ∈ set a*
    show *img (mkPoint a t) = t*
    proof −
      have *1*: *arr (mkPoint a t)*
        using *assms t mkPoint-in-hom* **by** *auto*
      have *Fun (mkPoint a t) ‘ {unity} = {t}*
        using *1 mkPoint-def* **by** *simp*
      thus *?thesis*
        by (*metis in-homE img-def mkIde-set mkPoint-in-hom elem-set-implies-incl-in*
            elem-set-implies-set-eq-singleton incl-in-def t terminal-char2 terminal-unity$_{SC}$)
    qed
  qed
qed

For each object a the elements of *hom unity a* are therefore in bijective correspondence with *set a*.

lemma *bij-betw-points-and-set*:
assumes *ide a*
shows *bij-betw img (hom unity a) (set a)*
proof (*intro bij-betwI*)
  show *img ∈ hom unity a → set a*
    using *assms mkPoint-img* **by** *auto*
  show *mkPoint a ∈ set a → hom unity a*
    using *assms img-mkPoint* **by** *auto*
  show $\bigwedge x.\ x \in hom\ unity\ a \Longrightarrow mkPoint\ a\ (img\ x) = x$
    using *assms mkPoint-img* **by** *auto*
  show $\bigwedge t.\ t \in set\ a \Longrightarrow img\ (mkPoint\ a\ t) = t$
    using *assms img-mkPoint* **by** *auto*
qed

lemma *setp-img-points*:
assumes *ide a*
shows *setp (img ‘ hom unity a)*
  using *assms*
  by (*metis image-subset-iff-funcset mkPoint-img(1) setp-respects-subset setp-set-ide*)

The function on the universe induced by an arrow f agrees, under the bijection between *hom unity (dom f)* and *Dom f*, with the action of f by composition on *hom unity (dom f)*.

lemma *Fun-point*:
assumes «x : $unity \to a$»
shows $Fun\ x = (\lambda\text{-} \in \{unity\}.\ img\ x)$
using *assms mkPoint-img img-mkPoint Fun-mkPoint* [*of a img x*] *img-point-elem-set*
by *auto*

lemma *comp-arr-mkPoint*:
assumes *arr f* **and** $t \in Dom\ f$
shows $f \cdot mkPoint\ (dom\ f)\ t = mkPoint\ (cod\ f)\ (Fun\ f\ t)$
proof (*intro arr-eqI*$_{SC}$)
have *0*: *seq f* (*mkPoint* (*dom f*) *t*)
using *assms mkPoint-in-hom* [*of dom f t*] **by** *auto*
have *1*: «$f \cdot mkPoint\ (dom\ f)\ t : unity \to cod\ f$»
using *assms mkPoint-in-hom* [*of dom f t*] **by** *auto*
show *par* ($f \cdot mkPoint\ (dom\ f)\ t$) (*mkPoint* (*cod f*) (*Fun f t*))
proof −
have «$mkPoint\ (cod\ f)\ (Fun\ f\ t) : unity \to cod\ f$»
using *assms Fun-mapsto mkPoint-in-hom* [*of cod f Fun f t*] **by** *auto*
thus *?thesis* **using** *1* **by** *fastforce*
qed
show $Fun\ (f \cdot mkPoint\ (dom\ f)\ t) = Fun\ (mkPoint\ (cod\ f)\ (Fun\ f\ t))$
proof −
have $Fun\ (f \cdot mkPoint\ (dom\ f)\ t) = restrict\ (Fun\ f\ o\ Fun\ (mkPoint\ (dom\ f)\ t))\ \{unity\}$
using *assms 0 1 Fun-comp terminal-char2 terminal-unity*$_{SC}$ **by** *auto*
also have ... = ($\lambda\text{-} \in \{unity\}.\ Fun\ f\ t$)
using *assms Fun-mkPoint* **by** *auto*
also have ... = *Fun* (*mkPoint* (*cod f*) (*Fun f t*))
using *assms Fun-mkPoint* [*of cod f Fun f t*] *Fun-mapsto* **by** *fastforce*
finally show *?thesis* **by** *auto*
qed
qed

lemma *comp-arr-point*$_{SSC}$:
assumes *arr f* **and** «x : $unity \to dom\ f$»
shows $f \cdot x = mkPoint\ (cod\ f)\ (Fun\ f\ (img\ x))$
by (*metis assms comp-arr-mkPoint img-point-elem-set mkPoint-img*(*2*))

This agreement allows us to express *Fun f* in terms of composition.

lemma *Fun-in-terms-of-comp*:
assumes *arr f*
shows $Fun\ f = restrict\ (img\ o\ S\ f\ o\ mkPoint\ (dom\ f))\ (Dom\ f)$
proof
fix *t*
have $t \notin Dom\ f \Longrightarrow Fun\ f\ t = restrict\ (img\ o\ S\ f\ o\ mkPoint\ (dom\ f))\ (Dom\ f)\ t$
using *assms* **by** (*simp add*: *Fun-def*)
moreover have $t \in Dom\ f \Longrightarrow$
$Fun\ f\ t = restrict\ (img\ o\ S\ f\ o\ mkPoint\ (dom\ f))\ (Dom\ f)\ t$
proof −
assume *t*: $t \in Dom\ f$

```
    have 1: f · mkPoint (dom f) t = mkPoint (cod f) (Fun f t)
      using assms t comp-arr-mkPoint by simp
    hence img (f · mkPoint (dom f) t) = img (mkPoint (cod f) (Fun f t)) by simp
    thus ?thesis
    proof −
      have Fun f t ∈ Cod f using assms t Fun-mapsto by auto
      thus ?thesis using assms t 1 img-mkPoint by auto
    qed
  qed
  ultimately show Fun f t = restrict (img o S f o mkPoint (dom f)) (Dom f) t by auto
qed
```

We therefore obtain a rule for proving parallel arrows equal by showing that they have the same action by composition on points.

```
lemma arr-eqI'_SC:
assumes par f f' and ⋀x. «x : unity → dom f» ⟹ f · x = f' · x
shows f = f'
  using assms Fun-in-terms-of-comp mkPoint-in-hom by (intro arr-eqI_SC, auto)
```

An arrow can therefore be specified by giving its action by composition on points. In many situations, this is more natural than specifying it as a function on the universe.

```
definition mkArr'
where mkArr' a b F = mkArr (set a) (set b) (img o F o mkPoint a)

lemma mkArr'-in-hom:
assumes ide a and ide b and F ∈ hom unity a → hom unity b
shows «mkArr' a b F : a → b»
proof −
  have img o F o mkPoint a ∈ set a → set b
    using assms(1,3) img-mkPoint(1) mkPoint-img(1) by fastforce
  thus ?thesis
    using assms mkArr'-def mkArr-in-hom [of set a set b] mkIde-set by simp
qed

lemma comp-point-mkArr':
assumes ide a and ide b and F ∈ hom unity a → hom unity b
shows ⋀x. «x : unity → a» ⟹ mkArr' a b F · x = F x
proof −
  fix x
  assume x: «x : unity → a»
  have Fun (mkArr' a b F) (img x) = img (F x)
    unfolding mkArr'-def
    using assms x Fun-mkArr img-point-elem-set mkPoint-img mkPoint-in-hom
    by (simp add: Pi-iff)
  hence mkArr' a b F · x = mkPoint b (img (F x))
    using assms x mkArr'-in-hom [of a b F] comp-arr-point_SSC by auto
  thus mkArr' a b F · x = F x
    using assms x mkPoint-img(2) by auto
qed
```

A third characterization of terminal objects is as those objects whose set of points is a singleton.

lemma *terminal-char3*:
assumes $\exists!x.\ \langle x : unity \to a\rangle$
shows *terminal a*
proof −
  have *a*: *ide a*
    using *assms ide-cod mem-Collect-eq* **by** *blast*
  hence *bij-betw img* (*hom unity a*) (*set a*)
    using *assms bij-betw-points-and-set* **by** *auto*
  hence *img* ‘ (*hom unity a*) = *set a*
    by (*simp add*: *bij-betw-def*)
  moreover have *hom unity a* = {*THE x*. *x* ∈ *hom unity a*}
    using *assms theI'* [*of* λ*x*. *x* ∈ *hom unity a*] **by** *auto*
  ultimately have *set a* = {*img* (*THE x*. *x* ∈ *hom unity a*)}
    by (*metis image-empty image-insert*)
  thus *?thesis* **using** *a terminal-char1* **by** *simp*
qed

The following is an alternative formulation of functional completeness, which says that any function on points uniquely determines an arrow.

lemma *fun-complete'*:
assumes *ide a* **and** *ide b* **and** $F \in hom\ unity\ a \to hom\ unity\ b$
shows $\exists!f.\ \langle f : a \to b\rangle \wedge (\forall x.\ \langle x : unity \to a\rangle \longrightarrow f \cdot x = F\ x)$
proof
  have *1*: $\langle mkArr'\ a\ b\ F : a \to b\rangle$ **using** *assms mkArr'-in-hom* **by** *auto*
  moreover have *2*: $\bigwedge x.\ \langle x : unity \to a\rangle \Longrightarrow mkArr'\ a\ b\ F \cdot x = F\ x$
    using *assms comp-point-mkArr'* **by** *auto*
  ultimately show $\langle mkArr'\ a\ b\ F : a \to b\rangle \wedge$
                $(\forall x.\ \langle x : unity \to a\rangle \longrightarrow mkArr'\ a\ b\ F \cdot x = F\ x)$ **by** *blast*
  fix *f*
  assume *f*: $\langle f : a \to b\rangle \wedge (\forall x.\ \langle x : unity \to a\rangle \longrightarrow f \cdot x = F\ x)$
  show *f* = *mkArr' a b F*
    using *f 1 2* **by** (*intro arr-eqI'*$_{SC}$ [*of f mkArr' a b F*], *fastforce*, *auto*)
qed

9.4.6 The ‘Determines Same Function’ Relation on Arrows

An important part of understanding the structure of a category of sets and functions is to characterize when it is that two arrows “determine the same function”. The following result provides one answer to this: two arrows with a common domain determine the same function if and only if they can be rendered equal by composing with a cospan of inclusions.

lemma *eq-Fun-iff-incl-joinable*:
assumes *span f f'*
shows *Fun f* = *Fun f'* ⟷
      $(\exists m\ m'.\ incl\ m \wedge incl\ m' \wedge seq\ m\ f \wedge seq\ m'\ f' \wedge m \cdot f = m' \cdot f')$
proof

assume *ff'*: *Fun f = Fun f'*
let *?b = mkIde (Cod f ∪ Cod f')*
let *?m = incl-of (cod f) ?b*
let *?m' = incl-of (cod f') ?b*
have *incl-m*: *incl ?m*
using *assms incl-incl-of [of cod f ?b] incl-in-def setp-respects-union* **by** *simp*
have *incl-m'*: *incl ?m'*
using *assms incl-incl-of [of cod f' ?b] incl-in-def setp-respects-union* **by** *simp*
have *m*: *?m = mkArr (Cod f) (Cod f ∪ Cod f') (λx. x)*
using *setp-respects-union* **by** (*simp add*: *assms*)
have *m'*: *?m' = mkArr (Cod f') (Cod f ∪ Cod f') (λx. x)*
using *setp-respects-union* **by** (*simp add*: *assms*)
have *seq*: *seq ?m f ∧ seq ?m' f'*
using *assms m m'* **using** *setp-respects-union* **by** *simp*
have *?m · f = ?m' · f'*
by (*metis assms comp-mkArr ff' incl-def incl-m incl-m' mkArr-Fun*)
hence *incl ?m ∧ incl ?m' ∧ seq ?m f ∧ seq ?m' f' ∧ ?m · f = ?m' · f'*
using *seq ‹incl ?m› ‹incl ?m'›* **by** *simp*
thus *∃ m m'. incl m ∧ incl m' ∧ seq m f ∧ seq m' f' ∧ m · f = m' · f'* **by** *auto*
next
assume *ff'*: *∃ m m'. incl m ∧ incl m' ∧ seq m f ∧ seq m' f' ∧ m · f = m' · f'*
show *Fun f = Fun f'*
using *ff'*
by (*metis Fun-comp Fun-ide comp-cod-arr ide-cod seqE Fun-incl*)
qed

Another answer to the same question: two arrows with a common domain determine the same function if and only if their corestrictions are equal.

lemma *eq-Fun-iff-eq-corestr*:
assumes *span f f'*
shows *Fun f = Fun f' ⟷ corestr f = corestr f'*
using *assms corestr-def Fun-corestr* **by** *metis*

9.4.7 Retractions, Sections, and Isomorphisms

An arrow is a retraction if and only if its image coincides with its codomain.

lemma *retraction-if-Img-eq-Cod*:
assumes *arr g* **and** *Img g = Cod g*
shows *retraction g*
and *ide (g · mkArr (Cod g) (Dom g) (inv-into (Dom g) (Fun g)))*
proof −
let *?F = inv-into (Dom g) (Fun g)*
let *?f = mkArr (Cod g) (Dom g) ?F*
have *f*: *arr ?f*
by (*simp add*: *assms inv-into-into*)
show *ide (g · ?f)*
proof −
have *g = mkArr (Dom g) (Cod g) (Fun g)* **using** *assms mkArr-Fun* **by** *auto*
hence *g · ?f = mkArr (Cod g) (Cod g) (Fun g o ?F)*

```
    using assms(1) f comp-mkArr by metis
  moreover have mkArr (Cod g) (Cod g) (λy. y) = ...
  proof (intro mkArr-eqI')
    show arr (mkArr (Cod g) (Cod g) (λy. y))
      using assms arr-cod-iff-arr by auto
    show ⋀y. y ∈ Cod g ⟹ y = (Fun g o ?F) y
      using assms by (simp add: f-inv-into-f)
  qed
  ultimately show ?thesis
    using assms f mkIde-as-mkArr by auto
 qed
 thus retraction g by auto
qed

lemma retraction-char:
shows retraction g ⟷ arr g ∧ Img g = Cod g
proof
 assume G: retraction g
 show arr g ∧ Img g = Cod g
 proof
  show arr g using G by blast
  show Img g = Cod g
  proof −
   from G obtain f where f: ide (g · f) by blast
   have Fun g ‘ Fun f ‘ Cod g = Cod g
   proof −
    have restrict (Fun g o Fun f) (Cod g) = restrict (λx. x) (Cod g)
     using f Fun-comp Fun-ide ide-compE by metis
    thus ?thesis
     by (metis image-comp image-ident image-restrict-eq)
   qed
   moreover have Fun f ‘ Cod g ⊆ Dom g
    using f Fun-mapsto arr-mkArr mkArr-Fun funcset-image
    by (metis seqE ide-compE ide-compE)
   moreover have Img g ⊆ Cod g
    using f Fun-mapsto by blast
   ultimately show ?thesis by blast
  qed
 qed
 next
 assume arr g ∧ Img g = Cod g
 thus retraction g using retraction-if-Img-eq-Cod by blast
qed
```

Every corestriction is a retraction.

```
lemma retraction-corestr:
assumes arr f
shows retraction (corestr f)
 using assms retraction-char Fun-corestr corestr-in-hom in-homE set-img
```

by *metis*

An arrow is a section if and only if it induces an injective function on its domain, except in the special case that it has an empty domain set and a nonempty codomain set.

lemma *section-if-inj*:
assumes *arr f* **and** *inj-on* (*Fun f*) (*Dom f*) **and** *Dom f* = {} ⟶ *Cod f* = {}
shows *section f*
and *ide* (*mkArr* (*Cod f*) (*Dom f*)
(λ*y*. *if y* ∈ *Img f then SOME x*. *x* ∈ *Dom f* ∧ *Fun f x* = *y*
else SOME x. *x* ∈ *Dom f*)
· *f*)
proof −
let *?P*= λ*y*. λ*x*. *x* ∈ *Dom f* ∧ *Fun f x* = *y*
let *?G* = λ*y*. *if y* ∈ *Img f then SOME x*. *?P y x else SOME x*. *x* ∈ *Dom f*
let *?g* = *mkArr* (*Cod f*) (*Dom f*) *?G*
have *g*: *arr ?g*
proof −
have *1*: *setp* (*Cod f*) **using** *assms* **by** *simp*
have *2*: *setp* (*Dom f*) **using** *assms* **by** *simp*
have *3*: *?G* ∈ *Cod f* → *Dom f*
proof
fix *y*
assume *Y*: *y* ∈ *Cod f*
show *?G y* ∈ *Dom f*
proof (*cases y* ∈ *Img f*)
assume *y* ∈ *Img f*
hence (∃ *x*. *?P y x*) ∧ *?G y* = (*SOME x*. *?P y x*) **using** *Y* **by** *auto*
hence *?P y* (*?G y*) **using** *someI-ex* [*of ?P y*] **by** *argo*
thus *?G y* ∈ *Dom f* **by** *auto*
next
assume *y* ∉ *Img f*
hence (∃ *x*. *x* ∈ *Dom f*) ∧ *?G y* = (*SOME x*. *x* ∈ *Dom f*) **using** *assms Y* **by** *auto*
thus *?G y* ∈ *Dom f* **using** *someI-ex* [*of* λ*x*. *x* ∈ *Dom f*] **by** *argo*
qed
qed
show *?thesis* **using** *1 2 3* **by** *simp*
qed
show *ide* (*?g* · *f*)
proof −
have *f* = *mkArr* (*Dom f*) (*Cod f*) (*Fun f*) **using** *assms mkArr-Fun* **by** *auto*
hence *?g* · *f* = *mkArr* (*Dom f*) (*Dom f*) (*?G o Fun f*)
using *assms*(*1*) *g comp-mkArr* [*of Dom f Cod f Fun f Dom f ?G*] **by** *argo*
moreover have *mkArr* (*Dom f*) (*Dom f*) (λ*x*. *x*) = ...
proof (*intro mkArr-eqI'*)
show *arr* (*mkArr* (*Dom f*) (*Dom f*) (λ*x*. *x*))
using *assms* **by** *auto*
show ⋀*x*. *x* ∈ *Dom f* ⟹ *x* = (*?G o Fun f*) *x*
proof −

```
      fix x
      assume x: x ∈ Dom f
      have Fun f x ∈ Img f using x by blast
      hence ∗: (∃ x'. ?P (Fun f x) x') ∧ ?G (Fun f x) = (SOME x'. ?P (Fun f x) x')
        by auto
      then have ?P (Fun f x) (?G (Fun f x))
        using someI-ex [of ?P (Fun f x)] by argo
      with ∗ have x = ?G (Fun f x)
        using assms x inj-on-def [of Fun f Dom f] by simp
      thus x = (?G o Fun f) x by simp
    qed
  qed
  ultimately show ?thesis
    using assms mkIde-as-mkArr mkIde-set by auto
 qed
 thus section f by auto
qed

lemma section-char:
shows section f ⟷ arr f ∧ (Dom f = {} ⟶ Cod f = {}) ∧ inj-on (Fun f) (Dom f)
proof
 assume f: section f
 from f obtain g where g: ide (g · f) using section-def by blast
 show arr f ∧ (Dom f = {} ⟶ Cod f = {}) ∧ inj-on (Fun f) (Dom f)
 proof −
  have arr f using f by blast
  moreover have Dom f = {} ⟶ Cod f = {}
  proof −
   have Cod f ≠ {} ⟶ Dom f ≠ {}
   proof
    assume Cod f ≠ {}
    from this obtain y where y ∈ Cod f by blast
    hence Fun g y ∈ Dom f
      using g Fun-mapsto
      by (metis seqE ide-compE image-eqI retractionI retraction-char)
    thus Dom f ≠ {} by blast
   qed
   thus ?thesis by auto
  qed
  moreover have inj-on (Fun f) (Dom f)
   by (metis Fun-comp Fun-ide g ide-compE inj-on-id2 inj-on-imageI2 inj-on-restrict-eq)
  ultimately show ?thesis by auto
 qed
 next
 assume F: arr f ∧ (Dom f = {} ⟶ Cod f = {}) ∧ inj-on (Fun f) (Dom f)
 thus section f using section-if-inj by auto
qed
```

Section-retraction pairs can also be characterized by an inverse relationship between the functions they induce.

lemma *section-retraction-char*:
shows *ide* $(g \cdot f) \longleftrightarrow$ *antipar f g* $\wedge$ *compose* (*Dom f*) (*Fun g*) (*Fun f*) $= (\lambda x \in Dom\ f.\ x)$
by (*metis Fun-comp cod-comp compose-eq′ dom-comp ide-char*$_{SC}$ *ide-compE seqE*)

Antiparallel arrows f and g are inverses if the functions they induce are inverses.

lemma *inverse-arrows-char*:
shows *inverse-arrows f g* $\longleftrightarrow$
antipar f g $\wedge$ *compose* (*Dom f*) (*Fun g*) (*Fun f*) $= (\lambda x \in Dom\ f.\ x)$
$\wedge$ *compose* (*Dom g*) (*Fun f*) (*Fun g*) $= (\lambda y \in Dom\ g.\ y)$
using *section-retraction-char* **by** *blast*

An arrow is an isomorphism if and only if the function it induces is a bijection.

lemma *iso-char*:
shows *iso f* $\longleftrightarrow$ *arr f* $\wedge$ *bij-betw* (*Fun f*) (*Dom f*) (*Cod f*)
by (*metis bij-betw-def image-empty iso-iff-section-and-retraction retraction-char section-char*)

The inverse of an isomorphism is constructed by inverting the induced function.

lemma *inv-char*:
assumes *iso f*
shows *inv f* $=$ *mkArr* (*Cod f*) (*Dom f*) (*inv-into* (*Dom f*) (*Fun f*))
proof −
let *?g* $=$ *mkArr* (*Cod f*) (*Dom f*) (*inv-into* (*Dom f*) (*Fun f*))
have *ide* (*f* $\cdot$ *?g*)
using *assms iso-is-retraction retraction-char retraction-if-Img-eq-Cod* **by** *simp*
moreover have *ide* (*?g* $\cdot$ *f*)
proof −
let *?g′* $=$ *mkArr* (*Cod f*) (*Dom f*)
$(\lambda y.$ *if* $y \in Img\ f$ *then SOME* $x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
else SOME $x.\ x \in Dom\ f)$
have *1*: *ide* (*?g′* $\cdot$ *f*)
using *assms iso-is-section section-char section-if-inj* **by** *simp*
moreover have *?g′* $=$ *?g*
proof
show *arr ?g′* **using** *1 ide-compE* **by** *blast*
show $\bigwedge y.\ y \in Cod\ f \Longrightarrow$ (*if* $y \in Img\ f$ *then SOME* $x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
else SOME $x.\ x \in Dom\ f$)
$=$ *inv-into* (*Dom f*) (*Fun f*) *y*
proof −
fix *y*
assume $y \in Cod\ f$
hence $y \in Img\ f$ **using** *assms iso-is-retraction retraction-char* **by** *metis*
thus (*if* $y \in Img\ f$ *then SOME* $x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
else SOME $x.\ x \in Dom\ f$)
$=$ *inv-into* (*Dom f*) (*Fun f*) *y*
using *inv-into-def* **by** *metis*
qed
qed
ultimately show *?thesis* **by** *auto*

```
  qed
  ultimately have inverse-arrows f ?g by auto
  thus ?thesis using inverse-unique by blast
qed

lemma Fun-inv:
assumes iso f
shows Fun (inv f) = restrict (inv-into (Dom f) (Fun f)) (Cod f)
  using assms inv-in-hom inv-char iso-inv-iso iso-is-arr Fun-mkArr by metis
```

9.4.8 Monomorphisms and Epimorphisms

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

```
lemma mono-char:
shows mono f ⟷ arr f ∧ inj-on (Fun f) (Dom f)
proof
  assume f: mono f
  hence arr f using mono-def by auto
  moreover have inj-on (Fun f) (Dom f)
  proof (intro inj-onI)
    have 0: inj-on (S f) (hom unity (dom f))
    proof −
      have hom unity (dom f) ⊆ {g. seq f g}
        using f mono-def arrI by auto
      hence ∃ A. hom unity (dom f) ⊆ A ∧ inj-on (S f) A
        using f mono-def by auto
      thus ?thesis
        by (meson subset-inj-on)
    qed
    fix x x′
    assume x: x ∈ Dom f and x′: x′ ∈ Dom f and xx′: Fun f x = Fun f x′
    have mkPoint (dom f) x ∈ hom unity (dom f) ∧
         mkPoint (dom f) x′ ∈ hom unity (dom f)
      using x x′ ‹arr f› mkPoint-in-hom by simp
    moreover have f · mkPoint (dom f) x = f · mkPoint (dom f) x′
      using ‹arr f› x x′ xx′ comp-arr-mkPoint by simp
    ultimately have mkPoint (dom f) x = mkPoint (dom f) x′
      using 0 inj-onD [of S f hom unity (dom f) mkPoint (dom f) x] by simp
    thus x = x′
      using ‹arr f› x x′ img-mkPoint(2) img-mkPoint(2) ide-dom by metis
  qed
  ultimately show arr f ∧ inj-on (Fun f) (Dom f) by auto
  next
  assume f: arr f ∧ inj-on (Fun f) (Dom f)
  show mono f
  proof
    show arr f using f by auto
    show ⋀g g′. ⟦seq f g; f · g = f · g′⟧ ⟹ g = g′
    proof −
```

```
      fix g g'
      assume fg: seq f g and eq: f · g = f · g'
      show g = g'
      proof (intro arr-eqI_SC)
        show par: par g g'
          using fg eq dom-comp by (metis seqE)
        show Fun g = Fun g'
          by (metis empty-is-image eq f fg ide-dom incl-in-def incl-in-img-cod
              initial-arr-unique initial-empty empty-def mono-cancel mkIde-set
              section-if-inj(1) section-is-mono seqE set-img subset-empty)
      qed
    qed
  qed
qed
```

Inclusions are monomorphisms.

```
lemma mono-imp-incl:
assumes incl f
shows mono f
  using assms incl-def Fun-incl mono-char by auto
```

A monomorphism is a section, except in case it has an empty domain set and a nonempty codomain set.

```
lemma mono-imp-section:
assumes mono f and Dom f = {} ⟶ Cod f = {}
shows section f
  using assms mono-char section-char by auto
```

An arrow is an epimorphism if and only if either its image coincides with its codomain, or else the universe has only a single element (in which case all arrows are epimorphisms).

```
lemma epi-char:
shows epi f ⟷ arr f ∧ (Img f = Cod f ∨ (∀ t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t'))
proof
  assume epi: epi f
  show arr f ∧ (Img f = Cod f ∨ (∀ t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t'))
  proof −
    have f: arr f using epi epi-implies-arr by auto
    moreover have ¬(∀ t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t') ⟹ Img f = Cod f
    proof −
      assume ¬(∀ t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t')
      from this obtain tt and ff
        where B: tt ∈ Univ ∧ ff ∈ Univ ∧ tt ≠ ff by blast
      show Img f = Cod f
      proof
        show Img f ⊆ Cod f using f Fun-mapsto by auto
        show Cod f ⊆ Img f
        proof
          let ?g = mkArr (Cod f) {ff, tt} (λy. tt)
          let ?g' = mkArr (Cod f) {ff, tt} (λy. if ∃ x. x ∈ Dom f ∧ Fun f x = y
```

```
                                    then tt else ff)
        let ?b = mkIde {ff, tt}
        have b: ide ?b
          by (metis B finite.emptyI finite-imp-setp finite-insert ide-mkIde
              insert-subset setp-imp-subset-Univ setp-singleton mem-Collect-eq)
        have g: «?g : cod f → ?b» ∧ Fun ?g = (λy ∈ Cod f. tt)
          using f B in-homI [of ?g cod f mkIde {ff, tt}] finite-imp-setp by simp
        have g': ?g' ∈ hom (cod f) ?b ∧
                 Fun ?g' = (λy ∈ Cod f. if ∃x. x ∈ Dom f ∧ Fun f x = y then tt else ff)
          using f B in-homI [of ?g'] finite-imp-setp by simp
        have ?g · f = ?g' · f
        proof (intro arr-eqI_SC)
          show par (?g · f) (?g' · f)
            using f g g' by auto
          show Fun (?g · f) = Fun (?g' · f)
            using f g g' Fun-comp comp-mkArr by fastforce
        qed
        hence gg': ?g = ?g'
          by (metis (no-types, lifting) epi-cancel epi f g in-homE seqI)
        fix y
        assume y: y ∈ Cod f
        have Fun ?g' y = tt using gg' g y by simp
        hence (if ∃x. x ∈ Dom f ∧ Fun f x = y then tt else ff) = tt
          using g' y by simp
        hence ∃x. x ∈ Dom f ∧ Fun f x = y
          using B by argo
        thus y ∈ Img f by blast
      qed
    qed
  qed
  ultimately show arr f ∧ (Img f = Cod f ∨ (∀t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t'))
    by fast
qed
next
show arr f ∧ (Img f = Cod f ∨ (∀t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t')) ⟹ epi f
proof −
  have arr f ∧ Img f = Cod f ⟹ epi f
    using retraction-char retraction-is-epi by presburger
  moreover have arr f ∧ (∀t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t') ⟹ epi f
  proof −
    assume f: arr f ∧ (∀t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t')
    have ⋀f f'. par f f' ⟹ f = f'
    proof −
      fix f f'
      assume ff': par f f'
      show f = f'
      proof (intro arr-eqI_SC)
        show par f f' using ff' by simp
        have ⋀t t'. t ∈ Cod f ∧ t' ∈ Cod f ⟹ t = t'
```

```
        using f ff' setp-imp-subset-Univ setp-set-ide ide-cod subsetD by blast
      thus Fun f = Fun f'
        using ff' Fun-mapsto [of f] Fun-mapsto [of f']
              extensional-arb [of Fun f Dom f] extensional-arb [of Fun f' Dom f]
        by fastforce
    qed
  qed
  moreover have ⋀g g'. par (g · f) (g' · f) ⟹ par g g'
    by force
  ultimately show epi f
    using f by (intro epiI; metis)
 qed
 ultimately show arr f ∧ (Img f = Cod f ∨  (∀t t'. t ∈ Univ ∧ t' ∈ Univ ⟶ t = t'))
                  ⟹ epi f
   by auto
 qed
qed
```

An epimorphism is a retraction, except in the case of a degenerate universe with only a single element.

```
lemma epi-imp-retraction:
assumes epi f and ∃t t'. t ∈ Univ ∧ t' ∈ Univ ∧ t ≠ t'
shows retraction f
  using assms epi-char retraction-char by auto
```

Retraction/inclusion factorization is unique (not just up to isomorphism – remember that the notion of inclusion is not categorical but depends on the arbitrarily chosen *img*).

```
lemma unique-retr-incl-fact:
assumes seq m e and seq m' e' and m · e = m' · e'
and incl m and incl m' and retraction e and retraction e'
shows m = m' and e = e'
proof −
  have 1: cod m = cod m' ∧ dom e = dom e'
    using assms(1−3) by (metis dom-comp cod-comp)
  hence 2: span e e' using assms(1−2) by blast
  hence 3: Fun e = Fun e'
    using assms eq-Fun-iff-incl-joinable by meson
  hence img e = img e' using assms 1 img-def by auto
  moreover have img e = cod e ∧ img e' = cod e'
    using assms(6−7) retraction-char img-def mkIde-set by simp
  ultimately have par e e' using 2 by simp
  thus e = e' using 3 arr-eqI_SC by blast
  hence par m m' using assms(1) assms(2) 1 by fastforce
  thus m = m' using assms(4) assms(5) incls-coherent by blast
qed

end
```

9.5 Concrete Set Categories

The *set-category* locale is useful for stating results that depend on a category of $'a$-sets and functions, without having to commit to a particular element type $'a$. However, in applications we often need to work with a category of sets and functions that is guaranteed to contain sets corresponding to the subsets of some extrinsically given type $'a$. A *concrete set category* is a set category S that is equipped with an injective function ι from type $'a$ to $S.Univ$. The following locale serves to facilitate some of the technical aspects of passing back and forth between elements of type $'a$ and the elements of $S.Univ$.

locale *concrete-set-category* = *set-category S setp*
  for $S :: 's\ comp$ (**infixr** $\langle\cdot_S\rangle$ 55)
  and *setp* :: $'s\ set \Rightarrow bool$
  and $U :: 'a\ set$
  and $\iota :: 'a \Rightarrow 's$ +
  assumes *UP-mapsto*: $\iota \in U \to Univ$
  and *inj-UP*: *inj-on* ι U
begin

  abbreviation *UP*
  where $UP \equiv \iota$

  abbreviation *DN*
  where $DN \equiv inv\text{-}into\ U\ UP$

  lemma *DN-mapsto*:
  shows $DN \in UP\ `\ U \to U$
    by (*simp add*: *inv-into-into*)

  lemma *DN-UP* [*simp*]:
  assumes $x \in U$
  shows $DN\ (UP\ x) = x$
    using *assms inj-UP inv-into-f-f* **by** *simp*

  lemma *UP-DN* [*simp*]:
  assumes $t \in UP\ `\ U$
  shows $UP\ (DN\ t) = t$
    using *assms o-def inj-UP* **by** *auto*

  lemma *bij-UP*:
  shows *bij-betw UP U* $(UP\ `\ U)$
    using *inj-UP inj-on-imp-bij-betw* **by** *blast*

  lemma *bij-DN*:
  shows *bij-betw DN* $(UP\ `\ U)\ U$
    using *bij-UP bij-betw-inv-into* **by** *blast*

end

locale *replete-concrete-set-category* =
replete-set-category S +
concrete-set-category S ‹λ*A*. *A* ⊆ *Univ*› *U UP*
for *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *U* :: *'a set*
and *UP* :: *'a* ⇒ *'s*

9.6 Sub-Set Categories

In this section, we show that a full subcategory of a set category, obtained by imposing suitable further restrictions on the subsets of the universe that correspond to objects, is again a set category.

locale *sub-set-category* =
S: *set-category* +
fixes *ssetp* :: *'a set* ⇒ *bool*
assumes *ssetp-singleton*: ⋀*t*. *t* ∈ *S.Univ* ⟹ *ssetp* {*t*}
and *subset-closed*: ⋀*B A*. ⟦*B* ⊆ *A*; *ssetp A*⟧ ⟹ *ssetp B*
and *union-closed*: ⋀*A B*. ⟦*ssetp A*; *ssetp B*⟧ ⟹ *ssetp* (*A* ∪ *B*)
and *containment*: ⋀*A*. *ssetp A* ⟹ *setp A*
begin

sublocale *full-subcategory S* ‹λ*a*. *S.ide a* ∧ *ssetp* (*S.set a*)›
by *unfold-locales auto*

lemma *is-full-subcategory*:
shows *full-subcategory S* (λ*a*. *S.ide a* ∧ *ssetp* (*S.set a*))
..

lemma *ide-char*$_{SSC}$:
shows *ide a* ⟷ *S.ide a* ∧ *ssetp* (*S.set a*)
using *ide-char*$_{SbC}$ *arr-char*$_{SbC}$ **by** *fastforce*

lemma *terminal-unity*$_{SSC}$:
shows *terminal S.unity*
proof
show *ide S.unity*
using *S.terminal-unity*$_{SC}$ *S.terminal-def* [*of S.unity*] *S.terminal-char2 ide-char*$_{SSC}$
ssetp-singleton
by *force*
thus ⋀*a*. *ide a* ⟹ ∃!*f*. *in-hom f a S.unity*
using *S.terminal-unity*$_{SC}$ *S.terminal-def ide-char*$_{SbC}$ *ide-char'* *in-hom-char*$_{FSbC}$
by (*metis* (*no-types*, *lifting*))
qed

lemma *terminal-char*:
shows *terminal t* ⟷ *S.terminal t*
proof
fix *t*

```
  assume t: S.terminal t
  have ide t
    using t ssetp-singleton ide-char_SSC S.terminal-char2 by force
  thus terminal t
    using t in-hom-char_FSbC ide-char_SSC arr-char_SbC S.terminal-def terminalI by auto
  next
  assume t: terminal t
  have 1: S.ide t
    using t ide-char_SbC terminal-def by simp
  moreover have card (S.set t) = 1
  proof −
    have card (S.set t) = card (S.hom S.unity t)
      using S.set-def S.inj-img
      by (metis 1 S.bij-betw-points-and-set bij-betw-same-card)
    also have ... = card (hom S.unity t)
      using t in-hom-char_FSbC terminal-def terminal-unity_SSC by auto
    also have ... = 1
    proof −
      have ∃!f. f ∈ hom S.unity t
        using t terminal-def terminal-unity_SSC by force
      moreover have ⋀A. card A = 1 ⟷ (∃!a. a ∈ A)
        apply (intro iffI)
          apply (metis card-1-singletonE empty-iff insert-iff)
        using card-1-singleton-iff by auto
      ultimately show ?thesis by auto
    qed
    finally show ?thesis by blast
  qed
  ultimately show S.terminal t
    using 1 S.terminal-char1 card-1-singleton-iff
    by (metis One-nat-def singleton-iff)
qed

sublocale set-category comp ssetp
proof
```

Here things are simpler if we define *img* appropriately so that we have $set = T.set$ after accounting for the definition $unity \equiv SOME\ t.\ terminal\ t$, which is different from that of S.unity.

```
  have 1: terminal (SOME t. terminal t)
    using terminal-unity_SSC someI-ex [of terminal] by blast
  obtain i where i: «i : S.unity → SOME t. terminal t»
    using terminal-unity_SSC someI-ex [of terminal] in-hom-char_FSbC terminal-def
    by auto
  obtain i' where i': «i' : (SOME t. terminal t) → S.unity»
    using terminal-unity_SSC someI-ex [of S.terminal] S.terminal-def
    by (metis (no-types, lifting) 1 terminal-def)
  have ii': inverse-arrows i i'
  proof
```

```
  have i' · i = S.unity
    using i i' terminal-unity_SSC
    by (metis (no-types, lifting) S.comp-in-homI' S.ide-in-hom S.ide-unity S.in-homE
        S.terminalE S.terminal-unity_SC in-hom-char_FSbC)
  thus 2: ide (comp i' i)
    by (metis (no-types, lifting) cod-comp comp-simp i i' ide-char' in-homE seqI')
  have i · i' = (SOME t. terminal t)
    using 1
    by (metis (no-types, lifting) 2 comp-simp i' ide-compE in-homE inverse-arrowsE
        iso-iff-mono-and-retraction point-is-mono retractionI section-retraction-of-iso(2))
  thus ide (comp i i')
    using comp-char
    by (metis (no-types, lifting) 2 ide-char' dom-comp i' ide-compE in-homE seq-char_SbC)
qed
interpret set-category-data comp ‹λx. S.some-img (x · i)› ..
have i-in-hom: «i : S.unity → unity»
  using i unity-def by simp
have i'-in-hom: «i' : unity → S.unity»
  using i' unity-def by simp
have ⋀a. ide a ⟹ set a = S.set a
proof −
  fix a
  assume a: ide a
  have set a = (λx. S.some-img (x · i)) ‘ hom unity a
    using set-def by simp
  also have ... = S.some-img ‘ S.hom S.unity a
  proof
    show (λx. S.some-img (x · i)) ‘ hom unity a ⊆ S.some-img ‘ S.hom S.unity a
      using i in-hom-char_FSbC i-in-hom by auto
    show S.some-img ‘ S.hom S.unity a ⊆ (λx. S.some-img (x · i)) ‘ hom unity a
    proof
      fix b
      assume b: b ∈ S.some-img ‘ S.hom S.unity a
      obtain x where x: x ∈ S.hom S.unity a ∧ S.some-img x = b
        using b by blast
      have x · i' ∈ hom unity a
        using x in-hom-char_FSbC S.comp-in-homI a i' ideD(1) unity-def by force
      moreover have S.some-img ((x · i') · i) = b
        by (metis (no-types, lifting) i ii' x S.comp-assoc calculation comp-simp
            ide-compE in-homE inverse-arrowsE mem-Collect-eq S.comp-arr-ide seqI'
            seq-char_SbC S.ide-unity unity-def)
      ultimately show b ∈ (λx. S.some-img (x · i)) ‘ hom unity a by blast
    qed
  qed
  also have ... = S.set a
    using S.set-def by auto
  finally show set a = S.set a by blast
qed
interpret T: set-category-given-img comp ‹λx. S.some-img (x · i)› ssetp
```

proof
show *Collect terminal* $\neq \{\}$
using $terminal\text{-}unity_{SSC}$ **by** *blast*
show $\bigwedge A'\ A.\ [\![A' \subseteq A;\ ssetp\ A]\!] \Longrightarrow ssetp\ A'$
using *subset-closed* **by** *blast*
show $\bigwedge A\ B.\ [\![ssetp\ A;\ ssetp\ B]\!] \Longrightarrow ssetp\ (A \cup B)$
using *union-closed* **by** *simp*
show $\bigwedge A.\ ssetp\ A \Longrightarrow A \subseteq Univ$
using *S.setp-imp-subset-Univ containment terminal-char* **by** *presburger*
show $\bigwedge a\ b.\ [\![ide\ a;\ ide\ b;\ set\ a = set\ b]\!] \Longrightarrow a = b$
using $ide\text{-}char_{SbC}$ ‹$\bigwedge a.\ ide\ a \Longrightarrow set\ a = S.set\ a$› *S.extensional-set* **by** *auto*
show $\bigwedge a.\ ide\ a \Longrightarrow ssetp\ (set\ a)$
using ‹$\bigwedge a.\ ide\ a \Longrightarrow set\ a = S.set\ a$› $ide\text{-}char_{SSC}$ **by** *force*
show $\bigwedge A.\ ssetp\ A \Longrightarrow \exists a.\ ide\ a \wedge set\ a = A$
using *S.set-complete* ‹$\bigwedge a.\ ide\ a \Longrightarrow set\ a = S.set\ a$› *containment* $ide\text{-}char_{SSC}$ **by** *blast*
show $\bigwedge t.\ terminal\ t \Longrightarrow t \in (\lambda x.\ S.some\text{-}img\ (x \cdot i))\ `\ hom\ unity\ t$
using *S.set-def S.stable-img* ‹$\bigwedge a.\ ide\ a \Longrightarrow set\ a = S.set\ a$› *set-def*
terminal-char terminal-def
by *force*
show $\bigwedge a.\ ide\ a \Longrightarrow inj\text{-}on\ (\lambda x.\ S.some\text{-}img\ (x \cdot i))\ (hom\ unity\ a)$
proof −
fix *a*
assume *a*: *ide a*
show *inj-on* $(\lambda x.\ S.some\text{-}img\ (x \cdot i))$ $(hom\ unity\ a)$
proof
fix *x y*
assume *x*: $x \in hom\ unity\ a$ **and** *y*: $y \in hom\ unity\ a$
and *eq*: $S.some\text{-}img\ (x \cdot i) = S.some\text{-}img\ (y \cdot i)$
have $x \cdot i = y \cdot i$
proof −
have $x \cdot i \in S.hom\ S.unity\ a \wedge y \cdot i \in S.hom\ S.unity\ a$
using $in\text{-}hom\text{-}char_{FSbC}$ ‹«$i : S.unity \to unity$»› *x y* **by** *blast*
thus *?thesis*
using *a eq* $ide\text{-}char_{SbC}$ *S.inj-img* [*of a*] *inj-on-def* [*of S.some-img*] **by** *simp*
qed
have $x = (x \cdot i) \cdot i'$
by (*metis* (*no-types*, *lifting*) *S.comp-arr-ide S.comp-assoc S.inverse-arrowsE*
S.match-4 i i' ii' inclusion-preserves-inverse mem-Collect-eq seqI'
$seq\text{-}char_{SbC}$ *unity-def x*)
also have ... $= (y \cdot i) \cdot i'$
using ‹$x \cdot i = y \cdot i$› **by** *simp*
also have ... $= y$
by (*metis* (*no-types*, *lifting*) *S.comp-arr-ide S.comp-assoc S.inverse-arrowsE*
S.match-4 i i' ii' inclusion-preserves-inverse mem-Collect-eq seqI'
$seq\text{-}char_{SbC}$ *unity-def y*)
finally show $x = y$ **by** *simp*
qed
qed
show $\bigwedge f\ f'.\ [\![par\ f\ f';\ \bigwedge x.\ in\text{-}hom\ x\ unity\ (dom\ f) \Longrightarrow comp\ f\ x = comp\ f'\ x]\!]$

$\Longrightarrow f = f'$
proof −
fix $f\ f'$
assume *par*: $par\ f\ f'$
assume *eq*: $\bigwedge x.\ in\text{-}hom\ x\ unity\ (dom\ f) \Longrightarrow comp\ f\ x = comp\ f'\ x$
have $S.par\ f\ f'$
using $par\ arr\text{-}char_{SbC}\ dom\text{-}char_{SbC}\ cod\text{-}char_{SbC}$ **by** *auto*
moreover have $\bigwedge x.\ S.in\text{-}hom\ x\ S.unity\ (S.dom\ f) \Longrightarrow f \cdot x = f' \cdot x$
proof −
fix x
assume x: $S.in\text{-}hom\ x\ S.unity\ (S.dom\ f)$
have $S.in\text{-}hom\ (x \cdot i')\ unity\ (S.dom\ f)$
using $i'\text{-}in\text{-}hom\ in\text{-}hom\text{-}char_{FSbC}\ x$ **by** *blast*
hence *1*: $in\text{-}hom\ (x \cdot i')\ unity\ (dom\ f)$
using $arr\text{-}dom\ dom\text{-}simp\ i\ in\text{-}hom\text{-}char_{FSbC}\ par\ unity\text{-}def$ **by** *force*
hence $comp\ f\ (x \cdot i') = comp\ f'\ (x \cdot i')$
using *eq* **by** *blast*
hence $(f \cdot (x \cdot i')) \cdot i = (f' \cdot (x \cdot i')) \cdot i$
using *comp-char*
by (*metis* (*no-types*, *lifting*) *1 comp-simp in-homE seqI par*)
thus $f \cdot x = f' \cdot x$
by (*metis* (*no-types*, *lifting*) *S.comp-arr-dom S.comp-assoc S.comp-inv-arr*
S.in-homE i-in-hom ii' in-hom-char$_{FSbC}$ *inclusion-preserves-inverse x*)
qed
ultimately show $f = f'$
using *S.extensional-arr* **by** *blast*
qed
show $\bigwedge a\ b\ F.\ [\![ide\ a;\ ide\ b;\ F \in hom\ unity\ a \to hom\ unity\ b]\!]$
$\Longrightarrow \exists f.\ in\text{-}hom\ f\ a\ b \wedge$
$(\forall x.\ in\text{-}hom\ x\ unity\ (dom\ f) \longrightarrow comp\ f\ x = F\ x)$
proof −
fix $a\ b\ F$
assume *a*: $ide\ a$ **and** *b*: $ide\ b$ **and** *F*: $F \in hom\ unity\ a \to hom\ unity\ b$
have $S.ide\ a$
using $a\ ide\text{-}char_{SbC}$ **by** *blast*
have $S.ide\ b$
using $b\ ide\text{-}char_{SbC}$ **by** *blast*
have *1*: $(\lambda x.\ F\ (x \cdot i') \cdot i) \in S.hom\ S.unity\ a \to S.hom\ S.unity\ b$
proof
fix x
assume x: $x \in S.hom\ S.unity\ a$
have $x \cdot i' \in S.hom\ unity\ a$
using $i'\text{-}in\text{-}hom\ in\text{-}hom\text{-}char_{FSbC}\ x$ **by** *blast*
hence $x \cdot i' \in hom\ unity\ a$
using $a\ in\text{-}hom\text{-}char_{FSbC}$
by (*metis* (*no-types*, *lifting*) *ideD*(*1*) $i'\text{-}in\text{-}hom\ in\text{-}hom\text{-}char_{FSbC}$ *mem-Collect-eq*)
hence $F\ (x \cdot i') \in hom\ unity\ b$
using $a\ b\ F$ **by** *blast*
hence $F\ (x \cdot i') \in S.hom\ unity\ b$

```
          using b in-hom-char_FSbC by blast
        thus F (x · i') · i ∈ S.hom S.unity b
          using i in-hom-char_FSbC unity-def by auto
      qed
      obtain f where f: S.in-hom f a b ∧ (∀x. S.in-hom x S.unity (S.dom f)
                        ⟶ f · x = (λx. F (x · i') · i) x)
        using 1 S.fun-complete-ax ‹S.ide a› ‹S.ide b› by presburger
      show ∃f. in-hom f a b ∧ (∀x. in-hom x unity (dom f) ⟶ comp f x = F x)
      proof −
        have in-hom f a b
          using f in-hom-char_FSbC ideD(1) a b by presburger
        moreover have ∀x. in-hom x unity (dom f) ⟶ comp f x = F x
        proof (intro allI impI)
          fix x
          assume x: in-hom x unity (dom f)
          have xi: S.in-hom (x · i) S.unity (S.dom f)
            using f x i in-hom-char_FSbC dom-char_SbC
            by (metis (no-types, lifting) in-homE unity-def calculation S.comp-in-homI)
          hence 1: f · (x · i) = F ((x · i) · i') · i
            using f by blast
          hence ((f · x) · i) · i' = (F x · i) · i'
            by (metis (no-types, lifting) xi S.comp-assoc S.inverse-arrowsE
                S.seqI' i' ii' in-hom-char_FSbC inclusion-preserves-inverse S.comp-arr-ide)
          hence f · x = F x
            by (metis (no-types, lifting) xi 1 S.invert-side-of-triangle(2) S.match-2 S.match-3
                S.seqI arr-char_SbC calculation S.in-homE S.inverse-unique S.isoI
                ii' in-homE inclusion-preserves-inverse)
          thus comp f x = F x
            using comp-char
            by (metis (no-types, lifting) comp-simp in-homE seqI calculation x)
        qed
        ultimately show ?thesis by blast
      qed
    qed
  qed
  show ∃img. set-category-given-img comp img ssetp
    using T.set-category-given-img-axioms by blast
qed

lemma is-set-category:
shows set-category comp ssetp
  ..

end

end
```

Chapter 10

SetCat

theory *SetCat*
imports *SetCategory ConcreteCategory*
begin

This theory proves the consistency of the *set-category* locale by giving a particular concrete construction of an interpretation for it. Applying the general construction given by *concrete-category*, we define arrows to be terms *MkArr A B F*, where A and B are sets and F is an extensional function that maps A to B.

This locale uses an extra dummy parameter just to fix the element type for sets. Without this, a type is used for each interpretation, which makes it impossible to construct set categories whose element types are related to the context. An additional parameter, *Setp*, allows some control over which subsets of the element type are assumed to correspond to objects of the category.

locale *setcat* =
fixes *elem-type* :: *'e itself*
and *Setp* :: *'e set* ⇒ *bool*
assumes *Setp-singleton*: *Setp* {*x*}
and *Setp-respects-subset*: *A'* ⊆ *A* ⟹ *Setp A* ⟹ *Setp A'*
and *union-preserves-Setp*: ⟦ *Setp A*; *Setp B* ⟧ ⟹ *Setp* (*A* ∪ *B*)
begin

lemma *finite-imp-Setp*: *finite A* ⟹ *Setp A*
using *Setp-singleton*
by (*metis finite-induct insert-is-Un Setp-respects-subset singleton-insert-inj-eq union-preserves-Setp*)

type-synonym *'b arr* = (*'b set*, *'b* ⇒ *'b*) *concrete-category.arr*

interpretation *S*: *concrete-category* ‹*Collect Setp*› ‹λ*A B*. *extensional A* ∩ (*A* → *B*)›
‹λ*A*. λ*x* ∈ *A*. *x*› ‹λ*C B A g f*. *compose A g f*›
using *compose-Id Id-compose*
apply *unfold-locales*
apply *auto*[3]

```
  apply blast
 by (metis IntD2 compose-assoc)

abbreviation comp :: 'e setcat.arr comp      (infixr ‹·› 55)
where comp ≡ S.COMP
notation S.in-hom                            (‹«- : - → -»›)

lemma is-category:
shows category comp
  using S.category-axioms by simp

lemma MkArr-expansion:
assumes S.arr f
shows f = S.MkArr (S.Dom f) (S.Cod f) (λx ∈ S.Dom f. S.Map f x)
proof (intro S.arr-eqI)
  show S.arr f by fact
  show S.arr (S.MkArr (S.Dom f) (S.Cod f) (λx ∈ S.Dom f. S.Map f x))
    using assms S.arr-char
    by (metis (mono-tags, lifting) Int-iff S.MkArr-Map extensional-restrict)
  show S.Dom f = S.Dom (S.MkArr (S.Dom f) (S.Cod f) (λx ∈ S.Dom f. S.Map f x))
    by simp
  show S.Cod f = S.Cod (S.MkArr (S.Dom f) (S.Cod f) (λx ∈ S.Dom f. S.Map f x))
    by simp
  show S.Map f = S.Map (S.MkArr (S.Dom f) (S.Cod f) (λx ∈ S.Dom f. S.Map f x))
    using assms S.arr-char
    by (metis (mono-tags, lifting) Int-iff S.MkArr-Map extensional-restrict)
qed

lemma arr-char:
shows S.arr f ⟷ f ≠ S.Null ∧ Setp (S.Dom f) ∧ Setp (S.Cod f) ∧
                S.Map f ∈ extensional (S.Dom f) ∩ (S.Dom f → S.Cod f)
  using S.arr-char by auto

lemma terminal-char:
shows S.terminal a ⟷ (∃x. a = S.MkIde {x})
proof
  show ∃x. a = S.MkIde {x} ⟹ S.terminal a
  proof −
    assume a: ∃x. a = S.MkIde {x}
    from this obtain x where x: a = S.MkIde {x} by blast
    have S.terminal (S.MkIde {x})
    proof
      show 1: S.ide (S.MkIde {x})
        using finite-imp-Setp S.ide-MkIde by auto
      show ⋀a. S.ide a ⟹ ∃!f. «f : a → S.MkIde {x}»
      proof
        fix a :: 'e setcat.arr
        assume a: S.ide a
        show «S.MkArr (S.Dom a) {x} (λ-∈S.Dom a. x) : a → S.MkIde {x}»
```

```
      using a 1 S.arr-MkArr S.dom-MkArr S.cod-MkArr S.ide-char_CC S.MkArr-in-hom
      by (intro S.MkArr-in-hom) auto
    fix f :: 'e setcat.arr
    assume f: «f : a → S.MkIde {x}»
    show f = S.MkArr (S.Dom a) {x} (λ- ∈ S.Dom a. x)
    proof −
      have 1: S.Dom f = S.Dom a ∧ S.Cod f = {x}
        using a f by (metis (mono-tags, lifting) S.Dom.simps(1) S.in-hom-char)
      moreover have S.Map f = (λ- ∈ S.Dom a. x)
      proof
        fix z
        have z ∉ S.Dom a ⟹ S.Map f z = (λ- ∈ S.Dom a. x) z
          using f 1 MkArr-expansion
          by (metis (mono-tags, lifting) S.Map.simps(1) S.in-homE restrict-apply)
        moreover have z ∈ S.Dom a ⟹ S.Map f z = (λ- ∈ S.Dom a. x) z
          using f 1 arr-char [of f] by fastforce
        ultimately show S.Map f z = (λ- ∈ S.Dom a. x) z by auto
      qed
      ultimately show ?thesis
       using f MkArr-expansion [of f] by fastforce
    qed
  qed
  thus S.terminal a using x by simp
qed
show S.terminal a ⟹ ∃ x. a = S.MkIde {x}
proof −
  assume a: S.terminal a
  hence ide-a: S.ide a using S.terminal-def by auto
  have 1: ∃!x. x ∈ S.Dom a
  proof −
    have S.Dom a = {} ⟹ ¬S.terminal a
    proof −
      assume S.Dom a = {}
      hence 1: a = S.MkIde {}
        using S.MkIde-Dom' ‹S.ide a› by fastforce
      have ⋀f. f ∈ S.hom (S.MkIde {undefined}) (S.MkIde ({} :: 'e set))
                ⟹ S.Map f ∈ {undefined} → {}
      proof −
        fix f
        assume f: f ∈ S.hom (S.MkIde {undefined}) (S.MkIde ({} :: 'e set))
        show S.Map f ∈ {undefined} → {}
          using f MkArr-expansion arr-char [of f] S.in-hom-char by auto
      qed
      hence S.hom (S.MkIde {undefined}) a = {} using 1 by auto
      moreover have S.ide (S.MkIde {undefined})
        using finite-imp-Setp
        by (metis (mono-tags, lifting) finite.intros(1−2) S.ide-MkIde mem-Collect-eq)
      ultimately show ¬S.terminal a by blast
```

```
      qed
      moreover have ⋀x x'. x ∈ S.Dom a ∧ x' ∈ S.Dom a ∧ x ≠ x' ⟹ ¬S.terminal a
      proof −
        fix x x'
        assume 1: x ∈ S.Dom a ∧ x' ∈ S.Dom a ∧ x ≠ x'
        let ?f = S.MkArr {undefined} (S.Dom a) (λ- ∈ {undefined}. x)
        let ?f' = S.MkArr {undefined} (S.Dom a) (λ- ∈ {undefined}. x')
        have S.par ?f ?f'
          using 1 S.Dom-in-Obj S.arr-MkArr S.cod-MkArr S.dom-MkArr S.ideD(1)
                Setp-singleton ide-a
          by force
        moreover have ?f ≠ ?f'
          using 1 by (metis S.arr.inject restrict-apply' singletonI)
        ultimately show ¬S.terminal a
          using S.cod-MkArr ide-a S.terminal-arr-unique [of ?f ?f'] by auto
      qed
      ultimately show ?thesis
        using a by auto
    qed
    hence S.Dom a = {THE x. x ∈ S.Dom a}
      using theI [of λx. x ∈ S.Dom a] by auto
    hence a = S.MkIde {THE x. x ∈ S.Dom a}
      using a S.terminal-def by (metis (mono-tags, lifting) S.MkIde-Dom')
    thus ∃x. a = S.MkIde {x}
      by auto
  qed
qed

definition IMG :: 'e setcat.arr ⇒ 'e setcat.arr
where IMG f = S.MkIde (S.Map f ‘ S.Dom f)

interpretation S: set-category-data comp IMG
  ..

lemma terminal-unity:
shows S.terminal S.unity
  using terminal-char S.unity-def someI-ex [of S.terminal]
  by (metis (mono-tags, lifting))
```

The inverse maps *arr-of* and *elem-of* are used to pass back and forth between the inhabitants of type $'a$ and the corresponding terminal objects. These are exported so that a client of the theory can relate the concrete element type $'a$ to the otherwise abstract arrow type.

```
definition arr-of :: 'e ⇒ 'e setcat.arr
where arr-of x ≡ S.MkIde {x}

definition elem-of :: 'e setcat.arr ⇒ 'e
where elem-of t ≡ the-elem (S.Dom t)
```

abbreviation U
where $U \equiv$ *elem-of S.unity*

lemma *arr-of-mapsto*:
shows *arr-of* $\in$ *UNIV* $\to$ *S.Univ*
  using *terminal-char arr-of-def* **by** *fast*

lemma *elem-of-mapsto*:
shows *elem-of* $\in$ *Univ* $\to$ *UNIV*
  by *auto*

lemma *elem-of-arr-of* [*simp*]:
shows *elem-of* (*arr-of x*) = *x*
  by (*simp add*: *elem-of-def arr-of-def*)

lemma *arr-of-elem-of* [*simp*]:
assumes *t* $\in$ *S.Univ*
shows *arr-of* (*elem-of t*) = *t*
  using *assms terminal-char arr-of-def elem-of-def*
  by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq elem-of-arr-of*)

lemma *inj-arr-of*:
shows *inj arr-of*
  by (*metis elem-of-arr-of injI*)

lemma *bij-arr-of*:
shows *bij-betw arr-of UNIV S.Univ*
proof (*intro bij-betwI*)
  show $\bigwedge x$ :: *'e*. *elem-of* (*arr-of x*) = *x* **by** *simp*
  show $\bigwedge t$. *t* $\in$ *S.Univ* $\Longrightarrow$ *arr-of* (*elem-of t*) = *t* **by** *simp*
  show *arr-of* $\in$ *UNIV* $\to$ *S.Univ* **using** *arr-of-mapsto* **by** *auto*
  show *elem-of* $\in$ *Collect S.terminal* $\to$ *UNIV* **by** *auto*
qed

lemma *bij-elem-of*:
shows *bij-betw elem-of S.Univ UNIV*
  using *elem-of-mapsto arr-of-mapsto*
  by (*intro bij-betwI*) *auto*

lemma *elem-of-img-arr-of-img* [*simp*]:
shows *elem-of* ‘ *arr-of* ‘ *A* = *A*
  by *force*

lemma *arr-of-img-elem-of-img* [*simp*]:
assumes *A* $\subseteq$ *S.Univ*
shows *arr-of* ‘ *elem-of* ‘ *A* = *A*
  using *assms* **by** *force*

lemma *Dom-terminal*:

assumes *S.terminal t*
shows *S.Dom t* = {*elem-of t*}
using *assms arr-of-def*
by (*metis* (*mono-tags*, *lifting*) *S.Dom.simps*(*1*) *elem-of-def terminal-char the-elem-eq*)

The image of a point $p \in$ *hom unity a* is a terminal object, which is given by the formula (*arr-of* ∘ *Fun p* ∘ *elem-of*) *unity*.

lemma *IMG-point*:
assumes «*p* : *S.unity* → *a*»
shows *IMG* ∈ *S.hom S.unity a* → *S.Univ*
and *IMG p* = (*arr-of o S.Map p o elem-of*) *S.unity*
proof −
show *IMG* ∈ *S.hom S.unity a* → *S.Univ*
proof
fix *f*
assume *f*: *f* ∈ *S.hom S.unity a*
have *S.terminal* (*S.MkIde* (*S.Map f* ‘ *S.Dom S.unity*))
using *terminal-char terminal-unity* **by** *force*
hence *S.MkIde* (*S.Map f* ‘ *S.Dom S.unity*) ∈ *S.Univ* **by** *simp*
moreover have *S.MkIde* (*S.Map f* ‘ *S.Dom S.unity*) = *IMG f*
using *f IMG-def S.in-hom-char*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
ultimately show *IMG f* ∈ *S.Univ* **by** *auto*
qed
have *IMG p* = *S.MkIde* (*S.Map p* ‘ *S.Dom p*) **using** *IMG-def* **by** *blast*
also have ... = *S.MkIde* (*S.Map p* ‘ {*U*})
using *assms S.in-hom-char terminal-unity Dom-terminal*
by (*metis* (*mono-tags*, *lifting*))
also have ... = (*arr-of o S.Map p o elem-of*) *S.unity* **by** (*simp add*: *arr-of-def*)
finally show *IMG p* = (*arr-of o S.Map p o elem-of*) *S.unity* **using** *assms* **by** *auto*
qed

The function *IMG* is injective on *hom unity a* and its inverse takes a terminal object t to the arrow in *hom unity a* corresponding to the constant-t function.

abbreviation *MkElem* :: *'e setcat.arr* => *'e setcat.arr* => *'e setcat.arr*
where *MkElem t a* ≡ *S.MkArr* {*U*} (*S.Dom a*) (λ- ∈ {*U*}. *elem-of t*)

lemma *MkElem-in-hom*:
assumes *S.arr f* **and** *x* ∈ *S.Dom f*
shows «*MkElem* (*arr-of x*) (*S.dom f*) : *S.unity* → *S.dom f*»
proof −
have (λ- ∈ {*U*}. *elem-of* (*arr-of x*)) ∈ {*U*} → *S.Dom* (*S.dom f*)
using *assms S.dom-char* [*of f*] **by** *simp*
moreover have *S.MkIde* {*U*} = *S.unity*
using *terminal-char terminal-unity*
by (*metis* (*mono-tags*, *lifting*) *elem-of-arr-of arr-of-def*)
moreover have *S.MkIde* (*S.Dom* (*S.dom f*)) = *S.dom f*
using *assms S.dom-char S.MkIde-Dom' S.ide-dom* **by** *blast*
ultimately show *?thesis*

```
    using assms S.MkArr-in-hom [of {U} S.Dom (S.dom f) λ- ∈ {U}. elem-of (arr-of x)]
    by (metis (no-types, lifting) S.Dom.simps(1) S.Dom-in-Obj IntI S.arr-dom S.ideD(1)
        restrict-extensional S.terminal-def terminal-unity)
qed

lemma MkElem-IMG:
assumes p ∈ S.hom S.unity a
shows MkElem (IMG p) a = p
proof −
  have 0: IMG p = arr-of (S.Map p U)
    using assms IMG-point(2) by auto
  have 1: S.Dom p = {U}
    using assms terminal-unity Dom-terminal
    by (metis (mono-tags, lifting) S.in-hom-char mem-Collect-eq)
  moreover have S.Cod p = S.Dom a
    using assms
    by (metis (mono-tags, lifting) S.in-hom-char mem-Collect-eq)
  moreover have S.Map p = (λ- ∈ {U}. elem-of (IMG p))
  proof
    fix e
    show S.Map p e = (λ- ∈ {U}. elem-of (IMG p)) e
    proof −
      have S.Map p e = (λx ∈ S.Dom p. S.Map p x) e
        using assms MkArr-expansion [of p]
        by (metis (mono-tags, lifting) CollectD S.Map.simps(1) S.in-homE)
      also have ... = (λ- ∈ {U}. elem-of (IMG p)) e
        using assms 0 1 by simp
      finally show ?thesis by blast
    qed
  qed
  ultimately show MkElem (IMG p) a = p
    using assms S.MkArr-Map CollectD
    by (metis (mono-tags, lifting) S.in-homE mem-Collect-eq)
qed

lemma inj-IMG:
assumes S.ide a
shows inj-on IMG (S.hom S.unity a)
proof (intro inj-onI)
  fix x y
  assume x: x ∈ S.hom S.unity a
  assume y: y ∈ S.hom S.unity a
  assume eq: IMG x = IMG y
  show x = y
  proof (intro S.arr-eqI)
    show S.arr x using x by blast
    show S.arr y using y by blast
    show S.Dom x = S.Dom y
      using x y S.in-hom-char by (metis (mono-tags, lifting) CollectD)
```

```
      show S.Cod x = S.Cod y
        using x y S.in-hom-char by (metis (mono-tags, lifting) CollectD)
      show S.Map x = S.Map y
      proof -
        have ⋀a. y ∈ S.hom S.unity a ⟹ S.MkArr {U} (S.Dom a) (λe∈{U}. elem-of (IMG
x)) = y
          using MkElem-IMG eq by presburger
        hence y = x
          using MkElem-IMG x y by blast
        thus ?thesis by meson
      qed
    qed
  qed

  lemma set-char:
  assumes S.ide a
  shows S.set a = arr-of ‘ S.Dom a
  proof
    show S.set a ⊆ arr-of ‘ S.Dom a
    proof
      fix t
      assume t ∈ S.set a
      from this obtain p where p: «p : S.unity → a» ∧ t = IMG p
        using S.set-def by blast
      have t = (arr-of o S.Map p o elem-of) S.unity
        using p IMG-point(2) by blast
      moreover have (S.Map p o elem-of) S.unity ∈ S.Dom a
        using p arr-char S.in-hom-char Dom-terminal terminal-unity
        by (metis (mono-tags, lifting) IntD2 Pi-split-insert-domain o-apply)
      ultimately show t ∈ arr-of ‘ S.Dom a by simp
    qed
    show arr-of ‘ S.Dom a ⊆ S.set a
    proof
      fix t
      assume t ∈ arr-of ‘ S.Dom a
      from this obtain x where x: x ∈ S.Dom a ∧ t = arr-of x by blast
      let ?p = MkElem (arr-of x) a
      have p: ?p ∈ S.hom S.unity a
        using assms x MkElem-in-hom [of S.dom a] S.ideD(1−2) by force
      moreover have IMG ?p = t
        using p x elem-of-arr-of IMG-def arr-of-def
        by (metis (no-types, lifting) S.Dom.simps(1) S.Map.simps(1) image-empty
            image-insert image-restrict-eq)
      ultimately show t ∈ S.set a using S.set-def by blast
    qed
  qed

  lemma Map-via-comp:
  assumes S.arr f
```

shows $S.Map\ f = (\lambda x \in S.Dom\ f.\ S.Map\ (f \cdot MkElem\ (arr\text{-}of\ x)\ (S.dom\ f))\ U)$
proof
fix x
have $x \notin S.Dom\ f \Longrightarrow S.Map\ f\ x = (\lambda x \in S.Dom\ f.\ S.Map\ (f \cdot MkElem\ (arr\text{-}of\ x)\ (S.dom\ f))\ U)\ x$
using *assms arr-char* [*of f*] *IntD1 extensional-arb restrict-apply* **by** *fastforce*
moreover have
$x \in S.Dom\ f \Longrightarrow S.Map\ f\ x = (\lambda x \in S.Dom\ f.\ S.Map\ (f \cdot MkElem\ (arr\text{-}of\ x)\ (S.dom\ f))\ U)\ x$
proof −
assume *x*: $x \in S.Dom\ f$
let *?X* = $MkElem\ (arr\text{-}of\ x)\ (S.dom\ f)$
have «*?X* : $S.unity \to S.dom\ f$»
using *assms x MkElem-in-hom* **by** *auto*
moreover have $S.Dom\ ?X = \{U\} \wedge S.Map\ ?X = (\lambda\text{-} \in \{U\}.\ x)$
using *x* **by** *simp*
ultimately have
$S.Map\ (f \cdot MkElem\ (arr\text{-}of\ x)\ (S.dom\ f)) = compose\ \{U\}\ (S.Map\ f)\ (\lambda\text{-} \in \{U\}.\ x)$
using *assms x S.Map-comp* [*of MkElem* (*arr-of x*) (*S.dom f*) *f*]
by (*metis* (*mono-tags*, *lifting*) *S.Cod.simps*(*1*) *S.Dom-dom S.arr-iff-in-hom S.seqE S.seqI'*)
thus *?thesis*
using *x* **by** (*simp add*: *compose-eq restrict-apply' singletonI*)
qed
ultimately show $S.Map\ f\ x = (\lambda x \in S.Dom\ f.\ S.Map\ (f \cdot MkElem\ (arr\text{-}of\ x)\ (S.dom\ f))\ U)\ x$
by *auto*
qed

lemma *arr-eqI'*:
assumes $S.par\ f\ f'$ **and** $\bigwedge t.$ «$t : S.unity \to S.dom\ f$» $\Longrightarrow f \cdot t = f' \cdot t$
shows $f = f'$
proof (*intro S.arr-eqI*)
show $S.arr\ f$ **using** *assms* **by** *simp*
show $S.arr\ f'$ **using** *assms* **by** *simp*
show $S.Dom\ f = S.Dom\ f'$
using *assms* **by** (*metis* (*mono-tags*, *lifting*) *S.Dom-dom*)
show $S.Cod\ f = S.Cod\ f'$
using *assms* **by** (*metis* (*mono-tags*, *lifting*) *S.Cod-cod*)
show $S.Map\ f = S.Map\ f'$
proof
have *1*: $\bigwedge x.\ x \in S.Dom\ f \Longrightarrow$ «$MkElem\ (arr\text{-}of\ x)\ (S.dom\ f) : S.unity \to S.dom\ f$»
using *MkElem-in-hom* **by** (*metis* (*mono-tags*, *lifting*) *assms*(*1*))
fix *x*
show $S.Map\ f\ x = S.Map\ f'\ x$
using *assms 1* ‹$S.Dom\ f = S.Dom\ f'$› **by** (*simp add*: *Map-via-comp*)
qed
qed

lemma *Setp-elem-of-img*:
assumes $A \in S.set$ ‘ $Collect\ S.ide$
shows $Setp\ (elem\text{-}of$ ‘ $A)$
proof −
  obtain a **where** a: $S.ide\ a \wedge S.set\ a = A$
    using *assms* **by** *blast*
  have $elem\text{-}of$ ‘ $S.set\ a = S.Dom\ a$
  proof −
    have $S.set\ a = arr\text{-}of$ ‘ $S.Dom\ a$
    using a *set-char* **by** *blast*
    moreover have $elem\text{-}of$ ‘ $arr\text{-}of$ ‘ $S.Dom\ a = S.Dom\ a$
      using *elem-of-arr-of* **by** *force*
    ultimately show *?thesis* **by** *presburger*
  qed
  thus *?thesis*
    using a $S.ide\text{-}char_{CC}$ **by** *auto*
qed

lemma *set-MkIde-elem-of-img*:
assumes $A \subseteq S.Univ$ **and** $S.ide\ (S.MkIde\ (elem\text{-}of$ ‘ $A))$
shows $S.set\ (S.MkIde\ (elem\text{-}of$ ‘ $A)) = A$
proof −
  have $S.Dom\ (S.MkIde\ (elem\text{-}of$ ‘ $A)) = elem\text{-}of$ ‘ A
    by *simp*
  moreover have $arr\text{-}of$ ‘ $elem\text{-}of$ ‘ $A = A$
    using *assms arr-of-elem-of* **by** *force*
  ultimately show *?thesis*
    using *assms Setp-elem-of-img set-char S.ide-MkIde* **by** *auto*
qed

lemma *set-img-Collect-ide-iff*:
shows $A \in S.set$ ‘ $Collect\ S.ide \longleftrightarrow A \subseteq S.Univ \wedge Setp\ (elem\text{-}of$ ‘ $A)$
proof
  show $A \in S.set$ ‘ $Collect\ S.ide \Longrightarrow A \subseteq S.Univ \wedge Setp\ (elem\text{-}of$ ‘ $A)$
    using *set-char arr-of-mapsto Setp-elem-of-img* **by** *auto*
  assume A: $A \subseteq S.Univ \wedge Setp\ (elem\text{-}of$ ‘ $A)$
  have $S.ide\ (S.MkIde\ (elem\text{-}of$ ‘ $A))$
    using A *S.ide-MkIde* **by** *blast*
  moreover have $S.set\ (S.MkIde\ (elem\text{-}of$ ‘ $A)) = A$
    using A *calculation set-MkIde-elem-of-img* **by** *presburger*
  ultimately show $A \in S.set$ ‘ $Collect\ S.ide$ **by** *blast*
qed

The main result, which establishes the consistency of the *set-category* locale and provides us with a way of obtaining “set categories” at arbitrary types.

theorem *is-set-category*:
shows $set\text{-}category\ comp\ (\lambda A.\ A \subseteq S.Univ \wedge Setp\ (elem\text{-}of$ ‘ $A))$
proof

```
show ∃ img :: 'e setcat.arr ⇒ 'e setcat.arr.
                set-category-given-img comp img (λA. A ⊆ S.Univ ∧ Setp (elem-of ‘ A))
proof
  show set-category-given-img comp IMG (λA. A ⊆ S.Univ ∧ Setp (elem-of ‘ A))
  proof
    show S.Univ ≠ {} using terminal-char by blast
    fix a :: 'e setcat.arr
    assume a: S.ide a
    show S.set a ⊆ S.Univ ∧ Setp (elem-of ‘ S.set a)
      using a set-img-Collect-ide-iff by auto
    show inj-on IMG (S.hom S.unity a) using a inj-IMG terminal-unity by blast
    next
    fix t :: 'e setcat.arr
    assume t: S.terminal t
    show t ∈ IMG ‘ S.hom S.unity t
    proof −
      have t ∈ S.set t
        using t set-char [of t]
        by (metis (mono-tags, lifting) S.Dom.simps(1) image-insert insertI1 arr-of-def
            terminal-char S.terminal-def)
      thus ?thesis
        using t S.set-def [of t] by simp
    qed
    next
    show ⋀A. A ⊆ S.Univ ∧ Setp (elem-of ‘ A) ⟹ ∃ a. S.ide a ∧ S.set a = A
      using set-img-Collect-ide-iff by fast
    next
    fix a b :: 'e setcat.arr
    assume a: S.ide a and b: S.ide b and ab: S.set a = S.set b
    show a = b
      using a b ab set-char inj-arr-of inj-image-eq-iff S.dom-char S.in-homE S.ide-in-hom
      by (metis (mono-tags, lifting))
    next
    fix f f' :: 'e setcat.arr
    assume par: S.par f f' and ff': ⋀x. «x : S.unity → S.dom f» ⟹ f · x = f' · x
    show f = f' using par ff' arr-eqI' by blast
    next
    fix a b :: 'e setcat.arr and F :: 'e setcat.arr ⇒ 'e setcat.arr
    assume a: S.ide a and b: S.ide b and F: F ∈ S.hom S.unity a → S.hom S.unity b
    show ∃ f. «f : a → b» ∧ (∀ x. «x : S.unity → S.dom f» ⟶ f · x = F x)
    proof
      let ?f = S.MkArr (S.Dom a) (S.Dom b) (λx ∈ S.Dom a. S.Map (F (MkElem (arr-of
x) a)) U)
      have 1: «?f : a → b»
      proof −
        have (λx ∈ S.Dom a. S.Map (F (MkElem (arr-of x) a)) U)
              ∈ extensional (S.Dom a) ∩ (S.Dom a → S.Dom b)
        proof
         show (λx ∈ S.Dom a. S.Map (F (MkElem (arr-of x) a)) U) ∈ extensional (S.Dom
```

```
a)
        using a F by simp
      show (λx ∈ S.Dom a. S.Map (F (MkElem (arr-of x) a)) U) ∈ S.Dom a → S.Dom
b
       proof
        fix x
        assume x: x ∈ S.Dom a
        show S.Map (F (MkElem (arr-of x) a)) U ∈ S.Dom b
        proof −
          have 1: F (MkElem (arr-of x) a) ∈ S.hom S.unity b
            using x a F MkElem-in-hom S.ide-char S.ideD(1−2)
            by force
          moreover have S.Dom (F (MkElem (arr-of x) a)) = {U}
            using 1 MkElem-IMG
            by (metis (mono-tags, lifting) S.Dom.simps(1))
          moreover have S.Cod (F (MkElem (arr-of x) a)) = S.Dom b
            using 1 by (metis (mono-tags, lifting) CollectD S.in-hom-char)
          ultimately have S.Map (F (MkElem (arr-of x) a)) ∈ {U} → S.Dom b
            using arr-char by blast
          thus ?thesis by blast
        qed
       qed
      qed
      hence «?f : S.MkIde (S.Dom a) → S.MkIde (S.Dom b)»
        using a b S.MkArr-in-hom S.ide-char_CC by blast
      thus ?thesis
        using a b by simp
    qed
    moreover have ⋀x. «x : S.unity → S.dom ?f» ⟹ ?f · x = F x
    proof −
      fix x
      assume x: «x : S.unity → S.dom ?f»
      have 2: x = MkElem (IMG x) a
        using a x 1 MkElem-IMG [of x a]
        by (metis (mono-tags, lifting) S.in-homE mem-Collect-eq)
      moreover have 5: S.Dom x = {U} ∧ S.Cod x = S.Dom a ∧
                       S.Map x = (λ- ∈ {U}. elem-of (IMG x))
        using x 2
        by (metis (no-types, lifting) S.Cod.simps(1) S.Dom.simps(1) S.Map.simps(1))
      moreover have S.Cod ?f = S.Dom b using 1 by simp
      ultimately have
          3: ?f · x =
           S.MkArr {U} (S.Dom b) (compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG
x)))
        by (metis (no-types, lifting) 1 S.Map.simps(1) S.comp-MkArr S.in-homE x)
      have 4: compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) = S.Map (F x)
      proof
        fix y
        have y ∉ {U} ⟹
```

```
        compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y = S.Map (F x) y
  proof −
    assume y: y ∉ {U}
    have compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y = undefined
      using y compose-def by simp
    also have ... = S.Map (F x) y
    proof −
      have 6: F x ∈ S.hom S.unity b using x F 1 by fastforce
      hence S.Dom (F x) = {U}
        by (metis (mono-tags, lifting) x 2 CollectD S.Dom.simps(1) S.in-hom-char)
      thus ?thesis
        using x y F 6 arr-char extensional-arb [of S.Map (F x) {U} y]
        by (metis (mono-tags, lifting) CollectD Int-iff S.in-hom-char)
    qed
    ultimately show ?thesis by auto
  qed
  moreover have
      y ∈ {U} ⟹
        compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y = S.Map (F x) y
  proof −
    assume y: y ∈ {U}
    have compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y =
          S.Map ?f (elem-of (IMG x))
      using y by (simp add: compose-eq restrict-apply')
    also have ... = (λx. S.Map (F (MkElem (arr-of x) a)) U) (elem-of (IMG x))
    proof −
      have elem-of (IMG x) ∈ S.Dom a
        using x y a 5 arr-char S.in-homE restrict-apply by force
      thus ?thesis
        using restrict-apply by simp
    qed
    also have ... = S.Map (F x) y
      using x y 1 2 MkElem-IMG by simp
    finally show
        compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y = S.Map (F x) y
      by auto
  qed
  ultimately show
      compose {U} (S.Map ?f) (λ- ∈ {U}. elem-of (IMG x)) y = S.Map (F x) y
    by auto
qed
show ?f · x = F x
proof (intro S.arr-eqI)
  have 5: ?f · x ∈ S.hom S.unity b using 1 x by blast
  have 6: F x ∈ S.hom S.unity b
    using x F 1
    by (metis (mono-tags, lifting) PiE S.in-homE mem-Collect-eq)
  show S.arr (comp ?f x) using 5 by blast
  show S.arr (F x) using 6 by blast
```

```
        show S.Dom (comp ?f x) = S.Dom (F x)
          using 5 6 by (metis (mono-tags, lifting) CollectD S.in-hom-char)
        show S.Cod (comp ?f x) = S.Cod (F x)
          using 5 6 by (metis (mono-tags, lifting) CollectD S.in-hom-char)
        show S.Map (comp ?f x) = S.Map (F x)
          using 3 4 by simp
      qed
    qed
    thus «?f : a → b» ∧ (∀x. «x : S.unity → S.dom ?f» ⟶ comp ?f x = F x)
      using 1 by blast
  qed
  next
  show ⋀A. A ⊆ S.Univ ∧ Setp (elem-of ‘ A) ⟹ A ⊆ S.Univ
    by simp
  show ⋀A' A. ⟦A' ⊆ A; A ⊆ S.Univ ∧ Setp (elem-of ‘ A)⟧
                ⟹ A' ⊆ S.Univ ∧ Setp (elem-of ‘ A')
    by (meson image-mono setcat.Setp-respects-subset setcat-axioms subset-trans)
  show ⋀A B. ⟦A ⊆ S.Univ ∧ Setp (elem-of ‘ A); B ⊆ S.Univ ∧ Setp (elem-of ‘ B)⟧
              ⟹ A ∪ B ⊆ S.Univ ∧ Setp (elem-of ‘ (A ∪ B))
    by (simp add: image-Un union-preserves-Setp)
 qed
qed
qed
```

SetCat can be viewed as a concrete set category over its own element type $'a$, using *arr-of* as the required injection from $'a$ to the universe of *SetCat*.

```
corollary is-concrete-set-category:
shows concrete-set-category comp (λA. A ⊆ S.Univ ∧ Setp (elem-of ‘ A)) UNIV arr-of
proof −
  interpret S': set-category comp ‹λA. A ⊆ S.Univ ∧ Setp (elem-of ‘ A)›
    using is-set-category by auto
  show ?thesis
  proof
    show 1: arr-of ∈ UNIV → S'.Univ
      using arr-of-def terminal-char by force
    show inj-on arr-of UNIV
      using inj-arr-of by blast
  qed
qed
```

As a consequence of the categoricity of the *set-category* axioms, if S interprets *set-category*, and if φ is a bijection between the universe of S and the elements of type $'a$, then S is isomorphic to the category *setcat* of $'a$ sets and functions between them constructed here.

```
corollary set-category-iso-SetCat:
fixes S :: 's comp and φ :: 's ⇒ 'e
assumes set-category S 𝒮
and bij-betw φ (set-category-data.Univ S) UNIV
and ⋀A. 𝒮 A ⟷ A ⊆ set-category-data.Univ S ∧ (arr-of ∘ φ) ‘ A ∈ S.set ‘ Collect S.ide
```

```
  shows ∃Φ. invertible-functor S comp Φ
              ∧ (∀m. set-category.incl S 𝒮 m
                      ⟶ set-category.incl comp (λA. A ∈ S.set ‘ Collect S.ide) (Φ m))
  proof −
    interpret S′: set-category S 𝒮 using assms by auto
    let ?ψ = inv-into S′.Univ φ
    have bij-betw (arr-of o φ) S′.Univ S.Univ
    proof (intro bij-betwI)
      show arr-of o φ ∈ S′.Univ → Collect S.terminal
        using assms(2) arr-of-mapsto by auto
      show ?ψ o elem-of ∈ S.Univ → S′.Univ
      proof
        fix x :: 'e setcat.arr
        assume x: x ∈ S.Univ
        show (inv-into S′.Univ φ ∘ elem-of) x ∈ S′.Univ
          using x assms(2) bij-betw-def comp-apply inv-into-into
          by (metis UNIV-I)
      qed
      fix t
      assume t ∈ S′.Univ
      thus (?ψ o elem-of) ((arr-of o φ) t) = t
        using assms(2) bij-betw-inv-into-left
        by (metis comp-apply elem-of-arr-of)
      next
      fix t′ :: 'e setcat.arr
      assume t′ ∈ S.Univ
      thus (arr-of o φ) ((?ψ o elem-of) t′) = t′
        using assms(2) by (simp add: bij-betw-def f-inv-into-f)
    qed
    thus ?thesis
      using assms is-set-category set-img-Collect-ide-iff
            set-category-is-categorical
              [of S 𝒮 comp λA. A ∈ S.set ‘ Collect S.ide arr-of o φ]
      by simp
  qed

  sublocale category comp
    using is-category by blast
  sublocale set-category comp ‹λA. A ⊆ Collect S.terminal ∧ Setp (elem-of ‘ A)›
    using is-set-category set-img-Collect-ide-iff by simp
  interpretation concrete-set-category comp ‹λA. A ⊆ Collect S.terminal ∧ Setp (elem-of ‘
A)›
                 UNIV arr-of
    using is-concrete-set-category by simp

end
```

Here we discard the temporary interpretations S, leaving only the exported definitions and facts.

context *setcat*
begin

We establish mappings to pass back and forth between objects and arrows of the category and sets and functions on the underlying elements.

interpretation *set-category comp* ‹λ*A*. *A* ⊆ *Collect terminal* ∧ *Setp* (*elem-of* ‘ *A*)›
using *is-set-category* **by** *blast*
interpretation *concrete-set-category comp* ‹λ*A*. *A* ⊆ *Univ* ∧ *Setp* (*elem-of* ‘ *A*)› *UNIV arr-of*
using *is-concrete-set-category* **by** *blast*

definition *set-of-ide* :: *'e setcat.arr* ⇒ *'e set*
where *set-of-ide a* ≡ *elem-of* ‘ *set a*

definition *ide-of-set* :: *'e set* ⇒ *'e setcat.arr*
where *ide-of-set A* ≡ *mkIde* (*arr-of* ‘ *A*)

lemma *bij-betw-ide-set*:
shows *set-of-ide* ∈ *Collect ide* → *Collect Setp*
and *ide-of-set* ∈ *Collect Setp* → *Collect ide*
and [*simp*]: *ide a* ⟹ *ide-of-set* (*set-of-ide a*) = *a*
and [*simp*]: *Setp A* ⟹ *set-of-ide* (*ide-of-set A*) = *A*
and *bij-betw set-of-ide* (*Collect ide*) (*Collect Setp*)
and *bij-betw ide-of-set* (*Collect Setp*) (*Collect ide*)
proof −
show *1*: *set-of-ide* ∈ *Collect ide* → *Collect Setp*
using *setp-set-ide set-of-ide-def* **by** *auto*
show *2*: *ide-of-set* ∈ *Collect Setp* → *Collect ide*
proof
fix *A* :: *'e set*
assume *A*: *A* ∈ *Collect Setp*
have *arr-of* ‘ *A* ⊆ *Univ*
using *A arr-of-mapsto* **by** *auto*
moreover have *Setp* (*elem-of* ‘ *arr-of* ‘ *A*)
using *A elem-of-arr-of Setp-respects-subset* **by** *simp*
ultimately have *ide* (*mkIde* (*arr-of* ‘ *A*))
using *ide-mkIde* **by** *blast*
thus *ide-of-set A* ∈ *Collect ide*
using *ide-of-set-def* **by** *simp*
qed
show *3*: ⋀*a*. *ide a* ⟹ *ide-of-set* (*set-of-ide a*) = *a*
using *arr-of-img-elem-of-img ide-of-set-def mkIde-set set-of-ide-def setp-set-ide*
by *presburger*
show *4*: ⋀*A*. *Setp A* ⟹ *set-of-ide* (*ide-of-set A*) = *A*
proof −
fix *A* :: *'e set*
assume *A*: *Setp A*
have *elem-of* ‘ *set* (*mkIde* (*arr-of* ‘ *A*)) = *elem-of* ‘ *arr-of* ‘ *A*
proof −
have *arr-of* ‘ *A* ⊆ *Univ*

```
        using A arr-of-mapsto by blast
      moreover have Setp (elem-of ‘ arr-of ‘ A)
        using A by simp
      ultimately have set (mkIde (arr-of ‘ A)) = arr-of ‘ A
        using A set-mkIde by blast
      thus ?thesis by auto
    qed
    also have ... = A
      using A elem-of-arr-of by force
    finally show set-of-ide (ide-of-set A) = A
      using ide-of-set-def set-of-ide-def by simp
  qed
  show bij-betw set-of-ide (Collect ide) (Collect Setp)
    using 1 2 3 4
    by (intro bij-betwI) blast+
  show bij-betw ide-of-set (Collect Setp) (Collect ide)
    using 1 2 3 4
    by (intro bij-betwI) blast+
qed

definition fun-of-arr :: 'e setcat.arr ⇒ 'e ⇒ 'e
where fun-of-arr f ≡ restrict (elem-of o Fun f o arr-of) (elem-of ‘Dom f)

definition arr-of-fun :: 'e set ⇒ 'e set ⇒ ('e ⇒ 'e) ⇒ 'e setcat.arr
where arr-of-fun A B F ≡ mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of o F o elem-of)

lemma bij-betw-hom-fun:
shows fun-of-arr ∈ hom a b → extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b)
and ⟦Setp A; Setp B⟧ ⟹ arr-of-fun A B ∈ (A → B) → hom (ide-of-set A) (ide-of-set B)
and f ∈ hom a b ⟹ arr-of-fun (set-of-ide a) (set-of-ide b) (fun-of-arr f) = f
and ⟦Setp A; Setp B; F ∈ A → B; F ∈ extensional A⟧ ⟹ fun-of-arr (arr-of-fun A B F) =
F
and ⟦ide a; ide b⟧ ⟹ bij-betw fun-of-arr (hom a b)
                        (extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b))
and ⟦Setp A; Setp B⟧ ⟹
      bij-betw (arr-of-fun A B)
               (extensional A ∩ (A → B)) (hom (ide-of-set A) (ide-of-set B))
proof −
  show 1: ⋀a b. fun-of-arr ∈
                  hom a b → extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b)
  proof
    fix a b f
    assume f: f ∈ hom a b
    have Dom: Dom f = arr-of ‘ set-of-ide a
      using f set-of-ide-def
      by (metis (mono-tags, lifting) arr-dom arr-mkIde bij-betw-ide-set(3)
          ide-dom ide-of-set-def in-homE mem-Collect-eq set-mkIde)
    have Cod: Cod f = arr-of ‘ set-of-ide b
      using f set-of-ide-def
```

```
    by (metis (mono-tags, lifting) arr-cod arr-mkIde bij-betw-ide-set(3)
        ide-cod ide-of-set-def in-homE mem-Collect-eq set-mkIde)
  have fun-of-arr f ∈ set-of-ide a → set-of-ide b
  proof
    fix x
    assume x: x ∈ set-of-ide a
    have 1: arr-of x ∈ Dom f
      using x Dom by blast
    hence Fun f (arr-of x) ∈ Cod f
      using f Fun-mapsto Cod by blast
    hence elem-of (Fun f (arr-of x)) ∈ set-of-ide b
      using Cod by auto
    hence restrict (elem-of o Fun f o arr-of) (elem-of ‘ Dom f) x ∈ set-of-ide b
      using 1 by force
    thus fun-of-arr f x ∈ set-of-ide b
      unfolding fun-of-arr-def by auto
  qed
  moreover have fun-of-arr f ∈ extensional (set-of-ide a)
    unfolding fun-of-arr-def
    using set-of-ide-def f by blast
  ultimately show fun-of-arr f ∈ extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b)
    by blast
qed
show 2: ⋀A B. ⟦Setp A; Setp B⟧ ⟹
                arr-of-fun A B ∈ (A → B) → hom (ide-of-set A) (ide-of-set B)
proof
  fix x and A B :: 'e set
  assume A: Setp A and B: Setp B
  assume x: x ∈ A → B
  show arr-of-fun A B x ∈ hom (ide-of-set A) (ide-of-set B)
  proof
    show «arr-of-fun A B x : ide-of-set A → ide-of-set B»
    proof
      show 1: arr (arr-of-fun A B x)
      proof −
        have arr-of ‘ A ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ A)
          using A arr-of-mapsto elem-of-arr-of
          by (metis (no-types, lifting) PiE arr-mkIde bij-betw-ide-set(2)
              ide-implies-incl ide-of-set-def incl-def mem-Collect-eq)
        moreover have arr-of ‘ B ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ B)
          using B arr-of-mapsto elem-of-arr-of
          by (metis (no-types, lifting) PiE arr-mkIde bij-betw-ide-set(2)
              ide-implies-incl ide-of-set-def incl-def mem-Collect-eq)
        moreover have arr-of ∘ x ∘ elem-of ∈ arr-of ‘ A → arr-of ‘ B
          using x by auto
        ultimately show ?thesis
          unfolding arr-of-fun-def by blast
      qed
      show dom (arr-of-fun A B x) = ide-of-set A
```

```
      using 1 dom-mkArr ide-of-set-def arr-of-fun-def by simp
    show cod (arr-of-fun A B x) = ide-of-set B
      using 1 cod-mkArr ide-of-set-def arr-of-fun-def by simp
  qed
 qed
qed
show 3: ⋀a b f. f ∈ hom a b ⟹ arr-of-fun (set-of-ide a) (set-of-ide b) (fun-of-arr f) = f
proof −
 fix a b f
 assume f: f ∈ hom a b
 have 1: Dom f = set a ∧ Cod f = set b
  using f by blast
 have Dom: Dom f ⊆ Univ ∧ Setp (elem-of ‘ Dom f)
  using setp-set-ide f ide-dom by blast
 have Cod: Cod f ⊆ Univ ∧ Setp (elem-of ‘ Cod f)
  using setp-set-ide f ide-cod by blast
 have mkArr (set a) (set b)
        (arr-of ∘ restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘ elem-of) =
      mkArr (Dom f) (Cod f)
        (arr-of ∘ restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘ elem-of)
  using 1 by simp
 also have ... = mkArr (Dom f) (Cod f) (Fun f)
 proof (intro mkArr-eqI′)
  show 2: ⋀x. x ∈ Dom f ⟹
            (arr-of ∘
              restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘
                elem-of) x =
            Fun f x
  proof −
   fix x
   assume x: x ∈ Dom f
   hence 1: elem-of x ∈ elem-of ‘ Dom f
    by blast
   have (arr-of ∘ restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘ elem-of) x =
        arr-of (restrict (elem-of o Fun f o arr-of) (elem-of ‘ Dom f) (elem-of x))
    by auto
   also have ... = arr-of ((elem-of o Fun f o arr-of) (elem-of x))
    using 1 by auto
   also have ... = arr-of (elem-of (Fun f (arr-of (elem-of x))))
    by auto
   also have ... = arr-of (elem-of (Fun f x))
    using x arr-of-elem-of ‹Dom f ⊆ Univ ∧ Setp (elem-of ‘ Dom f)› by auto
   also have ... = Fun f x
    using x f Dom Cod Fun-mapsto arr-of-elem-of [of Fun f x] by blast
   finally show (arr-of ∘
                  restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘
                    elem-of) x =
                 Fun f x
    by blast
```

```
    qed
    have arr-of ∘ restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘ elem-of
          ∈ Dom f → Cod f
    proof
      fix x
      assume x: x ∈ Dom f
      have (arr-of ∘ restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘ elem-of) x =
          Fun f x
        using 2 x by blast
      moreover have ... ∈ Cod f
        using f x Fun-mapsto by blast
      ultimately show (arr-of ∘
                       restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘
                        elem-of) x
                      ∈ Cod f
        by argo
    qed
    thus arr (mkArr (Dom f) (Cod f)
                 (arr-of ∘
                    restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘
                     elem-of))
      using Dom Cod by blast
  qed
  finally have mkArr (set a) (set b)
                   (arr-of ∘
                      restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) ∘
                       elem-of) = f
    using f mkArr-Fun
    by (metis (no-types, lifting) in-homE mem-Collect-eq)
  thus arr-of-fun (set-of-ide a) (set-of-ide b) (fun-of-arr f) = f
    using 1 f
    by (metis (no-types, lifting) Cod Dom arr-of-img-elem-of-img arr-of-fun-def
        fun-of-arr-def set-of-ide-def)
qed
show 4: ⋀A B F. ⟦Setp A; Setp B; F ∈ A → B; F ∈ extensional A⟧
                ⟹ fun-of-arr (arr-of-fun A B F) = F
proof
  fix F and A B :: 'e set
  assume A: Setp A and B: Setp B
  assume F: F ∈ A → B and ext: F ∈ extensional A
  have 4: arr (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of ∘ F ∘ elem-of))
  proof −
    have arr-of ‘ A ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ A)
      using A
      by (metis (no-types, lifting) PiE arr-mkIde bij-betw-ide-set(2) ide-implies-incl
          ide-of-set-def incl-def mem-Collect-eq)
    moreover have arr-of ‘ B ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ B)
      using B
      by (metis (no-types, lifting) PiE arr-mkIde bij-betw-ide-set(2) ide-implies-incl
```

```
        ide-of-set-def incl-def mem-Collect-eq)
    moreover have arr-of ∘ F ∘ elem-of ∈ arr-of ‘ A → arr-of ‘ B
      using F by auto
    ultimately show ?thesis by blast
  qed
  show ⋀x. fun-of-arr (arr-of-fun A B F) x = F x
  proof −
    fix x
    have fun-of-arr (arr-of-fun A B F) x =
         restrict (elem-of ∘
                   Fun (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of ∘ F ∘ elem-of)) ∘
                     arr-of) A x
    proof −
      have elem-of ‘ Dom (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of ∘ F ∘ elem-of)) = A
        using A 4 elem-of-arr-of dom-mkArr set-of-ide-def bij-betw-ide-set(4) ide-of-set-def
        by auto
      thus ?thesis
        using arr-of-fun-def fun-of-arr-def by auto
    qed
    also have ... = F x
    proof (cases x ∈ A)
      show x ∉ A ⟹ ?thesis
        using ext extensional-arb by fastforce
      assume x: x ∈ A
      show restrict
             (elem-of ∘
                Fun (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of ∘ F ∘ elem-of)) ∘
                  arr-of) A x =
           F x
        using x 4 Fun-mkArr by simp
    qed
    finally show fun-of-arr (arr-of-fun A B F) x = F x
      by blast
  qed
qed
show ⟦Setp A; Setp B⟧ ⟹
       bij-betw (arr-of-fun A B) (extensional A ∩ (A → B))
                (hom (ide-of-set A) (ide-of-set B))
proof −
  assume A: Setp A and B: Setp B
  have ide (ide-of-set A) ∧ ide (ide-of-set B)
    using A B bij-betw-ide-set(2) by auto
  have set-of-ide (ide-of-set A) = A ∧ set-of-ide (ide-of-set B) = B
    using A B by simp
  show ?thesis
    using A B 1 [of ide-of-set A ide-of-set B] 2 3 4
    apply (intro bij-betwI)
      apply auto
     apply blast
```

```
    by fastforce
  qed
  show ⟦ide a; ide b⟧ ⟹ bij-betw fun-of-arr (hom a b)
                          (extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b))
  proof (intro bij-betwI)
    assume a: ide a and b: ide b
    show fun-of-arr ∈ hom a b → extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b)
      using 1 by blast
    show arr-of-fun (set-of-ide a) (set-of-ide b) ∈
          extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b) → hom a b
      using a b 2 [of set-of-ide a set-of-ide b] setp-set-ide set-of-ide-def
            bij-betw-ide-set(3)
      by auto
    show ⋀f. f ∈ hom a b ⟹ arr-of-fun (set-of-ide a) (set-of-ide b) (fun-of-arr f) = f
      using a b 3 by blast
    show ⋀F. F ∈ extensional (set-of-ide a) ∩ (set-of-ide a → set-of-ide b) ⟹
               fun-of-arr (arr-of-fun (set-of-ide a) (set-of-ide b) F) = F
      using a b 4 [of set-of-ide a set-of-ide b]
      by (metis (no-types, lifting) IntE set-of-ide-def setp-set-ide)
  qed
qed

lemma fun-of-arr-ide:
assumes ide a
shows fun-of-arr a = restrict id (elem-of ‘ Dom a)
proof
  fix x
  show fun-of-arr a x = restrict id (elem-of ‘ Dom a) x
  proof (cases x ∈ elem-of ‘ Dom a)
    show x ∉ elem-of ‘ Dom a ⟹ ?thesis
      using fun-of-arr-def extensional-arb by auto
    assume x: x ∈ elem-of ‘ Dom a
    have fun-of-arr a x = restrict (elem-of ∘ Fun a ∘ arr-of) (elem-of ‘ Dom a) x
      using x fun-of-arr-def by simp
    also have ... = elem-of (Fun a (arr-of x))
      using x by auto
    also have ... = elem-of ((λx∈set a. x) (arr-of x))
      using assms x Fun-ide by auto
    also have ... = elem-of (arr-of x)
    proof −
      have x ∈ elem-of ‘ set a
        using assms x ideD(2) by force
      hence arr-of x ∈ set a
        by (metis (mono-tags, lifting) arr-of-img-elem-of-img assms image-eqI setp-set-ide)
      thus ?thesis by simp
    qed
    also have ... = x
      by simp
    also have ... = restrict id (elem-of ‘ Dom a) x
```

```
      using x by auto
    finally show ?thesis by blast
  qed
qed

lemma arr-of-fun-id:
assumes Setp A
shows arr-of-fun A A (restrict id A) = ide-of-set A
proof −
  have A: arr-of ‘ A ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ A)
    using assms elem-of-arr-of
    by (metis (no-types, lifting) PiE arr-mkIde bij-betw-ide-set(2) ide-implies-incl
        ide-of-set-def incl-def mem-Collect-eq)
  have arr-of-fun A A (restrict id A) =
        mkArr (arr-of ‘ A) (arr-of ‘ A) (arr-of ∘ restrict id A ∘ elem-of)
    unfolding arr-of-fun-def by simp
  also have ... = mkArr (arr-of ‘ A) (arr-of ‘ A) (λx. x)
    using A arr-mkArr
    by (intro mkArr-eqI′) auto
  also have ... = ide-of-set A
    using A ide-of-set-def mkIde-as-mkArr by simp
  finally show ?thesis by blast
qed

lemma fun-of-arr-comp:
assumes f ∈ hom a b and g ∈ hom b c
shows fun-of-arr (comp g f) = restrict (fun-of-arr g ∘ fun-of-arr f) (set-of-ide a)
proof −
  have 1: seq g f
    using assms by blast
  have fun-of-arr (comp g f) =
        restrict (elem-of ∘ Fun (comp g f) ∘ arr-of) (elem-of ‘ Dom (comp g f))
    unfolding fun-of-arr-def by blast
  also have ... = restrict (elem-of ∘ Fun (comp g f) ∘ arr-of) (elem-of ‘ Dom f)
    using assms dom-comp seqI′ by auto
  also have ... = restrict (elem-of ∘ restrict (Fun g ∘ Fun f) (Dom f) ∘ arr-of)
                           (elem-of ‘ Dom f)
    using 1 Fun-comp by auto
  also have ... = restrict (restrict (elem-of o Fun g o arr-of) (elem-of ‘ Dom g) ∘
                              restrict (elem-of o Fun f o arr-of) (elem-of ‘ Dom f))
                           (elem-of ‘ Dom f)
  proof
    fix x
    show restrict (elem-of ∘ restrict (Fun g ∘ Fun f) (Dom f) ∘ arr-of) (elem-of ‘ Dom f) x
=
          restrict (restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g) ∘
                    restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f))
                   (elem-of ‘ Dom f) x
    proof (cases x ∈ elem-of ‘ Dom f)
```

```
      show x ∉ elem-of ‘ Dom f ⟹ ?thesis
        by auto
      assume x: x ∈ elem-of ‘ Dom f
      have 2: arr-of x ∈ Dom f
      proof −
        have arr-of x ∈ arr-of ‘ elem-of ‘ Dom f
          using x by simp
        thus ?thesis
          by (metis (no-types, lifting) 1 arr-of-img-elem-of-img ide-dom seqE setp-set-ide)
      qed
      have 3: Dom g = Cod f
        using assms by fastforce
      have restrict (elem-of ∘ restrict (Fun g ∘ Fun f) (Dom f) ∘ arr-of) (elem-of ‘ Dom f)
x =
          elem-of (Fun g (Fun f (arr-of x)))
        using x 2 by simp
      also have ... = restrict
                        (restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g) ∘
                          restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f))
                        (elem-of ‘ Dom f) x
      proof −
        have restrict (restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g) ∘
                         restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f))
                       (elem-of ‘ Dom f) x =
             elem-of (Fun g (Fun f (arr-of x)))
        proof −
          have restrict (restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g) ∘
                           restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f))
                         (elem-of ‘ Dom f) x =
               (restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g) o
                 restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f)) x
            using 2 by force
          also have ... = restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g)
                            (restrict (elem-of ∘ Fun f ∘ arr-of) (elem-of ‘ Dom f) x)
            by simp
          also have ... = restrict (elem-of ∘ Fun g ∘ arr-of) (elem-of ‘ Dom g)
                            (elem-of (Fun f (arr-of x)))
            using 2 by force
          also have ... = (elem-of o Fun g o arr-of) (elem-of (Fun f (arr-of x)))
          proof −
            have elem-of (Fun f (arr-of x)) ∈ elem-of ‘ Dom g
              using assms 2 3 Fun-mapsto [of f] by blast
            thus ?thesis by simp
          qed
          also have ... = elem-of (Fun g (arr-of (elem-of (Fun f (arr-of x)))))
            by simp
          also have ... = elem-of (Fun g (Fun f (arr-of x)))
          proof −
            have Fun f (arr-of x) ∈ Univ
```

```
            using assms 2 setp-set-ide ide-cod Fun-mapsto by blast
          thus ?thesis
            using 2 by simp
        qed
        finally show ?thesis by blast
      qed
      thus ?thesis by simp
    qed
    finally show ?thesis by blast
  qed
qed
also have ... = restrict (fun-of-arr g o fun-of-arr f) (elem-of ‘ Dom f)
  unfolding fun-of-arr-def by blast
finally show ?thesis
  unfolding set-of-ide-def
  using assms by blast
qed

lemma arr-of-fun-comp:
assumes Setp A and Setp B and Setp C
and F ∈ extensional A ∩ (A → B) and G ∈ extensional B ∩ (B → C)
shows arr-of-fun A C (G o F) = comp (arr-of-fun B C G) (arr-of-fun A B F)
proof −
  have A: arr-of ‘ A ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ A)
    using assms elem-of-arr-of
    by (metis (no-types, lifting) Pi-iff arr-mkIde bij-betw-ide-set(2)
        ide-implies-incl ide-of-set-def incl-def mem-Collect-eq)
  have B: arr-of ‘ B ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ B)
    using assms elem-of-arr-of
    by (metis (no-types, lifting) Pi-iff arr-mkIde bij-betw-ide-set(2)
        ide-implies-incl ide-of-set-def incl-def mem-Collect-eq)
  have C: arr-of ‘ C ⊆ Univ ∧ Setp (elem-of ‘ arr-of ‘ C)
    using assms elem-of-arr-of
    by (metis (no-types, lifting) Pi-iff arr-mkIde bij-betw-ide-set(2)
        ide-implies-incl ide-of-set-def incl-def mem-Collect-eq)
  have arr-of-fun A C (G o F) = mkArr (arr-of ‘ A) (arr-of ‘ C) (arr-of ∘ (G ∘ F) ∘
elem-of)
    unfolding arr-of-fun-def by simp
  also have ... = mkArr (arr-of ‘ A) (arr-of ‘ C)
                        ((arr-of ∘ G ∘ elem-of) o (arr-of o F ∘ elem-of))
  proof (intro mkArr-eqI′)
    show arr (mkArr (arr-of ‘ A) (arr-of ‘ C) (arr-of ∘ (G ∘ F) ∘ elem-of))
    proof −
      have arr-of ∘ (G ∘ F) ∘ elem-of ∈ arr-of ‘ A → arr-of ‘ C
        using assms by force
      thus ?thesis
        using A B C by blast
    qed
    show ⋀x. x ∈ arr-of ‘ A ⟹
```

```
            (arr-of ∘ (G ∘ F) ∘ elem-of) x =
            ((arr-of ∘ G ∘ elem-of) o (arr-of o F ∘ elem-of)) x
      by simp
  qed
  also have ... = comp (mkArr (arr-of ‘ B) (arr-of ‘ C) (arr-of ∘ G ∘ elem-of))
                       (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of o F o elem-of))
  proof −
    have arr (mkArr (arr-of ‘ B) (arr-of ‘ C) (arr-of ∘ G ∘ elem-of))
      using assms B C elem-of-arr-of by fastforce
    moreover have arr (mkArr (arr-of ‘ A) (arr-of ‘ B) (arr-of o F o elem-of))
      using assms A B elem-of-arr-of by fastforce
    ultimately show ?thesis
      using comp-mkArr by auto
  qed
  also have ... = comp (arr-of-fun B C G) (arr-of-fun A B F)
    using arr-of-fun-def by presburger
  finally show ?thesis by blast
qed

end
```

When there is no restriction on the sets that determine objects, the resulting set category is replete. This is the normal use case, which we want to streamline as much as possible, so it is useful to introduce a special locale for this purpose.

```
locale replete-setcat =
fixes elem-type :: 'e itself
begin

  interpretation SC: setcat elem-type ‹λ-. True›
    by unfold-locales blast

  definition comp
  where comp ≡ SC.comp

  definition arr-of
  where arr-of ≡ SC.arr-of

  definition elem-of
  where elem-of ≡ SC.elem-of

  sublocale replete-set-category comp
    unfolding comp-def
    using SC.is-set-category replete-set-category-def set-category-def by auto

  lemma is-replete-set-category:
  shows replete-set-category comp
    ..

  lemma is-set-category_RSC:
```

shows *set-category comp* ($\lambda A.\ A \subseteq Univ$)
using *is-set-category* **by** *blast*

sublocale *concrete-set-category comp setp UNIV arr-of*
using *SC.is-concrete-set-category comp-def SC.set-img-Collect-ide-iff arr-of-def*
by *auto*

lemma *is-concrete-set-category*:
shows *concrete-set-category comp setp UNIV arr-of*
..

lemma *bij-arr-of*:
shows *bij-betw arr-of UNIV Univ*
using *SC.bij-arr-of comp-def arr-of-def* **by** *presburger*

lemma *bij-elem-of*:
shows *bij-betw elem-of Univ UNIV*
using *SC.bij-elem-of comp-def elem-of-def* **by** *auto*

end

end

Chapter 11

ProductCategory

theory *ProductCategory*
imports *Category EpiMonoIso*
begin

This theory defines the product of two categories *C1* and *C2*, which is the category *C* whose arrows are ordered pairs consisting of an arrow of *C1* and an arrow of *C2*, with composition defined componentwise. As the ordered pair (*C1.null*, *C2.null*) is available to serve as *C.null*, we may directly identify the arrows of the product category *C* with ordered pairs, leaving the type of arrows of *C* transparent.

locale *product-category* =
  C1: *category C1* +
  C2: *category C2*
for *C1* :: *'a1 comp* (**infixr** ‹·$_1$› *55*)
and *C2* :: *'a2 comp* (**infixr** ‹·$_2$› *55*)
begin

  type-synonym (*'aa1*, *'aa2*) *arr* = *'aa1* ∗ *'aa2*

  notation *C1.in-hom* (‹«- : - →$_1$ -»›)
  notation *C2.in-hom* (‹«- : - →$_2$ -»›)

  abbreviation (*input*) *Null* :: (*'a1*, *'a2*) *arr*
  where *Null* ≡ (*C1.null*, *C2.null*)

  abbreviation (*input*) *Arr* :: (*'a1*, *'a2*) *arr* ⇒ *bool*
  where *Arr f* ≡ *C1.arr* (*fst f*) ∧ *C2.arr* (*snd f*)

  abbreviation (*input*) *Ide* :: (*'a1*, *'a2*) *arr* ⇒ *bool*
  where *Ide f* ≡ *C1.ide* (*fst f*) ∧ *C2.ide* (*snd f*)

  abbreviation (*input*) *Dom* :: (*'a1*, *'a2*) *arr* ⇒ (*'a1*, *'a2*) *arr*
  where *Dom f* ≡ (*if Arr f then* (*C1.dom* (*fst f*), *C2.dom* (*snd f*)) *else Null*)

  abbreviation (*input*) *Cod* :: (*'a1*, *'a2*) *arr* ⇒ (*'a1*, *'a2*) *arr*

where *Cod f* ≡ (*if Arr f then* (*C1.cod* (*fst f*), *C2.cod* (*snd f*)) *else Null*)

definition *comp* :: ('*a1*, '*a2*) *arr* ⇒ ('*a1*, '*a2*) *arr* ⇒ ('*a1*, '*a2*) *arr*
where *comp g f* = (*if Arr f* ∧ *Arr g* ∧ *Cod f* = *Dom g then*
(*C1* (*fst g*) (*fst f*), *C2* (*snd g*) (*snd f*))
else Null)

notation *comp* (**infixr** ‹·› *55*)

lemma *not-Arr-Null*:
shows ¬*Arr Null*
by *simp*

interpretation *partial-composition comp*
proof
show ∃!*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n*
proof
let *?P* = λ*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n*
show *1*: *?P Null* **using** *comp-def not-Arr-Null* **by** *metis*
thus ⋀*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n* ⟹ *n* = *Null* **by** *metis*
qed
qed

notation *in-hom* (‹«- : - → -»›)

lemma *null-char* [*simp*]:
shows *null* = *Null*
proof −
let *?P* = λ*n*. ∀*f*. *n* · *f* = *n* ∧ *f* · *n* = *n*
have *?P Null* **using** *comp-def not-Arr-Null* **by** *metis*
thus *?thesis*
unfolding *null-def* **using** *the1-equality* [*of ?P Null*] *ex-un-null* **by** *blast*
qed

lemma *ide-Ide*:
assumes *Ide a*
shows *ide a*
unfolding *ide-def comp-def null-char*
using *assms C1.not-arr-null C1.ide-in-hom C1.comp-arr-dom C1.comp-cod-arr*
C2.comp-arr-dom C2.comp-cod-arr
by *auto*

lemma *has-domain-char*:
shows *domains f* ≠ {} ⟷ *Arr f*
proof
show *domains f* ≠ {} ⟹ *Arr f*
unfolding *domains-def comp-def null-char* **by** (*auto*; *metis*)
assume *f*: *Arr f*
show *domains f* ≠ {}

```
  proof −
    have ide (Dom f) ∧ comp f (Dom f) ≠ null
      using f comp-def ide-Ide C1.comp-arr-dom C1.arr-dom-iff-arr C2.arr-dom-iff-arr
      by auto
    thus ?thesis using domains-def by blast
  qed
qed

lemma has-codomain-char:
shows codomains f ≠ {} ⟷ Arr f
proof
  show codomains f ≠ {} ⟹ Arr f
    unfolding codomains-def comp-def null-char by (auto; metis)
  assume f: Arr f
  show codomains f ≠ {}
  proof −
    have ide (Cod f) ∧ comp (Cod f) f ≠ null
      using f comp-def ide-Ide C1.comp-cod-arr C1.arr-cod-iff-arr C2.arr-cod-iff-arr
      by auto
    thus ?thesis using codomains-def by blast
  qed
qed

lemma arr-char [iff]:
shows arr f ⟷ Arr f
  using has-domain-char has-codomain-char arr-def by simp

lemma arrI_PC [intro]:
assumes C1.arr f1 and C2.arr f2
shows arr (f1, f2)
  using assms by simp

lemma arrE:
assumes arr f
and C1.arr (fst f) ∧ C2.arr (snd f) ⟹ T
shows T
  using assms by auto

lemma seqI_PC [intro]:
assumes C1.seq g1 f1 ∧ C2.seq g2 f2
shows seq (g1, g2) (f1, f2)
  using assms comp-def by auto

lemma seqE_PC [elim]:
assumes seq g f
and C1.seq (fst g) (fst f) ⟹ C2.seq (snd g) (snd f) ⟹ T
shows T
  using assms comp-def
  by (metis (no-types, lifting) C1.seqI C2.seqI Pair-inject not-arr-null null-char)
```

lemma *seq-char* [*iff*]:
shows *seq g f* ⟷ *C1.seq* (*fst g*) (*fst f*) ∧ *C2.seq* (*snd g*) (*snd f*)
using *comp-def* **by** *auto*

lemma *Dom-comp*:
assumes *seq g f*
shows *Dom* (*g* · *f*) = *Dom f*
proof −
have *C1.arr* (*fst f*) ∧ *C1.arr* (*fst g*) ∧ *C1.dom* (*fst g*) = *C1.cod* (*fst f*)
using *assms* **by** *blast*
moreover have *C2.arr* (*snd f*) ∧ *C2.arr* (*snd g*) ∧ *C2.dom* (*snd g*) = *C2.cod* (*snd f*)
using *assms* **by** *blast*
ultimately show *?thesis*
by (*simp add*: *comp-def*)
qed

lemma *Cod-comp*:
assumes *seq g f*
shows *Cod* (*g* · *f*) = *Cod g*
proof −
have *C1.arr* (*fst f*) ∧ *C1.arr* (*fst g*) ∧ *C1.dom* (*fst g*) = *C1.cod* (*fst f*)
using *assms* **by** *blast*
moreover have *C2.arr* (*snd f*) ∧ *C2.arr* (*snd g*) ∧ *C2.dom* (*snd g*) = *C2.cod* (*snd f*)
using *assms* **by** *blast*
ultimately show *?thesis*
by (*simp add*: *comp-def*)
qed

theorem *is-category*:
shows *category comp*
proof
fix *f*
show (*domains f* ≠ {}) = (*codomains f* ≠ {})
using *has-domain-char has-codomain-char* **by** *simp*
fix *g*
show *g* · *f* ≠ *null* ⟹ *seq g f*
using *comp-def seq-char* **by** (*metis C1.seqI C2.seqI Pair-inject null-char*)
fix *h*
show *seq h g* ⟹ *seq* (*h* · *g*) *f* ⟹ *seq g f*
using *seq-char*
by (*metis category.seqE category.seqI Dom-comp*
product-category-axioms product-category-def fst-conv snd-conv)
show *seq h* (*g* · *f*) ⟹ *seq g f* ⟹ *seq h g*
using *seq-char*
by (*metis category.seqE category.seqI Cod-comp*
product-category-axioms product-category-def fst-conv snd-conv)
show *seq g f* ⟹ *seq h g* ⟹ *seq* (*h* · *g*) *f*
using *seq-char*

by (*metis arrE category.seqE category.seqI Dom-comp*
product-category-axioms product-category-def fst-conv snd-conv)
show *seq g f* $\Longrightarrow$ *seq h g* $\Longrightarrow$ $(h \cdot g) \cdot f = h \cdot g \cdot f$
using *comp-def null-char seq-char C1.comp-assoc C2.comp-assoc*
by (*elim seqE*$_{PC}$ *C1.seqE C2.seqE*, *simp*)
qed

sublocale *category comp*
using *is-category comp-def* **by** *auto*

lemma *dom-char*:
shows *dom f* = *Dom f*
proof (*cases Arr f*)
show ¬*Arr f* $\Longrightarrow$ *dom f* = *Dom f*
unfolding *dom-def* **using** *has-domain-char* **by** *auto*
show *Arr f* $\Longrightarrow$ *dom f* = *Dom f*
using *ide-Ide* **apply** (*intro dom-eqI*, *simp*)
using *seq-char comp-def C1.arr-dom-iff-arr C2.arr-dom-iff-arr* **by** *auto*
qed

lemma *dom-simp* [*simp*]:
assumes *arr f*
shows *dom f* = (*C1.dom* (*fst f*), *C2.dom* (*snd f*))
using *assms dom-char* **by** *auto*

lemma *cod-char*:
shows *cod f* = *Cod f*
proof (*cases Arr f*)
show ¬*Arr f* $\Longrightarrow$ *cod f* = *Cod f*
unfolding *cod-def* **using** *has-codomain-char* **by** *auto*
show *Arr f* $\Longrightarrow$ *cod f* = *Cod f*
using *ide-Ide seqI* **apply** (*intro cod-eqI*, *simp*)
using *seq-char comp-def C1.arr-cod-iff-arr C2.arr-cod-iff-arr* **by** *auto*
qed

lemma *cod-simp* [*simp*]:
assumes *arr f*
shows *cod f* = (*C1.cod* (*fst f*), *C2.cod* (*snd f*))
using *assms cod-char* **by** *auto*

lemma *in-homI*$_{PC}$ [*intro*, *simp*]:
assumes «*fst f*: *fst a* $\rightarrow_1$ *fst b*» **and** «*snd f*: *snd a* $\rightarrow_2$ *snd b*»
shows «*f*: *a* $\rightarrow$ *b*»
using *assms* **by** *fastforce*

lemma *in-homE*$_{PC}$ [*elim*]:
assumes «*f*: *a* $\rightarrow$ *b*»
and «*fst f*: *fst a* $\rightarrow_1$ *fst b*» $\Longrightarrow$ «*snd f*: *snd a* $\rightarrow_2$ *snd b*» $\Longrightarrow$ *T*
shows *T*

using *assms*
by (*metis C1.in-homI C2.in-homI arr-char cod-simp dom-simp fst-conv in-homE snd-conv*)

lemma *ide-char*$_{PC}$ [*iff*]:
shows *ide f* $\longleftrightarrow$ *Ide f*
using *ide-in-hom C1.ide-in-hom C2.ide-in-hom* **by** *blast*

lemma *comp-char*:
shows *g* $\cdot$ *f* = (*if C1.arr* (*C1* (*fst g*) (*fst f*)) $\wedge$ *C2.arr* (*C2* (*snd g*) (*snd f*)) *then*
(*C1* (*fst g*) (*fst f*), *C2* (*snd g*) (*snd f*))
else Null)
using *comp-def* **by** *auto*

lemma *comp-simp* [*simp*]:
assumes *C1.seq* (*fst g*) (*fst f*) **and** *C2.seq* (*snd g*) (*snd f*)
shows *g* $\cdot$ *f* = (*fst g* $\cdot_1$ *fst f*, *snd g* $\cdot_2$ *snd f*)
using *assms comp-char* **by** *simp*

lemma *iso-char* [*iff*]:
shows *iso f* $\longleftrightarrow$ *C1.iso* (*fst f*) $\wedge$ *C2.iso* (*snd f*)
proof
assume *f*: *iso f*
obtain *g* **where** *g*: *inverse-arrows f g* **using** *f* **by** *auto*
have *1*: *ide* (*g* $\cdot$ *f*) $\wedge$ *ide* (*f* $\cdot$ *g*)
using *f g* **by** (*simp add*: *inverse-arrows-def*)
have *g* $\cdot$ *f* = (*fst g* $\cdot_1$ *fst f*, *snd g* $\cdot_2$ *snd f*) $\wedge$ *f* $\cdot$ *g* = (*fst f* $\cdot_1$ *fst g*, *snd f* $\cdot_2$ *snd g*)
using *1 comp-char arr-char* **by** (*meson ideD*(*1*) *seq-char*)
hence *C1.ide* (*fst g* $\cdot_1$ *fst f*) $\wedge$ *C2.ide* (*snd g* $\cdot_2$ *snd f*) $\wedge$
C1.ide (*fst f* $\cdot_1$ *fst g*) $\wedge$ *C2.ide* (*snd f* $\cdot_2$ *snd g*)
using *1 ide-char* **by** *simp*
hence *C1.inverse-arrows* (*fst f*) (*fst g*) $\wedge$ *C2.inverse-arrows* (*snd f*) (*snd g*)
by *auto*
thus *C1.iso* (*fst f*) $\wedge$ *C2.iso* (*snd f*) **by** *auto*
next
assume *f*: *C1.iso* (*fst f*) $\wedge$ *C2.iso* (*snd f*)
obtain *g1* **where** *g1*: *C1.inverse-arrows* (*fst f*) *g1* **using** *f* **by** *blast*
obtain *g2* **where** *g2*: *C2.inverse-arrows* (*snd f*) *g2* **using** *f* **by** *blast*
have *C1.ide* (*g1* $\cdot_1$ *fst f*) $\wedge$ *C2.ide* (*g2* $\cdot_2$ *snd f*) $\wedge$
C1.ide (*fst f* $\cdot_1$ *g1*) $\wedge$ *C2.ide* (*snd f* $\cdot_2$ *g2*)
using *g1 g2 ide-char*$_{PC}$ **by** *force*
hence *inverse-arrows f* (*g1*, *g2*)
using *f g1 g2 ide-char*$_{PC}$ *comp-char* **by** (*intro inverse-arrowsI*, *auto*)
thus *iso f* **by** *auto*
qed

lemma *isoI*$_{PC}$ [*intro*, *simp*]:
assumes *C1.iso* (*fst f*) **and** *C2.iso* (*snd f*)
shows *iso f*
using *assms* **by** *simp*

```
    lemma isoD:
    assumes iso f
    shows C1.iso (fst f) and C2.iso (snd f)
      using assms by auto

    lemma inv-simp [simp]:
    assumes iso f
    shows inv f = (C1.inv (fst f), C2.inv (snd f))
    proof −
      have inverse-arrows f (C1.inv (fst f), C2.inv (snd f))
      proof
        have 1: C1.inverse-arrows (fst f) (C1.inv (fst f))
          using assms iso-char C1.inv-is-inverse by simp
        have 2: C2.inverse-arrows (snd f) (C2.inv (snd f))
          using assms iso-char C2.inv-is-inverse by simp
        show ide ((C1.inv (fst f), C2.inv (snd f)) · f)
          using 1 2 ide-char_PC comp-char by auto
        show ide (f · (C1.inv (fst f), C2.inv (snd f)))
          using 1 2 ide-char_PC comp-char by auto
      qed
      thus ?thesis using inverse-unique by auto
    qed

  end

end
```

Chapter 12

NaturalTransformation

theory *NaturalTransformation*
imports *Functor*
begin

12.1 Definition of a Natural Transformation

As is the case for functors, the "object-free" definition of category makes it possible to view natural transformations as functions on arrows. In particular, a natural transformation between functors F and G from A to B can be represented by the map that takes each arrow f of A to the diagonal of the square in B corresponding to the transformation of $F\,f$ to $G\,f$. The images of the identities of A under this map are the usual components of the natural transformation. This representation exhibits natural transformations as a kind of generalization of functors, and in fact we can directly identify functors with identity natural transformations. However, functors are still necessary to state the defining conditions for a natural transformation, as the domain and codomain of a natural transformation cannot be recovered from the map on arrows that represents it.

Like functors, natural transformations preserve arrows and map non-arrows to null. Natural transformations also "preserve" domain and codomain, but in a more general sense than functors. The naturality conditions, which express the two ways of factoring the diagonal of a commuting square, are degenerate in the case of an identity transformation.

locale *natural-transformation* =
  A: *category* A +
  B: *category* B +
  F: *functor* $A\ B\ F$ +
  G: *functor* $A\ B\ G$
for $A :: 'a\ comp$ (**infixr** $\langle\cdot_A\rangle$ *55*)
and $B :: 'b\ comp$ (**infixr** $\langle\cdot_B\rangle$ *55*)
and $F :: 'a \Rightarrow 'b$
and $G :: 'a \Rightarrow 'b$
and $\tau :: 'a \Rightarrow 'b$ +
assumes *extensionality*: $\neg A.arr\ f \Longrightarrow \tau\ f = B.null$

and *preserves-arr* [*simp*]: $A.arr\ f \Longrightarrow B.arr\ (\tau\ f)$
and *naturality1* [*iff*]: $A.arr\ f \Longrightarrow G\ f\ \cdot_B\ \tau\ (A.dom\ f) = \tau\ f$
and *naturality2* [*iff*]: $A.arr\ f \Longrightarrow \tau\ (A.cod\ f)\ \cdot_B\ F\ f = \tau\ f$
begin

lemma *preserves-dom* [*iff*]:
assumes $A.arr\ f$
shows $B.dom\ (\tau\ f) = F\ (A.dom\ f)$
using *assms naturality2 B.dom-comp* [*of* τ $(A.cod\ f)$ $F\ f$] **by** *auto*

lemma *preserves-cod* [*iff*]:
assumes $A.arr\ f$
shows $B.cod\ (\tau\ f) = G\ (A.cod\ f)$
using *assms naturality2 B.cod-comp* [*of* $G\ f\ \tau\ (A.dom\ f)$] **by** *auto*

lemma *naturality*:
assumes $A.arr\ f$
shows $\tau\ (A.cod\ f)\ \cdot_B\ F\ f = G\ f\ \cdot_B\ \tau\ (A.dom\ f)$
using *assms naturality1 naturality2* **by** *simp*

The following fact for natural transformations provides us with the same advantages as the corresponding fact for functors.

lemma *preserves-reflects-arr* [*iff*]:
shows $B.arr\ (\tau\ f) \longleftrightarrow A.arr\ f$
using *extensionality A.arr-cod-iff-arr B.arr-cod-iff-arr preserves-cod* **by** *force*

lemma *preserves-hom* [*intro*]:
assumes «$f : a \rightarrow_A b$»
shows «$\tau\ f : F\ a \rightarrow_B G\ b$»
using *assms*
by (*metis A.in-homE B.arr-cod-iff-arr B.in-homI G.preserves-arr G.preserves-cod preserves-cod preserves-dom*)

lemma *preserves-comp-1*:
assumes $A.seq\ f'\ f$
shows $\tau\ (f' \cdot_A f) = G\ f' \cdot_B \tau\ f$
using *assms*
by (*metis A.seqE A.dom-comp B.comp-assoc G.preserves-comp naturality1*)

lemma *preserves-comp-2*:
assumes $A.seq\ f'\ f$
shows $\tau\ (f' \cdot_A f) = \tau\ f' \cdot_B F\ f$
using *assms*
by (*metis A.arr-cod-iff-arr A.cod-comp B.comp-assoc F.preserves-comp naturality2*)

A natural transformation that also happens to be a functor is equal to its own domain and codomain.

lemma *functor-implies-equals-dom*:
assumes *functor A B* τ

shows $F = \tau$
proof
interpret τ: *functor A B* τ **using** *assms* **by** *auto*
fix *f*
show $F\ f = \tau\ f$
using *assms*
by (*metis A.dom-cod B.comp-cod-arr F.extensionality F.preserves-arr F.preserves-cod* τ*.preserves-dom extensionality naturality2 preserves-dom*)
qed

lemma *functor-implies-equals-cod*:
assumes *functor A B* τ
shows $G = \tau$
proof
interpret τ: *functor A B* τ **using** *assms* **by** *auto*
fix *f*
show $G\ f = \tau\ f$
using *assms*
by (*metis A.cod-dom B.comp-arr-dom G.extensionality G.preserves-arr G.preserves-dom B.cod-dom functor-implies-equals-dom extensionality naturality1 preserves-cod preserves-dom*)
qed

end

12.2 Components of a Natural Transformation

The values taken by a natural transformation on identities are the *components* of the transformation. We have the following basic technique for proving two natural transformations equal: show that they have the same components.

lemma *natural-transformation-eqI*:
assumes *natural-transformation A B F G* σ **and** *natural-transformation A B F G* σ'
and $\bigwedge a$. *partial-composition.ide A a* $\Longrightarrow$ $\sigma\ a = \sigma'\ a$
shows $\sigma = \sigma'$
proof −
interpret *A*: *category A* **using** *assms*(*1*) *natural-transformation-def* **by** *blast*
interpret σ: *natural-transformation A B F G* σ **using** *assms*(*1*) **by** *auto*
interpret σ': *natural-transformation A B F G* σ' **using** *assms*(*2*) **by** *auto*
have $\bigwedge f$. $\sigma\ f = \sigma'\ f$
using *assms*(*3*) σ*.naturality2* σ'*.naturality2* σ*.extensionality* σ'*.extensionality A.ide-cod*
by *metis*
thus *?thesis* **by** *auto*
qed

As equality of natural transformations is determined by equality of components, a natural transformation may be uniquely defined by specifying its components. The extension to all arrows is given by *naturality1* or equivalently by *naturality2*.

locale *transformation-by-components* =

```
  A: category A +
  B: category B +
  F: functor A B F +
  G: functor A B G
for A :: 'a comp      (infixr ‹·A› 55)
and B :: 'b comp      (infixr ‹·B› 55)
and F :: 'a ⇒ 'b
and G :: 'a ⇒ 'b
and t :: 'a ⇒ 'b +
assumes maps-ide-in-hom [intro]: A.ide a ⟹ «t a : F a →B G a»
and is-natural: A.arr f ⟹ t (A.cod f) ·B F f = G f ·B t (A.dom f)
begin

  definition map
  where map f = (if A.arr f then t (A.cod f) ·B F f else B.null)

  lemma map-simp-ide [simp]:
  assumes A.ide a
  shows map a = t a
    using assms map-def B.comp-arr-dom [of t a] maps-ide-in-hom by fastforce

  lemma is-natural-transformation:
  shows natural-transformation A B F G map
    using map-def is-natural
    apply (unfold-locales, simp-all)
      apply (metis A.ide-dom B.seqI G.preserves-arr G.preserves-dom B.in-homE
                   maps-ide-in-hom)
     apply (metis A.ide-dom B.comp-arr-dom B.in-homE maps-ide-in-hom)
    by (metis B.comp-assoc A.comp-cod-arr F.preserves-comp)

end

sublocale transformation-by-components ⊆ natural-transformation A B F G map
  using is-natural-transformation by auto

lemma transformation-by-components-idem [simp]:
assumes natural-transformation A B F G τ
shows transformation-by-components.map A B F τ = τ
proof −
  interpret τ: natural-transformation A B F G τ using assms by blast
  interpret τ': transformation-by-components A B F G τ
    by (unfold-locales, auto)
  show ?thesis
    using assms τ'.map-simp-ide τ'.is-natural-transformation
          natural-transformation-eqI
    by blast
qed
```

12.3 Functors as Natural Transformations

A functor is a special case of a natural transformation, in the sense that the same map that defines the functor also defines an identity natural transformation.

lemma *functor-is-transformation* [*simp*]:
assumes *functor A B F*
shows *natural-transformation A B F F F*
proof −
  interpret *functor A B F* **using** *assms* **by** *auto*
  show *natural-transformation A B F F F*
    using *extensionality B.comp-arr-dom B.comp-cod-arr*
    by (*unfold-locales*, *simp-all*)
qed

sublocale *functor* ⊆ *as-nat-trans*: *natural-transformation A B F F F*
  by (*simp add*: *functor-axioms*)

12.4 Constant Natural Transformations

A constant natural transformation is one whose components are all the same arrow.

locale *constant-transformation* =
  A: *category A* +
  B: *category B* +
  F: *constant-functor A B B.dom g* +
  G: *constant-functor A B B.cod g*
for *A* :: *'a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_B$› *55*)
and *g* :: *'b* +
assumes *value-is-arr*: *B.arr g*
begin

  definition *map*
  where *map f* ≡ *if A.arr f then g else B.null*

  lemma *map-simp* [*simp*]:
  assumes *A.arr f*
  shows *map f* = *g*
    using *assms map-def* **by** *auto*

  lemma *is-natural-transformation*:
  shows *natural-transformation A B F.map G.map map*
    apply *unfold-locales*
    using *map-def value-is-arr B.comp-arr-dom B.comp-cod-arr* **by** *auto*

  lemma *is-functor-if-value-is-ide*:
  assumes *B.ide g*
  shows *functor A B map*
    apply *unfold-locales* **using** *assms map-def* **by** *auto*

end

sublocale *constant-transformation* $\subseteq$ *natural-transformation A B F.map G.map map*
using *is-natural-transformation* **by** *auto*

context *constant-transformation*
begin

lemma *equals-dom-if-value-is-ide*:
assumes *B.ide g*
shows *map* = *F.map*
using *assms functor-implies-equals-dom is-functor-if-value-is-ide* **by** *auto*

lemma *equals-cod-if-value-is-ide*:
assumes *B.ide g*
shows *map* = *G.map*
using *assms functor-implies-equals-dom is-functor-if-value-is-ide* **by** *auto*

end

12.5 Vertical Composition

Vertical composition is a way of composing natural transformations σ: $F \to G$ and τ: $G \to H$, between parallel functors F, G, and H to obtain a natural transformation from F to H. The composite is traditionally denoted by $\tau\ o\ \sigma$, however in the present setting this notation is misleading because it is horizontal composite, rather than vertical composite, that coincides with composition of natural transformations as functions on arrows.

locale *vertical-composite* =
A: *category A* +
B: *category B* +
F: *functor A B F* +
G: *functor A B G* +
H: *functor A B H* +
σ: *natural-transformation A B F G* σ +
τ: *natural-transformation A B G H* τ
for *A* :: *'a comp* (**infixr** $\langle\cdot_A\rangle$ *55*)
and *B* :: *'b comp* (**infixr** $\langle\cdot_B\rangle$ *55*)
and *F* :: *'a* $\Rightarrow$ *'b*
and *G* :: *'a* $\Rightarrow$ *'b*
and *H* :: *'a* $\Rightarrow$ *'b*
and σ :: *'a* $\Rightarrow$ *'b*
and τ :: *'a* $\Rightarrow$ *'b*
begin

Vertical composition takes an arrow «$a : b \to_A f$» to an arrow in *B.hom* (*F a*) (*G b*), which we can obtain by forming either of the composites $\tau\ b \cdot_B \sigma\ f$ or $\tau\ f \cdot_B \sigma\ a$, which are equal to each other.

definition *map*
where *map f* = (*if A.arr f then* τ (*A.cod f*) $\cdot_B$ σ *f else B.null*)

lemma *map-seq*:
assumes *A.arr f*
shows *B.seq* (τ (*A.cod f*)) (σ *f*)
using *assms* **by** *auto*

lemma *map-simp-ide*:
assumes *A.ide a*
shows *map a* = τ *a* $\cdot_B$ σ *a*
using *assms map-def* **by** *auto*

lemma *map-simp-1*:
assumes *A.arr f*
shows *map f* = τ (*A.cod f*) $\cdot_B$ σ *f*
using *assms* **by** (*simp add*: *map-def*)

lemma *map-simp-2*:
assumes *A.arr f*
shows *map f* = τ *f* $\cdot_B$ σ (*A.dom f*)
using *assms*
by (*metis B.comp-assoc* σ*.naturality2* σ*.naturality* τ*.naturality1* τ*.naturality map-simp-1*)

lemma *is-natural-transformation*:
shows *natural-transformation A B F H map*
using *map-def map-simp-1 map-simp-2 map-seq B.comp-assoc*
apply (*unfold-locales*, *simp-all*)
by (*metis B.comp-assoc* τ*.naturality1*)

end

sublocale *vertical-composite* $\subseteq$ *natural-transformation A B F H map*
using *is-natural-transformation* **by** *auto*

Functors are the identities for vertical composition.

lemma *vcomp-ide-dom* [*simp*]:
assumes *natural-transformation A B F G* τ
shows *vertical-composite.map A B F* τ = τ
proof (*intro natural-transformation-eqI*)
interpret τ: *natural-transformation A B F G* τ
using *assms* **by** *blast*
interpret τ*oF*: *vertical-composite A B F F G F* τ **..**
show *natural-transformation A B F G* τ **..**
show *natural-transformation A B F G* (*vertical-composite.map A B F* τ) **..**
show $\bigwedge$*a*. τ*.A.ide a* $\Longrightarrow$ τ*oF.map a* = τ *a*
using τ*oF.map-def* τ*oF.extensionality* τ*.extensionality* τ*.naturality2* **by** *metis*
qed

lemma *vcomp-ide-cod* [*simp*]:
assumes *natural-transformation A B F G τ*
shows *vertical-composite.map A B τ G = τ*
proof (*intro natural-transformation-eqI*)
  interpret *τ*: *natural-transformation A B F G τ*
    using *assms* **by** *blast*
  interpret *Goτ*: *vertical-composite A B F G G τ G* **..**
  show *natural-transformation A B F G τ* **..**
  show *natural-transformation A B F G* (*vertical-composite.map A B τ G*) **..**
  show ⋀*a*. *τ.A.ide a* ⟹ *Goτ.map a* = *τ a*
    using *Goτ.map-def Goτ.extensionality τ.extensionality*
    by (*metis Goτ.map-simp-ide τ.A.comp-ide-self τ.preserves-comp-1*)
qed

Vertical composition is associative.

lemma *vcomp-assoc* [*simp*]:
assumes *natural-transformation A B F G ϱ*
and *natural-transformation A B G H σ*
and *natural-transformation A B H K τ*
shows *vertical-composite.map A B* (*vertical-composite.map A B ϱ σ*) *τ*
        = *vertical-composite.map A B ϱ* (*vertical-composite.map A B σ τ*)
proof −
  interpret *A*: *category A*
    using *assms*(*1*) *natural-transformation-def functor-def* **by** *blast*
  interpret *B*: *category B*
    using *assms*(*1*) *natural-transformation-def functor-def* **by** *blast*
  interpret *ϱ*: *natural-transformation A B F G ϱ* **using** *assms*(*1*) **by** *auto*
  interpret *σ*: *natural-transformation A B G H σ* **using** *assms*(*2*) **by** *auto*
  interpret *τ*: *natural-transformation A B H K τ* **using** *assms*(*3*) **by** *auto*
  interpret *ϱσ*: *vertical-composite A B F G H ϱ σ* **..**
  interpret *στ*: *vertical-composite A B G H K σ τ* **..**
  interpret *ϱ-στ*: *vertical-composite A B F G K ϱ στ.map* **..**
  interpret *ϱσ-τ*: *vertical-composite A B F H K ϱσ.map τ* **..**
  show *?thesis*
    using *ϱσ-τ.is-natural-transformation ϱ-στ.natural-transformation-axioms*
      ϱσ.map-simp-ide ϱσ-τ.map-simp-ide ϱ-στ.map-simp-ide στ.map-simp-ide B.comp-assoc
    by (*intro natural-transformation-eqI*, *auto*)
qed

12.6 Natural Isomorphisms

A natural isomorphism is a natural transformation each of whose components is an isomorphism. Equivalently, a natural isomorphism is a natural transformation that is invertible with respect to vertical composition.

locale *natural-isomorphism* = *natural-transformation A B F G τ*
for *A* :: *'a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_B$› *55*)
and *F* :: *'a* ⇒ *'b*

and $G :: 'a \Rightarrow 'b$
and $\tau :: 'a \Rightarrow 'b +$
assumes *components-are-iso* [*simp*]: $A.ide\ a \Longrightarrow B.iso\ (\tau\ a)$
begin

lemma *inv-naturality*:
assumes $A.arr\ f$
shows $F\ f\ \cdot_B\ B.inv\ (\tau\ (A.dom\ f)) = B.inv\ (\tau\ (A.cod\ f))\ \cdot_B\ G\ f$
using *assms naturality1 naturality2 components-are-iso B.invert-side-of-triangle B.comp-assoc A.ide-cod A.ide-dom preserves-reflects-arr*
by *fastforce*

Natural isomorphisms preserve isomorphisms, in the sense that the sides of of the naturality square determined by an isomorphism are all isomorphisms, so the diagonal is, as well.

lemma *preserves-iso*:
assumes $A.iso\ f$
shows $B.iso\ (\tau\ f)$
using *assms*
by (*metis A.ide-dom A.iso-is-arr B.isos-compose G.preserves-iso components-are-iso naturality2 naturality preserves-reflects-arr*)

end

Since the function that represents a functor is formally identical to the function that represents the corresponding identity natural transformation, no additional locale is needed for identity natural transformations. However, an identity natural transformation is also a natural isomorphism, so it is useful for *functor* to inherit from the *natural-isomorphism* locale.

sublocale *functor* $\subseteq$ *as-nat-iso*: *natural-isomorphism A B F F F*
apply *unfold-locales*
using *preserves-ide B.ide-is-iso* **by** *simp*

definition *naturally-isomorphic*
where *naturally-isomorphic A B F G* = $(\exists \tau.$ *natural-isomorphism A B F G* $\tau)$

lemma *naturally-isomorphic-respects-full-functor*:
assumes *naturally-isomorphic A B F G*
and *full-functor A B F*
shows *full-functor A B G*
proof −
obtain φ **where** φ: *natural-isomorphism A B F G* φ
using *assms naturally-isomorphic-def* **by** *blast*
interpret φ: *natural-isomorphism A B F G* φ
using φ **by** *auto*
interpret $\varphi.F$: *full-functor A B F*
using *assms* **by** *auto*
write *A* (**infixr** $\langle\cdot_A\rangle$ *55*)
write *B* (**infixr** $\langle\cdot_B\rangle$ *55*)

```
  write φ.A.in-hom (‹«- : - →_A -»›)
  write φ.B.in-hom (‹«- : - →_B -»›)
  show full-functor A B G
  proof
    fix a a′ g
    assume a′: φ.A.ide a′ and a: φ.A.ide a
    and g: «g : G a′ →_B G a»
    show ∃f. «f : a′ →_A a» ∧ G f = g
    proof −
      let ?g′ = φ.B.inv (φ a) ·_B g ·_B φ a′
      have g′: «?g′ : F a′ →_B F a»
        using a a′ g φ.preserves-hom φ.components-are-iso φ.B.inv-in-hom by force
      obtain f′ where f′: «f′ : a′ →_A a» ∧ F f′ = ?g′
        using a a′ g′ φ.F.is-full [of a a′ ?g′] by blast
      moreover have G f′ = g
        by (metis f′ φ.A.arrI φ.B.arrI φ.B.inv-inv φ.B.invert-side-of-triangle(1−2)
            φ.B.iso-inv-iso φ.G.as-nat-trans.natural-transformation-axioms
            φ.components-are-iso φ.naturality a a′ category.in-homE f′ g′
            natural-transformation.axioms(1))
      ultimately show ?thesis by auto
    qed
  qed
qed

lemma naturally-isomorphic-respects-faithful-functor:
assumes naturally-isomorphic A B F G
and faithful-functor A B F
shows faithful-functor A B G
proof −
  obtain φ where φ: natural-isomorphism A B F G φ
    using assms naturally-isomorphic-def by blast
  interpret φ: natural-isomorphism A B F G φ
    using φ by auto
  interpret φ.F: faithful-functor A B F
    using assms by auto
  show faithful-functor A B G
    using φ.naturality1 φ.components-are-iso φ.B.iso-is-section φ.B.section-is-mono
          φ.B.mono-cancel φ.F.is-faithful φ.naturality
          φ.natural-transformation-axioms φ.preserves-reflects-arr φ.A.ide-cod
    by (unfold-locales, metis)
qed

locale inverse-transformation =
  A: category A +
  B: category B +
  F: functor A B F +
  G: functor A B G +
  τ: natural-isomorphism A B F G τ
for A :: 'a comp      (infixr ‹·_A› 55)
```

and $B :: 'b\ comp$ (**infixr** ‹$\cdot_B$› 55)
and $F :: 'a \Rightarrow 'b$
and $G :: 'a \Rightarrow 'b$
and $\tau :: 'a \Rightarrow 'b$
begin

interpretation τ': *transformation-by-components A B G F* ‹$\lambda a.\ B.inv\ (\tau\ a)$›
proof
fix $f :: 'a$
show *A.ide f* $\Longrightarrow$ «*B.inv* (τ *f*) : *G f* $\rightarrow_B$ *F f*»
using *B.inv-in-hom* τ*.components-are-iso A.ide-in-hom* **by** *blast*
show *A.arr f* $\Longrightarrow$ *B.inv* (τ (*A.cod f*)) $\cdot_B$ *G f* = *F f* $\cdot_B$ *B.inv* (τ (*A.dom f*))
by (*metis A.ide-cod A.ide-dom B.invert-opposite-sides-of-square* τ*.components-are-iso*
τ*.naturality2* τ*.naturality* τ*.preserves-reflects-arr*)
qed

definition *map*
where *map* = τ'*.map*

lemma *map-ide-simp* [*simp*]:
assumes *A.ide a*
shows *map a* = *B.inv* (τ *a*)
using *assms map-def* **by** *fastforce*

lemma *map-simp*:
assumes *A.arr f*
shows *map f* = *B.inv* (τ (*A.cod f*)) $\cdot_B$ *G f*
using *assms map-def* **by** (*simp add*: τ'*.map-def*)

lemma *is-natural-transformation*:
shows *natural-transformation A B G F map*
by (*simp add*: τ'*.natural-transformation-axioms map-def*)

lemma *inverts-components*:
assumes *A.ide a*
shows *B.inverse-arrows* (τ *a*) (*map a*)
using *assms* τ*.components-are-iso B.ide-is-iso B.inv-is-inverse B.inverse-arrows-def map-def*
by (*metis* τ'*.map-simp-ide*)

end

sublocale *inverse-transformation* $\subseteq$ *natural-transformation A B G F map*
using *is-natural-transformation* **by** *auto*

sublocale *inverse-transformation* $\subseteq$ *natural-isomorphism A B G F map*
by (*simp add*: *natural-isomorphism.intro natural-isomorphism-axioms.intro*
natural-transformation-axioms)

lemma *inverse-inverse-transformation* [*simp*]:

assumes *natural-isomorphism A B F G τ*
shows *inverse-transformation.map A B F* (*inverse-transformation.map A B G τ*) = *τ*
proof −
  interpret *τ*: *natural-isomorphism A B F G τ*
    using *assms* **by** *auto*
  interpret *τ′*: *inverse-transformation A B F G τ* **..**
  interpret *τ″*: *inverse-transformation A B G F τ′.map* **..**
  show *τ″.map* = *τ*
    using *τ.natural-transformation-axioms τ″.natural-transformation-axioms*
    by (*intro natural-transformation-eqI*, *auto*)
qed

locale *inverse-transformations* =
  A: *category A* +
  B: *category B* +
  F: *functor A B F* +
  G: *functor A B G* +
  τ: *natural-transformation A B F G τ* +
  τ′: *natural-transformation A B G F τ′*
for *A* :: *′a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *′b comp* (**infixr** ‹·$_B$› *55*)
and *F* :: *′a* ⇒ *′b*
and *G* :: *′a* ⇒ *′b*
and *τ* :: *′a* ⇒ *′b*
and *τ′* :: *′a* ⇒ *′b* +
assumes *inv*: *A.ide a* ⟹ *B.inverse-arrows* (*τ a*) (*τ′ a*)

sublocale *inverse-transformations* ⊆ *natural-isomorphism A B F G τ*
  by (*meson B.category-axioms τ.natural-transformation-axioms B.iso-def inv*
      natural-isomorphism.intro natural-isomorphism-axioms.intro)
sublocale *inverse-transformations* ⊆ *natural-isomorphism A B G F τ′*
  by (*meson category.inverse-arrows-sym category.iso-def inverse-transformations-axioms*
      inverse-transformations-axioms-def inverse-transformations-def
      natural-isomorphism.intro natural-isomorphism-axioms.intro)

lemma *inverse-transformations-sym*:
assumes *inverse-transformations A B F G σ σ′*
shows *inverse-transformations A B G F σ′ σ*
  using *assms*
  by (*simp add*: *category.inverse-arrows-sym inverse-transformations-axioms-def*
      inverse-transformations-def)

lemma *inverse-transformations-inverse*:
assumes *inverse-transformations A B F G σ σ′*
shows *vertical-composite.map A B σ σ′* = *F*
and *vertical-composite.map A B σ′ σ* = *G*
proof −
  interpret *A*: *category A*
    using *assms*(*1*) *inverse-transformations-def natural-transformation-def* **by** *blast*

```
  interpret inv: inverse-transformations A B F G σ σ' using assms by auto
  interpret σσ': vertical-composite A B F G F σ σ' ..
  show vertical-composite.map A B σ σ' = F
    using σσ'.is-natural-transformation inv.F.as-nat-trans.natural-transformation-axioms
          σσ'.map-simp-ide inv.B.comp-inv-arr inv.inv
    by (intro natural-transformation-eqI, simp-all)
  interpret inv': inverse-transformations A B G F σ' σ
    using assms inverse-transformations-sym by blast
  interpret σ'σ: vertical-composite A B G F G σ' σ ..
  show vertical-composite.map A B σ' σ = G
    using σ'σ.is-natural-transformation inv.G.as-nat-trans.natural-transformation-axioms
          σ'σ.map-simp-ide inv'.inv inv.B.comp-inv-arr
    by (intro natural-transformation-eqI, simp-all)
qed

lemma inverse-transformations-compose:
assumes inverse-transformations A B F G σ σ'
and inverse-transformations A B G H τ τ'
shows inverse-transformations A B F H
        (vertical-composite.map A B σ τ) (vertical-composite.map A B τ' σ')
proof −
  interpret A: category A using assms(1) inverse-transformations-def by blast
  interpret B: category B using assms(1) inverse-transformations-def by blast
  interpret σσ': inverse-transformations A B F G σ σ' using assms(1) by auto
  interpret ττ': inverse-transformations A B G H τ τ' using assms(2) by auto
  interpret στ: vertical-composite A B F G H σ τ ..
  interpret τ'σ': vertical-composite A B H G F τ' σ' ..
  show ?thesis
    using B.inverse-arrows-compose σσ'.inv στ.map-simp-ide τ'σ'.map-simp-ide ττ'.inv
    by (unfold-locales, auto)
qed

lemma vertical-composite-iso-inverse [simp]:
assumes natural-isomorphism A B F G τ
shows vertical-composite.map A B τ (inverse-transformation.map A B G τ) = F
proof −
  interpret τ: natural-isomorphism A B F G τ using assms by auto
  interpret τ': inverse-transformation A B F G τ ..
  interpret ττ': vertical-composite A B F G F τ τ'.map ..
  show ?thesis
    using ττ'.is-natural-transformation τ.F.as-nat-trans.natural-transformation-axioms
          τ'.inverts-components τ.B.comp-inv-arr ττ'.map-simp-ide
    by (intro natural-transformation-eqI, auto)
qed

lemma vertical-composite-inverse-iso [simp]:
assumes natural-isomorphism A B F G τ
shows vertical-composite.map A B (inverse-transformation.map A B G τ) τ = G
proof −
```

interpret τ: *natural-isomorphism A B F G* τ **using** *assms* **by** *auto*
interpret τ': *inverse-transformation A B F G* τ **..**
interpret $\tau'\tau$: *vertical-composite A B G F G* τ'*.map* τ **..**
show *?thesis*
using $\tau'\tau$*.is-natural-transformation* τ*.G.as-nat-trans.natural-transformation-axioms*
τ'*.inverts-components* $\tau'\tau$*.map-simp-ide* τ*.B.comp-arr-inv*
by (*intro natural-transformation-eqI*, *auto*)
qed

lemma *natural-isomorphisms-compose*:
assumes *natural-isomorphism A B F G* σ **and** *natural-isomorphism A B G H* τ
shows *natural-isomorphism A B F H* (*vertical-composite.map A B* σ τ)
proof −
interpret *A*: *category A*
using *assms*(*1*) *natural-isomorphism-def natural-transformation-def* **by** *blast*
interpret *B*: *category B*
using *assms*(*1*) *natural-isomorphism-def natural-transformation-def* **by** *blast*
interpret σ: *natural-isomorphism A B F G* σ **using** *assms*(*1*) **by** *auto*
interpret τ: *natural-isomorphism A B G H* τ **using** *assms*(*2*) **by** *auto*
interpret $\sigma\tau$: *vertical-composite A B F G H* σ τ **..**
interpret *natural-isomorphism A B F H* $\sigma\tau$*.map*
using $\sigma\tau$*.map-simp-ide* **by** (*unfold-locales*, *auto*)
show *?thesis* **..**
qed

lemma *naturally-isomorphic-reflexive*:
assumes *functor A B F*
shows *naturally-isomorphic A B F F*
proof −
interpret *F*: *functor A B F* **using** *assms* **by** *auto*
have *natural-isomorphism A B F F F* **..**
thus *?thesis* **using** *naturally-isomorphic-def* **by** *blast*
qed

lemma *naturally-isomorphic-symmetric*:
assumes *naturally-isomorphic A B F G*
shows *naturally-isomorphic A B G F*
proof −
obtain φ **where** φ: *natural-isomorphism A B F G* φ
using *assms naturally-isomorphic-def* **by** *blast*
interpret φ: *natural-isomorphism A B F G* φ
using φ **by** *auto*
interpret ψ: *inverse-transformation A B F G* φ **..**
have *natural-isomorphism A B G F* ψ*.map* **..**
thus *?thesis* **using** *naturally-isomorphic-def* **by** *blast*
qed

lemma *naturally-isomorphic-transitive* [*trans*]:
assumes *naturally-isomorphic A B F G*

and *naturally-isomorphic A B G H*
shows *naturally-isomorphic A B F H*
proof −
obtain φ **where** φ: *natural-isomorphism A B F G* φ
using *assms naturally-isomorphic-def* **by** *blast*
interpret φ: *natural-isomorphism A B F G* φ
using φ **by** *auto*
obtain ψ **where** ψ: *natural-isomorphism A B G H* ψ
using *assms naturally-isomorphic-def* **by** *blast*
interpret ψ: *natural-isomorphism A B G H* ψ
using ψ **by** *auto*
interpret $\psi\varphi$: *vertical-composite A B F G H* φ ψ **..**
have *natural-isomorphism A B F H* $\psi\varphi$*.map*
using φ ψ *natural-isomorphisms-compose* **by** *blast*
thus *?thesis*
using *naturally-isomorphic-def* **by** *blast*
qed

12.7 Horizontal Composition

Horizontal composition is a way of composing parallel natural transformations σ from F to G and τ from H to K, where functors F and G map A to B and H and K map B to C, to obtain a natural transformation from $H \circ F$ to $K \circ G$.

Since horizontal composition turns out to coincide with ordinary composition of natural transformations as functions, there is little point in defining a cumbersome locale for horizontal composite.

lemma *horizontal-composite*:
assumes *natural-transformation A B F G* σ
and *natural-transformation B C H K* τ
shows *natural-transformation A C* (*H o F*) (*K o G*) (τ *o* σ)
proof −
interpret σ: *natural-transformation A B F G* σ
using *assms*(*1*) **by** *simp*
interpret τ: *natural-transformation B C H K* τ
using *assms*(*2*) **by** *simp*
interpret *HF*: *composite-functor A B C F H* **..**
interpret *KG*: *composite-functor A B C G K* **..**
show *natural-transformation A C* (*H o F*) (*K o G*) (τ *o* σ)
using σ*.extensionality* τ*.extensionality*
apply (*unfold-locales*, *auto*)
apply (*metis* σ*.naturality1* σ*.preserves-reflects-arr* τ*.preserves-comp-1*)
by (*metis* σ*.naturality2* σ*.preserves-reflects-arr* τ*.preserves-comp-2*)
qed

lemma *hcomp-ide-dom* [*simp*]:
assumes *natural-transformation A B F G* τ
shows τ *o* (*identity-functor.map A*) = τ
proof −

```
  interpret τ: natural-transformation A B F G τ using assms by auto
  show τ o τ.A.map = τ
    using τ.A.map-def τ.extensionality by fastforce
qed

lemma hcomp-ide-cod [simp]:
assumes natural-transformation A B F G τ
shows (identity-functor.map B) o τ = τ
proof −
  interpret τ: natural-transformation A B F G τ using assms by auto
  show τ.B.map o τ = τ
    using τ.B.map-def τ.extensionality by auto
qed
```

Horizontal composition of a functor with a vertical composite.

```
lemma whisker-right:
assumes functor A B F
and natural-transformation B C H K τ and natural-transformation B C K L τ'
shows (vertical-composite.map B C τ τ') o F = vertical-composite.map A C (τ o F) (τ' o F)
proof −
  interpret F: functor A B F using assms(1) by auto
  interpret τ: natural-transformation B C H K τ using assms(2) by auto
  interpret τ': natural-transformation B C K L τ' using assms(3) by auto
  interpret τoF: natural-transformation A C ‹H o F› ‹K o F› ‹τ o F›
    using τ.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
          horizontal-composite
    by blast
  interpret τ'oF: natural-transformation A C ‹K o F› ‹L o F› ‹τ' o F›
    using τ'.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
          horizontal-composite
    by blast
  interpret τ'τ: vertical-composite B C H K L τ τ' ..
  interpret τ'τoF: natural-transformation A C ‹H o F› ‹L o F› ‹τ'τ.map o F›
    using τ'τ.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
          horizontal-composite
    by blast
  interpret τ'oF-τoF: vertical-composite A C ‹H o F› ‹K o F› ‹L o F› ‹τ o F› ‹τ' o F› ..
  show ?thesis
    using τ'oF-τoF.map-def τ'τ.map-def τ'τoF.extensionality by auto
qed
```

Horizontal composition of a vertical composite with a functor.

```
lemma whisker-left:
assumes functor B C K
and natural-transformation A B F G τ and natural-transformation A B G H τ'
shows K o (vertical-composite.map A B τ τ') = vertical-composite.map A C (K o τ) (K o τ')
proof −
  interpret K: functor B C K using assms(1) by auto
  interpret τ: natural-transformation A B F G τ using assms(2) by auto
```

interpret τ': *natural-transformation A B G H* τ' **using** *assms*(*3*) **by** *auto*
interpret $\tau'\tau$: *vertical-composite A B F G H* τ τ' **..**
interpret $K o \tau$: *natural-transformation A C* ‹*K o F*› ‹*K o G*› ‹*K o* τ›
using τ.*natural-transformation-axioms K.as-nat-trans.natural-transformation-axioms horizontal-composite*
by *blast*
interpret $K o \tau'$: *natural-transformation A C* ‹*K o G*› ‹*K o H*› ‹*K o* τ'›
using τ'.*natural-transformation-axioms K.as-nat-trans.natural-transformation-axioms horizontal-composite*
by *blast*
interpret $K o \tau'\tau$: *natural-transformation A C* ‹*K o F*› ‹*K o H*› ‹*K o* $\tau'\tau$.*map*›
using $\tau'\tau$.*natural-transformation-axioms K.as-nat-trans.natural-transformation-axioms horizontal-composite*
by *blast*
interpret $K o \tau'$-$K o \tau$: *vertical-composite A C* ‹*K o F*› ‹*K o G*› ‹*K o H*› ‹*K o* τ› ‹*K o* τ'› **..**
show *K o* $\tau'\tau$.*map* = $K o \tau'$-$K o \tau$.*map*
using $K o \tau'$-$K o \tau$.*map-def* $\tau'\tau$.*map-def* $K o \tau'\tau$.*extensionality* $K o \tau'$-$K o \tau$.*map-simp-1* $\tau'\tau$.*map-simp-1*
by *auto*
qed

The interchange law for horizontal and vertical composition.

lemma *interchange*:
assumes *natural-transformation B C F G* τ **and** *natural-transformation B C G H* ν
and *natural-transformation C D K L* σ **and** *natural-transformation C D L M* μ
shows *vertical-composite.map C D* σ μ ∘ *vertical-composite.map B C* τ ν =
vertical-composite.map B D (σ ∘ τ) (μ ∘ ν)
proof −
interpret τ: *natural-transformation B C F G* τ
using *assms*(*1*) **by** *auto*
interpret ν: *natural-transformation B C G H* ν
using *assms*(*2*) **by** *auto*
interpret σ: *natural-transformation C D K L* σ
using *assms*(*3*) **by** *auto*
interpret μ: *natural-transformation C D L M* μ
using *assms*(*4*) **by** *auto*
interpret $\nu\tau$: *vertical-composite B C F G H* τ ν **..**
interpret $\mu\sigma$: *vertical-composite C D K L M* σ μ **..**
interpret $\sigma o \tau$: *natural-transformation B D* ‹*K o F*› ‹*L o G*› ‹σ *o* τ›
using σ.*natural-transformation-axioms* τ.*natural-transformation-axioms horizontal-composite*
by *blast*
interpret $\mu o \nu$: *natural-transformation B D* ‹*L o G*› ‹*M o H*› ‹μ *o* ν›
using μ.*natural-transformation-axioms* ν.*natural-transformation-axioms horizontal-composite*
by *blast*
interpret $\mu\sigma o \nu\tau$: *natural-transformation B D* ‹*K o F*› ‹*M o H*› ‹$\mu\sigma$.*map o* $\nu\tau$.*map*›
using $\mu\sigma$.*natural-transformation-axioms* $\nu\tau$.*natural-transformation-axioms horizontal-composite*
by *blast*

```
  interpret μoν-σoτ: vertical-composite B D ‹K o F› ‹L o G› ‹M o H› ‹σ o τ› ‹μ o ν› ..
  show μσ.map o ντ.map = μoν-σoτ.map
  proof (intro natural-transformation-eqI)
    show natural-transformation B D (K ∘ F) (M ∘ H) (μσ.map o ντ.map) ..
    show natural-transformation B D (K ∘ F) (M ∘ H) μoν-σoτ.map ..
    show ⋀a. τ.A.ide a ⟹ (μσ.map o ντ.map) a = μoν-σoτ.map a
    proof −
      fix a
      assume a: τ.A.ide a
      have (μσ.map o ντ.map) a = D (μ (H a)) (σ (C (ν a) (τ a)))
        using a μσ.map-simp-1 ντ.map-simp-2 by simp
      also have ... = D (μ (ν a)) (σ (τ a))
        using a
        by (metis (full-types) μ.naturality1 μσ.map-simp-1 μσ.preserves-comp-1
            ντ.map-seq ντ.map-simp-1 ντ.preserves-cod σ.B.comp-assoc τ.A.ide-char τ.B.seqE)
      also have ... = μoν-σoτ.map a
        using a μoν-σoτ.map-simp-ide by simp
      finally show (μσ.map o ντ.map) a = μoν-σoτ.map a by blast
    qed
  qed
qed
```

A special-case of the interchange law in which two of the natural transformations are functors. It comes up reasonably often, and the reasoning is awkward.

```
lemma interchange-spc:
assumes natural-transformation B C F G σ
and natural-transformation C D H K τ
shows τ ∘ σ = vertical-composite.map B D (H o σ) (τ o G)
and τ ∘ σ = vertical-composite.map B D (τ o F) (K o σ)
proof −
  show τ ∘ σ = vertical-composite.map B D (H ∘ σ) (τ ∘ G)
  proof −
    have vertical-composite.map C D H τ ∘ vertical-composite.map B C σ G =
          vertical-composite.map B D (H ∘ σ) (τ ∘ G)
    by (meson assms functor-is-transformation interchange natural-transformation.axioms(3−4))
    thus ?thesis
      using assms by force
  qed
  show τ ∘ σ = vertical-composite.map B D (τ ∘ F) (K ∘ σ)
  proof −
    have vertical-composite.map C D τ K ∘ vertical-composite.map B C F σ =
          vertical-composite.map B D (τ ∘ F) (K ∘ σ)
    by (meson assms functor-is-transformation interchange natural-transformation.axioms(3−4))
    thus ?thesis
      using assms by force
  qed
qed

end
```

Chapter 13

BinaryFunctor

theory *BinaryFunctor*
imports *ProductCategory NaturalTransformation*
begin

This theory develops various properties of binary functors, which are functors defined on product categories.

locale *binary-functor* =
A1: *category A1* +
A2: *category A2* +
B: *category B* +
A1xA2: *product-category A1 A2* +
functor A1xA2.comp B F
for *A1* :: *'a1 comp* (**infixr** ‹·$_{A1}$› *55*)
and *A2* :: *'a2 comp* (**infixr** ‹·$_{A2}$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_{B}$› *55*)
and *F* :: *'a1 ∗ 'a2 ⇒ 'b*
begin

notation *A1.in-hom* (‹«- : - →$_{A1}$ -»›)
notation *A2.in-hom* (‹«- : - →$_{A2}$ -»›)

end

A product functor is a binary functor obtained by placing two functors in parallel.

locale *product-functor* =
A1: *category A1* +
A2: *category A2* +
B1: *category B1* +
B2: *category B2* +
F1: *functor A1 B1 F1* +
F2: *functor A2 B2 F2* +
A1xA2: *product-category A1 A2* +
B1xB2: *product-category B1 B2*
for *A1* :: *'a1 comp* (**infixr** ‹·$_{A1}$› *55*)
and *A2* :: *'a2 comp* (**infixr** ‹·$_{A2}$› *55*)

```
and B1 :: 'b1 comp      (infixr ‹·B1› 55)
and B2 :: 'b2 comp      (infixr ‹·B2› 55)
and F1 :: 'a1 ⇒ 'b1
and F2 :: 'a2 ⇒ 'b2
begin

  notation A1xA2.comp    (infixr ‹·A1xA2› 55)
  notation B1xB2.comp    (infixr ‹·B1xB2› 55)
  notation A1.in-hom     (‹«- : - →A1 -»›)
  notation A2.in-hom     (‹«- : - →A2 -»›)
  notation B1.in-hom     (‹«- : - →B1 -»›)
  notation B2.in-hom     (‹«- : - →B2 -»›)
  notation A1xA2.in-hom  (‹«- : - →A1xA2 -»›)
  notation B1xB2.in-hom  (‹«- : - →B1xB2 -»›)

  definition map
  where map f = (if A1.arr (fst f) ∧ A2.arr (snd f)
                 then (F1 (fst f), F2 (snd f)) else (F1 A1.null, F2 A2.null))

  lemma map-simp [simp]:
  assumes A1xA2.arr f
  shows map f = (F1 (fst f), F2 (snd f))
    using assms map-def by simp

  lemma is-functor:
  shows functor A1xA2.comp B1xB2.comp map
    using B1xB2.dom-char B1xB2.cod-char F1.extensionality F2.extensionality
    apply (unfold-locales)
    using map-def A1.arr-dom-iff-arr A1.arr-cod-iff-arr A2.arr-dom-iff-arr A2.arr-cod-iff-arr
       apply auto[4]
    using A1xA2.seqE map-simp by fastforce

end

sublocale product-functor ⊆ functor A1xA2.comp B1xB2.comp map
  using is-functor by auto
sublocale product-functor ⊆ binary-functor A1 A2 B1xB2.comp map ..
```

The following locale is concerned with a binary functor from a category to itself. It defines related functors that are useful when considering monoidal structure on a category.

```
locale binary-endofunctor =
  C: category C +
  CC: product-category C C +
  CCC: product-category C CC.comp +
  binary-functor C C C T
for C :: 'a comp      (infixr ‹·› 55)
and T :: 'a * 'a ⇒ 'a
begin
```

definition *ToTC*
where *ToTC f* ≡ *if CCC.arr f then T* (*T* (*fst f*, *fst* (*snd f*)), *snd* (*snd f*)) *else C.null*

lemma *functor-ToTC*:
shows *functor CCC.comp C ToTC*
using *ToTC-def* **apply** *unfold-locales*
apply *auto*[4]
proof −
fix *f g*
assume *gf*: *CCC.seq g f*
show *ToTC* (*CCC.comp g f*) = *ToTC g* · *ToTC f*
using *gf* **unfolding** *CCC.seq-char CC.seq-char ToTC-def*
apply *auto*
by (*metis CC.comp-simp CC.seqI*$_{PC}$ *fst-conv preserves-comp preserves-seq snd-conv*)
qed

lemma *ToTC-simp* [*simp*]:
assumes *C.arr f* **and** *C.arr g* **and** *C.arr h*
shows *ToTC* (*f*, *g*, *h*) = *T* (*T* (*f*, *g*), *h*)
using *assms ToTC-def CCC.arr-char* **by** *simp*

definition *ToCT*
where *ToCT f* ≡ *if CCC.arr f then T* (*fst f*, *T* (*fst* (*snd f*), *snd* (*snd f*))) *else C.null*

lemma *functor-ToCT*:
shows *functor CCC.comp C ToCT*
using *ToCT-def* **apply** *unfold-locales*
apply *auto*[4]
proof −
fix *f g*
assume *gf*: *CCC.seq g f*
show *ToCT* (*CCC.comp g f*) = *ToCT g* · *ToCT f*
using *gf* **unfolding** *CCC.seq-char CC.seq-char ToCT-def*
apply *auto*
by (*metis CC.comp-simp CC.seq-char as-nat-trans.preserves-comp-2 fst-conv*
preserves-reflects-arr snd-conv)
qed

lemma *ToCT-simp* [*simp*]:
assumes *C.arr f* **and** *C.arr g* **and** *C.arr h*
shows *ToCT* (*f*, *g*, *h*) = *T* (*f*, *T* (*g*, *h*))
using *assms ToCT-def CCC.arr-char* **by** *simp*

end

A symmetry functor is a binary functor that exchanges its two arguments.

locale *symmetry-functor* =
A1: *category A1* +

```
A2: category A2 +
A1xA2: product-category A1 A2 +
A2xA1: product-category A2 A1
for A1 :: 'a1 comp      (infixr ‹·A1› 55)
and A2 :: 'a2 comp      (infixr ‹·A2› 55)
begin

  notation A1xA2.comp    (infixr ‹·A1xA2› 55)
  notation A2xA1.comp    (infixr ‹·A2xA1› 55)
  notation A1xA2.in-hom  (‹«- : - →A1xA2 -»›)
  notation A2xA1.in-hom  (‹«- : - →A2xA1 -»›)

  definition map :: 'a1 * 'a2 ⇒ 'a2 * 'a1
  where map f = (if A1xA2.arr f then (snd f, fst f) else A2xA1.null)

  lemma map-simp [simp]:
  assumes A1xA2.arr f
  shows map f = (snd f, fst f)
    using assms map-def by meson

  lemma is-functor:
  shows functor A1xA2.comp A2xA1.comp map
    using map-def A1.arr-dom-iff-arr A1.arr-cod-iff-arr A2.arr-dom-iff-arr A2.arr-cod-iff-arr
    apply (unfold-locales)
       apply auto[4]
    by force

end

sublocale symmetry-functor ⊆ functor A1xA2.comp A2xA1.comp map
  using is-functor by auto
sublocale symmetry-functor ⊆ binary-functor A1 A2 A2xA1.comp map ..

context binary-functor
begin

  abbreviation sym
  where sym ≡ (λf. F (snd f, fst f))

  lemma sym-is-binary-functor:
  shows binary-functor A2 A1 B sym
  proof −
    interpret A2xA1: product-category A2 A1 ..
    interpret S: symmetry-functor A2 A1 ..
    interpret SF: composite-functor A2xA1.comp A1xA2.comp B S.map F ..
    have binary-functor A2 A1 B (F o S.map) ..
    moreover have F o S.map = (λf. F (snd f, fst f))
      using extensionality SF.extensionality S.map-def by fastforce
    ultimately show ?thesis using sym-def by auto
```

qed

Fixing one or the other argument of a binary functor to be an identity yields a functor of the other argument.

lemma *fixing-ide-gives-functor-1*:
assumes *A1.ide a1*
shows *functor A2 B* (λ*f2*. *F* (*a1*, *f2*))
using *assms*
apply *unfold-locales*
using *extensionality*
apply *auto*[4]
by (*metis A1.ideD*(1) *A1.comp-ide-self A1xA2.comp-simp A1xA2.seq-char fst-conv as-nat-trans.preserves-comp-2 snd-conv*)

lemma *fixing-ide-gives-functor-2*:
assumes *A2.ide a2*
shows *functor A1 B* (λ*f1*. *F* (*f1*, *a2*))
using *assms*
apply (*unfold-locales*)
using *extensionality*
apply *auto*[4]
by (*metis A1xA2.comp-simp A1xA2.seq-char A2.ideD*(1) *A2.comp-ide-self fst-conv as-nat-trans.preserves-comp-2 snd-conv*)

Fixing one or the other argument of a binary functor to be an arrow yields a natural transformation.

lemma *fixing-arr-gives-natural-transformation-1*:
assumes *A1.arr f1*
shows *natural-transformation A2 B* (λ*f2*. *F* (*A1.dom f1*, *f2*)) (λ*f2*. *F* (*A1.cod f1*, *f2*))
(λ*f2*. *F* (*f1*, *f2*))
proof −
let *?Fdom* = λ*f2*. *F* (*A1.dom f1*, *f2*)
interpret *Fdom*: *functor A2 B ?Fdom* **using** *assms fixing-ide-gives-functor-1* **by** *auto*
let *?Fcod* = λ*f2*. *F* (*A1.cod f1*, *f2*)
interpret *Fcod*: *functor A2 B ?Fcod* **using** *assms fixing-ide-gives-functor-1* **by** *auto*
let *?τ* = λ*f2*. *F* (*f1*, *f2*)
show *natural-transformation A2 B ?Fdom ?Fcod ?τ*
using *assms*
apply *unfold-locales*
using *extensionality*
apply *auto*[3]
using *A1xA2.arr-char preserves-comp A1.comp-cod-arr A1xA2.comp-char A2.comp-arr-dom*
apply (*metis fst-conv snd-conv*)
using *A1xA2.arr-char preserves-comp A2.comp-cod-arr A1xA2.comp-char A1.comp-arr-dom*
by (*metis fst-conv snd-conv*)
qed

lemma *fixing-arr-gives-natural-transformation-2*:
assumes *A2.arr f2*

shows *natural-transformation A1 B* (λ*f1*. *F* (*f1*, *A2.dom f2*)) (λ*f1*. *F* (*f1*, *A2.cod f2*))
(λ*f1*. *F* (*f1*, *f2*))
proof −
interpret *F′*: *binary-functor A2 A1 B sym*
using *assms*(*1*) *sym-is-binary-functor* **by** *auto*
have *natural-transformation A1 B* (λ*f1*. *sym* (*A2.dom f2*, *f1*)) (λ*f1*. *sym* (*A2.cod f2*, *f1*))
(λ*f1*. *sym* (*f2*, *f1*))
using *assms F′.fixing-arr-gives-natural-transformation-1* **by** *fast*
thus *?thesis* **by** *simp*
qed

Fixing one or the other argument of a binary functor to be a composite arrow yields a natural transformation that is a vertical composite.

lemma *preserves-comp-1*:
assumes *A1.seq f1′ f1*
shows (λ*f2*. *F* (*f1′* ⋅$_{A1}$ *f1*, *f2*)) =
vertical-composite.map A2 B (λ*f2*. *F* (*f1*, *f2*)) (λ*f2*. *F* (*f1′*, *f2*))
proof −
interpret *τ*: *natural-transformation A2 B*
‹λ*f2*. *F* (*A1.dom f1*, *f2*)› ‹λ*f2*. *F* (*A1.cod f1*, *f2*)› ‹λ*f2*. *F* (*f1*, *f2*)›
using *assms fixing-arr-gives-natural-transformation-1* **by** *blast*
interpret *τ′*: *natural-transformation A2 B*
‹λ*f2*. *F* (*A1.cod f1*, *f2*)› ‹λ*f2*. *F* (*A1.cod f1′*, *f2*)› ‹λ*f2*. *F* (*f1′*, *f2*)›
using *assms fixing-arr-gives-natural-transformation-1 A1.seqE* **by** *metis*
interpret *τ′oτ*: *vertical-composite A2 B*
‹λ*f2*. *F* (*A1.dom f1*, *f2*)› ‹λ*f2*. *F* (*A1.cod f1*, *f2*)› ‹λ*f2*. *F* (*A1.cod f1′*, *f2*)›
‹λ*f2*. *F* (*f1*, *f2*)› ‹λ*f2*. *F* (*f1′*, *f2*)› **..**
show (λ*f2*. *F* (*f1′* ⋅$_{A1}$ *f1*, *f2*)) = *τ′oτ.map*
proof
fix *f2*
have ¬*A2.arr f2* ⟹ *F* (*f1′* ⋅$_{A1}$ *f1*, *f2*) = *τ′oτ.map f2*
using *τ′oτ.extensionality extensionality* **by** *simp*
moreover have *A2.arr f2* ⟹ *F* (*f1′* ⋅$_{A1}$ *f1*, *f2*) = *τ′oτ.map f2*
using *τ′oτ.map-simp-1 assms fixing-arr-gives-natural-transformation-2*
natural-transformation.preserves-comp-1
by *fastforce*
ultimately show *F* (*f1′* ⋅$_{A1}$ *f1*, *f2*) = *τ′oτ.map f2* **by** *blast*
qed
qed

lemma *preserves-comp-2*:
assumes *A2.seq f2′ f2*
shows (λ*f1*. *F* (*f1*, *f2′* ⋅$_{A2}$ *f2*)) =
vertical-composite.map A1 B (λ*f1*. *F* (*f1*, *f2*)) (λ*f1*. *F* (*f1*, *f2′*))
proof −
interpret *F′*: *binary-functor A2 A1 B sym*
using *assms*(*1*) *sym-is-binary-functor* **by** *auto*
have (λ*f1*. *sym* (*f2′* ⋅$_{A2}$ *f2*, *f1*)) =
vertical-composite.map A1 B (λ*f1*. *sym* (*f2*, *f1*)) (λ*f1*. *sym* (*f2′*, *f1*))

```
    using assms F'.preserves-comp-1 by fastforce
  thus ?thesis by simp
qed

end
```

A binary functor transformation is a natural transformation between binary functors. We need a certain property of such transformations; namely, that if one or the other argument is fixed to be an identity, the result is a natural transformation.

```
locale binary-functor-transformation =
  A1: category A1 +
  A2: category A2 +
  B: category B +
  A1xA2: product-category A1 A2 +
  F: binary-functor A1 A2 B F +
  G: binary-functor A1 A2 B G +
  natural-transformation A1xA2.comp B F G τ
for A1 :: 'a1 comp      (infixr ‹·A1› 55)
and A2 :: 'a2 comp      (infixr ‹·A2› 55)
and B :: 'b comp        (infixr ‹·B› 55)
and F :: 'a1 * 'a2 ⇒ 'b
and G :: 'a1 * 'a2 ⇒ 'b
and τ :: 'a1 * 'a2 ⇒ 'b
begin

  notation A1xA2.comp    (infixr ‹·A1xA2› 55)
  notation A1xA2.in-hom  (‹«- : - →A1xA2 -»›)

  lemma fixing-ide-gives-natural-transformation-1:
  assumes A1.ide a1
  shows natural-transformation A2 B (λf2. F (a1, f2)) (λf2. G (a1, f2)) (λf2. τ (a1, f2))
  proof −
    interpret Fa1: functor A2 B ‹λf2. F (a1, f2)›
      using assms F.fixing-ide-gives-functor-1 by simp
    interpret Ga1: functor A2 B ‹λf2. G (a1, f2)›
      using assms G.fixing-ide-gives-functor-1 by simp
    show ?thesis
      using assms extensionality naturality1 naturality2
      apply (unfold-locales, auto)
       apply (metis A1.ide-char)
      by (metis A1.ide-char)
  qed

  lemma fixing-ide-gives-natural-transformation-2:
  assumes A2.ide a2
  shows natural-transformation A1 B (λf1. F (f1, a2)) (λf1. G (f1, a2)) (λf1. τ (f1, a2))
  proof −
    interpret Fa2: functor A1 B ‹λf1. F (f1, a2)›
      using assms F.fixing-ide-gives-functor-2 by simp
```

```
    interpret Ga2: functor A1 B ‹λf1. G (f1, a2)›
      using assms G.fixing-ide-gives-functor-2 by simp
    show ?thesis
      using assms extensionality naturality1 naturality2
      apply (unfold-locales, auto)
       apply (metis A2.ide-char)
      by (metis A2.ide-char)
  qed

 end

end
```

Chapter 14

FunctorCategory

theory *FunctorCategory*
imports *ConcreteCategory BinaryFunctor*
begin

The functor category $[A, B]$ is the category whose objects are functors from A to B and whose arrows correspond to natural transformations between these functors.

14.1 Construction

Since the arrows of a functor category cannot (in the context of the present development) be directly identified with natural transformations, but rather only with natural transformations that have been equipped with their domain and codomain functors, and since there is no natural value to serve as *null*, we use the general-purpose construction given by *concrete-category* to define this category.

locale *functor-category* =
  A: *category A* +
  B: *category B*
for *A* :: *'a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_B$› *55*)
begin

  notation *A.in-hom* (‹«- : - →$_A$ -»›)
  notation *B.in-hom* (‹«- : - →$_B$ -»›)

  type-synonym (*'aa*, *'bb*) *arr* = (*'aa* ⇒ *'bb*, *'aa* ⇒ *'bb*) *concrete-category.arr*

  sublocale *concrete-category* ‹*Collect* (*functor A B*)›
    ‹λ*F G*. *Collect* (*natural-transformation A B F G*)› ‹λ*F*. *F*›
    ‹λ*F G H* τ σ. *vertical-composite.map A B* σ τ›
    using *vcomp-assoc*
    apply (*unfold-locales*, *simp-all*)
  proof −
    fix *F G H* σ τ

assume F: *functor* $(\cdot_A)$ $(\cdot_B)$ F
assume G: *functor* $(\cdot_A)$ $(\cdot_B)$ G
assume H: *functor* $(\cdot_A)$ $(\cdot_B)$ H
assume σ: *natural-transformation* $(\cdot_A)$ $(\cdot_B)$ F G σ
assume τ: *natural-transformation* $(\cdot_A)$ $(\cdot_B)$ G H τ
interpret F: *functor* A B F **using** F **by** *simp*
interpret G: *functor* A B G **using** G **by** *simp*
interpret H: *functor* A B H **using** H **by** *simp*
interpret σ: *natural-transformation* A B F G σ
using σ **by** *simp*
interpret τ: *natural-transformation* A B G H τ
using τ **by** *simp*
interpret $\tau\sigma$: *vertical-composite* A B F G H σ τ
..
show *natural-transformation* $(\cdot_A)$ $(\cdot_B)$ F H (*vertical-composite.map* $(\cdot_A)$ $(\cdot_B)$ σ τ)
using $\tau\sigma$.*map-def* $\tau\sigma$.*is-natural-transformation* **by** *simp*
qed

lemma *is-concrete-category*:
shows *concrete-category* (*Collect* (*functor* A B))
(λF G. *Collect* (*natural-transformation* A B F G)) (λF. F)
(λF G H τ σ. *vertical-composite.map* A B σ τ)
..

abbreviation *comp* (**infixr** ‹·› *55*)
where *comp* ≡ *COMP*
notation *in-hom* (‹«- : - → -»›)

lemma *is-category*:
shows *category comp*
..

lemma *arrI* [*intro*]:
assumes $f \neq$ *null* **and** *natural-transformation* A B (*Dom* f) (*Cod* f) (*Map* f)
shows *arr* f
using *assms arr-char null-char*
by (*simp add*: *natural-transformation-def*)

lemma *arrE* [*elim*]:
assumes *arr* f
and $f \neq$ *null* ⟹ *natural-transformation* A B (*Dom* f) (*Cod* f) (*Map* f) ⟹ T
shows T
using *assms arr-char null-char* **by** *simp*

lemma *arr-MkArr* [*iff*]:
shows *arr* (*MkArr* F G τ) ⟷ *natural-transformation* A B F G τ
using *arr-char null-char arr-MkArr natural-transformation-def* **by** *fastforce*

lemma *ide-char* [*iff*]:

shows *ide t* $\longleftrightarrow$ *t* $\neq$ *null* $\wedge$ *functor A B* (*Map t*) $\wedge$ *Dom t* = *Map t* $\wedge$ *Cod t* = *Map t*
using *ide-char*$_{CC}$ *null-char* **by** *fastforce*

end

14.2 Additional Properties

In this section some additional facts are proved, which make it easier to work with the *functor-category* locale.

context *functor-category*
begin

lemma *Map-comp* [*simp*]:
assumes *seq t' t* **and** *A.seq a' a*
shows *Map* (*t'* · *t*) (*a'* ·$_A$ *a*) = *Map t' a'* ·$_B$ *Map t a*
proof −
interpret *t*: *natural-transformation A B* ‹*Dom t*› ‹*Cod t*› ‹*Map t*›
using *assms*(*1*) *arr-char seq-char* **by** *blast*
interpret *t'*: *natural-transformation A B* ‹*Cod t*› ‹*Cod t'*› ‹*Map t'*›
using *assms*(*1*) *arr-char seq-char* **by** *force*
interpret *t'ot*: *vertical-composite A B* ‹*Dom t*› ‹*Cod t*› ‹*Cod t'*› ‹*Map t*› ‹*Map t'*› **..**
show *?thesis*
using *B.comp-assoc assms seq-char t'.preserves-comp-2 t'ot.map-simp-2* **by** *auto*
qed

lemma *Map-comp'*:
assumes *seq t' t*
shows *Map* (*t'* · *t*) = *vertical-composite.map A B* (*Map t*) (*Map t'*)
proof −
interpret *t*: *natural-transformation A B* ‹*Dom t*› ‹*Cod t*› ‹*Map t*›
using *assms*(*1*) *arr-char seq-char* **by** *blast*
interpret *t'*: *natural-transformation A B* ‹*Cod t*› ‹*Cod t'*› ‹*Map t'*›
using *assms*(*1*) *arr-char seq-char* **by** *force*
interpret *t'ot*: *vertical-composite A B* ‹*Dom t*› ‹*Cod t*› ‹*Cod t'*› ‹*Map t*› ‹*Map t'*› **..**
show *?thesis*
using *assms*(*1*) *seq-char t'ot.natural-transformation-axioms* **by** *simp*
qed

lemma *MkArr-eqI*:
assumes *F* = *F'* **and** *G* = *G'* **and** $\tau = \tau'$
shows *MkArr F G* τ = *MkArr F' G'* τ'
using *assms* **by** *simp*

lemma *iso-char* [*iff*]:
shows *iso t* $\longleftrightarrow$ *t* $\neq$ *null* $\wedge$ *natural-isomorphism A B* (*Dom t*) (*Cod t*) (*Map t*)
proof
assume *t*: *iso t*
show *t* $\neq$ *null* $\wedge$ *natural-isomorphism A B* (*Dom t*) (*Cod t*) (*Map t*)

```
proof
  show t ≠ null using t arr-char iso-is-arr by auto
  from t obtain t' where t': inverse-arrows t t' by blast
  interpret τ: natural-transformation A B ‹Dom t› ‹Cod t› ‹Map t›
    using t arr-char iso-is-arr by auto
  interpret τ': natural-transformation A B ‹Cod t› ‹Dom t› ‹Map t'›
    using t' arr-char dom-char seq-char
    by (metis arrE ide-compE inverse-arrowsE)
  interpret τ'oτ: vertical-composite A B ‹Dom t› ‹Cod t› ‹Dom t› ‹Map t› ‹Map t'› ..
  interpret τoτ': vertical-composite A B ‹Cod t› ‹Dom t› ‹Cod t› ‹Map t'› ‹Map t› ..
  show natural-isomorphism A B (Dom t) (Cod t) (Map t)
  proof
    fix a
    assume a: A.ide a
    show B.iso (Map t a)
    proof
      have 1: τ'oτ.map = Dom t ∧ τoτ'.map = Cod t
        using t t'
        by (metis (no-types, lifting) Map-dom concrete-category.Map-comp
            concrete-category-axioms ide-compE inverse-arrowsE seq-char)
      show B.inverse-arrows (Map t a) (Map t' a)
        using a 1 τoτ'.map-simp-ide τ'oτ.map-simp-ide τ.F.preserves-ide τ.G.preserves-ide
        by auto
    qed
  qed
qed
next
assume t: t ≠ null ∧ natural-isomorphism A B (Dom t) (Cod t) (Map t)
show iso t
proof
  interpret τ: natural-isomorphism A B ‹Dom t› ‹Cod t› ‹Map t›
    using t by auto
  interpret τ': inverse-transformation A B ‹Dom t› ‹Cod t› ‹Map t› ..
  have 1: vertical-composite.map A B (Map t) τ'.map = Dom t ∧
          vertical-composite.map A B τ'.map (Map t) = Cod t
    using τ.natural-isomorphism-axioms vertical-composite-inverse-iso
          vertical-composite-iso-inverse
    by blast
  show inverse-arrows t (MkArr (Cod t) (Dom t) (τ'.map))
  proof
    show 2: ide (MkArr (Cod t) (Dom t) τ'.map · t)
      using t 1
     by (metis (no-types, lifting) MkArr-Map MkIde-Dom τ'.natural-transformation-axioms
         τ.natural-transformation-axioms arrI arr-MkArr comp-MkArr ide-dom)
    show ide (t · MkArr (Cod t) (Dom t) τ'.map)
      using t 1 2
      by (metis Dom.simps(1) Map.simps(1) τ.natural-transformation-axioms arrI
          cod-char cod-comp comp-char ide-char' ide-compE)
  qed
```

```
    qed
  qed

end
```

14.3 Evaluation Functor

This section defines the evaluation map that applies an arrow of the functor category [A, B] to an arrow of A to obtain an arrow of B and shows that it is functorial.

```
locale evaluation-functor =
  A: category A +
  B: category B +
  A-B: functor-category A B +
  A-BxA: product-category A-B.comp A
for A :: 'a comp          (infixr ‹·A› 55)
and B :: 'b comp          (infixr ‹·B› 55)
begin

  notation A-B.comp       (infixr ‹·[A,B]› 55)
  notation A-BxA.comp     (infixr ‹·[A,B]xA› 55)
  notation A-B.in-hom     (‹«- : - →[A,B] -»›)
  notation A-BxA.in-hom   (‹«- : - →[A,B]xA -»›)

  definition map
  where map Fg ≡ if A-BxA.arr Fg then A-B.Map (fst Fg) (snd Fg) else B.null

  lemma map-simp:
  assumes A-BxA.arr Fg
  shows map Fg = A-B.Map (fst Fg) (snd Fg)
    using assms map-def by auto

  lemma is-functor:
  shows functor A-BxA.comp B map
  proof
    show ⋀Fg. ¬ A-BxA.arr Fg ⟹ map Fg = B.null
      using map-def by auto
    fix Fg
    assume Fg: A-BxA.arr Fg
    let ?F = fst Fg and ?g = snd Fg
    have F: A-B.arr ?F using Fg by auto
    have g: A.arr ?g using Fg by auto
    have DomF: A-B.Dom ?F = A-B.Map (A-B.dom ?F) using F by simp
    have CodF: A-B.Cod ?F = A-B.Map (A-B.cod ?F) using F by simp
    interpret F: natural-transformation A B ‹A-B.Dom ?F› ‹A-B.Cod ?F› ‹A-B.Map ?F›
      using Fg A-B.arr-char [of ?F] by blast
    show B.arr (map Fg) using Fg map-def by auto
    show B.dom (map Fg) = map (A-BxA.dom Fg)
      using g Fg map-def DomF
```

```
    by (metis (no-types, lifting) A-BxA.arr-dom A-BxA.dom-simp F.preserves-dom
        fst-conv snd-conv)
  show B.cod (map Fg) = map (A-BxA.cod Fg)
    using g Fg map-def CodF
    by (metis (no-types, lifting) A-BxA.arr-cod A-BxA.cod-simp F.preserves-cod
        fst-conv snd-conv)
  next
  fix Fg Fg'
  assume 1: A-BxA.seq Fg' Fg
  let ?F = fst Fg and ?g = snd Fg
  let ?F' = fst Fg' and ?g' = snd Fg'
  have F': A-B.arr ?F' using 1 A-BxA.seqE by blast
  have CodF: A-B.Cod ?F = A-B.Map (A-B.cod ?F)
    using 1 by (metis A-B.Map-cod A-B.seqE A-BxA.seqE_PC)
  have DomF': A-B.Dom ?F' = A-B.Map (A-B.dom ?F')
    using F' by simp
  have seq-F'F: A-B.seq ?F' ?F using 1 by blast
  have seq-g'g: A.seq ?g' ?g using 1 by blast
  interpret F: natural-transformation A B ‹A-B.Dom ?F› ‹A-B.Cod ?F› ‹A-B.Map ?F›
    using 1 A-B.arr-char by blast
  interpret F': natural-transformation A B ‹A-B.Cod ?F› ‹A-B.Cod ?F'› ‹A-B.Map ?F'›
    using 1 A-B.arr-char seq-F'F CodF DomF' A-B.seqE
    by (metis mem-Collect-eq)
  interpret F'oF: vertical-composite A B ‹A-B.Dom ?F› ‹A-B.Cod ?F› ‹A-B.Cod ?F'›
                                         ‹A-B.Map ?F› ‹A-B.Map ?F'› ..
  show map (Fg' ·[A,B]xA Fg) = map Fg' ·B map Fg
    unfolding map-def
    using 1 seq-F'F seq-g'g by auto
qed

end

sublocale evaluation-functor ⊆ functor A-BxA.comp B map
  using is-functor by auto
sublocale evaluation-functor ⊆ binary-functor A-B.comp A B map ..
```

14.4 Currying

This section defines the notion of currying of a natural transformation between binary functors, to obtain a natural transformation between functors into a functor category, along with the inverse operation of uncurrying. We have only proved here what is needed to establish the results in theory *Limit* about limits in functor categories and have not attempted to fully develop the functoriality and naturality properties of these notions.

```
locale currying =
A1: category A1 +
A2: category A2 +
B: category B
for A1 :: 'a1 comp          (infixr ‹·A1› 55)
```

and *A2* :: *'a2 comp* (**infixr** ‹·$_{A2}$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_{B}$› *55*)
begin

interpretation *A1xA2*: *product-category A1 A2* **..**
interpretation *A2-B*: *functor-category A2 B* **..**
interpretation *A2-BxA2*: *product-category A2-B.comp A2* **..**
interpretation *E*: *evaluation-functor A2 B* **..**

notation *A1xA2.comp* (**infixr** ‹·$_{A1xA2}$› *55*)
notation *A2-B.comp* (**infixr** ‹·$_{[A2,B]}$› *55*)
notation *A2-BxA2.comp* (**infixr** ‹·$_{[A2,B]xA2}$› *55*)
notation *A1xA2.in-hom* (‹«- : - →$_{A1xA2}$ -»›)
notation *A2-B.in-hom* (‹«- : - →$_{[A2,B]}$ -»›)
notation *A2-BxA2.in-hom* (‹«- : - →$_{[A2,B]xA2}$ -»›)

A proper definition for *curry* requires that it be parametrized by binary functors F and G that are the domain and codomain of the natural transformations to which it is being applied. Similar parameters are not needed in the case of *uncurry*.

definition *curry* :: (*'a1* × *'a2* ⇒ *'b*) ⇒ (*'a1* × *'a2* ⇒ *'b*) ⇒ (*'a1* × *'a2* ⇒ *'b*)
⇒ *'a1* ⇒ (*'a2*, *'b*) *A2-B.arr*
where *curry F G τ f1* = (*if A1.arr f1 then*
A2-B.MkArr (λ*f2*. *F* (*A1.dom f1*, *f2*)) (λ*f2*. *G* (*A1.cod f1*, *f2*))
(λ*f2*. *τ* (*f1*, *f2*))
else A2-B.null)

definition *uncurry* :: (*'a1* ⇒ (*'a2*, *'b*) *A2-B.arr*) ⇒ *'a1* × *'a2* ⇒ *'b*
where *uncurry τ f* ≡ *if A1xA2.arr f then E.map* (*τ* (*fst f*), *snd f*) *else B.null*

lemma *curry-simp*:
assumes *A1.arr f1*
shows *curry F G τ f1* = *A2-B.MkArr* (λ*f2*. *F* (*A1.dom f1*, *f2*)) (λ*f2*. *G* (*A1.cod f1*, *f2*))
(λ*f2*. *τ* (*f1*, *f2*))
using *assms curry-def* **by** *auto*

lemma *uncurry-simp*:
assumes *A1xA2.arr f*
shows *uncurry τ f* = *E.map* (*τ* (*fst f*), *snd f*)
using *assms uncurry-def* **by** *auto*

lemma *curry-in-hom*:
assumes *f1*: *A1.arr f1*
and *natural-transformation A1xA2.comp B F G τ*
shows «*curry F G τ f1* : *curry F F F* (*A1.dom f1*) →$_{[A2,B]}$ *curry G G G* (*A1.cod f1*)»
proof −
interpret *τ*: *natural-transformation A1xA2.comp B F G τ* **using** *assms* **by** *auto*
show *?thesis*
proof −
interpret *F-dom-f1*: *functor A2 B* ‹λ*f2*. *F* (*A1.dom f1*, *f2*)›

```
    using f1 τ.F.extensionality apply (unfold-locales, simp-all)
   by (metis A1.arr-dom A1.comp-arr-dom A1.dom-dom A1xA2.comp-simp A1xA2.seqI_PC
       τ.F.as-nat-trans.preserves-comp-2 fst-conv snd-conv)
  interpret G-cod-f1: functor A2 B ‹λf2. G (A1.cod f1, f2)›
    using f1 τ.G.extensionality A1.arr-cod-iff-arr
    apply (unfold-locales, simp-all)
    by (metis A1.comp-arr-dom A1.dom-cod A1xA2.comp-simp A1xA2.seqI_PC
       τ.G.preserves-comp fst-conv snd-conv)
  have natural-transformation A2 B (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2))
                               (λf2. τ (f1, f2))
    using f1 τ.extensionality apply (unfold-locales, simp-all)
  proof −
    fix f2
    assume f2: A2.arr f2
    show G (A1.cod f1, f2) ·_B τ (f1, A2.dom f2) = τ (f1, f2)
      using f1 f2 τ.preserves-comp-1 [of (A1.cod f1, f2) (f1, A2.dom f2)]
            A1.comp-cod-arr A2.comp-arr-dom
      by simp
    show τ (f1, A2.cod f2) ·_B F (A1.dom f1, f2) = τ (f1, f2)
      using f1 f2 τ.preserves-comp-2 [of (f1, A2.cod f2) (A1.dom f1, f2)]
            A1.comp-arr-dom A2.comp-cod-arr
      by simp
  qed
  thus ?thesis
    using f1 curry-simp by auto
 qed
qed

lemma curry-preserves-functors:
assumes functor A1xA2.comp B F
shows functor A1 A2-B.comp (curry F F F)
proof −
 interpret F: functor A1xA2.comp B F using assms by auto
 interpret F: binary-functor A1 A2 B F ..
 show ?thesis
   using curry-def F.fixing-arr-gives-natural-transformation-1
         A2-B.comp-char F.preserves-comp-1 curry-simp A2-B.seq-char
   apply unfold-locales by auto
qed

lemma curry-preserves-transformations:
assumes natural-transformation A1xA2.comp B F G τ
shows natural-transformation A1 A2-B.comp (curry F F F) (curry G G G) (curry F G τ)
proof −
 interpret τ: natural-transformation A1xA2.comp B F G τ using assms by auto
 interpret τ: binary-functor-transformation A1 A2 B F G τ ..
 interpret curry-F: functor A1 A2-B.comp ‹curry F F F›
   using curry-preserves-functors τ.F.functor-axioms by simp
 interpret curry-G: functor A1 A2-B.comp ‹curry G G G›
```

```
  using curry-preserves-functors τ.G.functor-axioms by simp
show ?thesis
proof
  show ⋀f2. ¬ A1.arr f2 ⟹ curry F G τ f2 = A2-B.null
    using curry-def by simp
  fix f1
  assume f1: A1.arr f1
  show A2-B.arr (curry F G τ f1)
    using assms f1 curry-in-hom by blast
  show curry G G G f1 ⋅[A2,B] curry F G τ (A1.dom f1) = curry F G τ f1
  proof −
    interpret τ-dom-f1: natural-transformation A2 B ‹λf2. F (A1.dom f1, f2)›
                        ‹λf2. G (A1.dom f1, f2)› ‹λf2. τ (A1.dom f1, f2)›
      using assms f1 curry-in-hom A1.ide-dom τ.fixing-ide-gives-natural-transformation-1
      by blast
    interpret G-f1: natural-transformation A2 B
                        ‹λf2. G (A1.dom f1, f2)› ‹λf2. G (A1.cod f1, f2)› ‹λf2. G (f1, f2)›
      using f1 τ.G.fixing-arr-gives-natural-transformation-1 by simp
    interpret G-f1oτ-dom-f1: vertical-composite A2 B
                             ‹λf2. F (A1.dom f1, f2)› ‹λf2. G (A1.dom f1, f2)›
                             ‹λf2. G (A1.cod f1, f2)›
                             ‹λf2. τ (A1.dom f1, f2)› ‹λf2. G (f1, f2)› ..
    have curry G G G f1 ⋅[A2,B] curry F G τ (A1.dom f1)
      = A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2)) G-f1oτ-dom-f1.map
    proof −
      have A2-B.seq (curry G G G f1) (curry F G τ (A1.dom f1))
        using f1 curry-in-hom [of A1.dom f1] τ.natural-transformation-axioms by force
      thus ?thesis
        using f1 curry-simp A2-B.comp-char [of curry G G G f1 curry F G τ (A1.dom f1)]
        by simp
    qed
    also have ... = A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2))
                               (λf2. τ (f1, f2))
    proof (intro A2-B.MkArr-eqI)
      show (λf2. F (A1.dom f1, f2)) = (λf2. F (A1.dom f1, f2)) by simp
      show (λf2. G (A1.cod f1, f2)) = (λf2. G (A1.cod f1, f2)) by simp
      show G-f1oτ-dom-f1.map = (λf2. τ (f1, f2))
      proof
        fix f2
        have ¬A2.arr f2 ⟹ G-f1oτ-dom-f1.map f2 = (λf2. τ (f1, f2)) f2
          using f1 G-f1oτ-dom-f1.extensionality τ.extensionality by simp
        moreover have A2.arr f2 ⟹ G-f1oτ-dom-f1.map f2 = (λf2. τ (f1, f2)) f2
        proof −
          interpret τ-f1: natural-transformation A2 B ‹λf2. F (A1.dom f1, f2)›
                          ‹λf2. G (A1.cod f1, f2)› ‹λf2. τ (f1, f2)›
            using assms f1 curry-in-hom [of f1] curry-simp by auto
          fix f2
          assume f2: A2.arr f2
          show G-f1oτ-dom-f1.map f2 = (λf2. τ (f1, f2)) f2
```

```
      using f1 f2 G-f1oτ-dom-f1.map-simp-2 B.comp-assoc τ.naturality1
      by fastforce
    qed
    ultimately show G-f1oτ-dom-f1.map f2 = (λf2. τ (f1, f2)) f2 by blast
  qed
 qed
 also have ... = curry F G τ f1 using f1 curry-def by simp
 finally show ?thesis by blast
qed
show curry F G τ (A1.cod f1) ·[A2,B] curry F F F f1 = curry F G τ f1
proof −
 interpret τ-cod-f1: natural-transformation A2 B ‹λf2. F (A1.cod f1, f2)›
                     ‹λf2. G (A1.cod f1, f2)› ‹λf2. τ (A1.cod f1, f2)›
  using assms f1 curry-in-hom A1.ide-cod τ.fixing-ide-gives-natural-transformation-1
  by blast
 interpret F-f1: natural-transformation A2 B
                     ‹λf2. F (A1.dom f1, f2)› ‹λf2. F (A1.cod f1, f2)› ‹λf2. F (f1, f2)›
  using f1 τ.F.fixing-arr-gives-natural-transformation-1 by simp
 interpret τ-cod-f1oF-f1: vertical-composite A2 B
                          ‹λf2. F (A1.dom f1, f2)› ‹λf2. F (A1.cod f1, f2)›
                          ‹λf2. G (A1.cod f1, f2)›
                          ‹λf2. F (f1, f2)› ‹λf2. τ (A1.cod f1, f2)› ..
 have curry F G τ (A1.cod f1) ·[A2,B] curry F F F f1
    = A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2)) τ-cod-f1oF-f1.map
 proof −
  have
      curry F F F f1 =
        A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. F (A1.cod f1, f2))
                   (λf2. F (f1, f2)) ∧
       «curry F F F f1 : curry F F F (A1.dom f1) →[A2,B] curry F F F (A1.cod f1)»
    using f1 curry-F.preserves-hom curry-simp by blast
  moreover have
      curry F G τ (A1.dom f1) =
        A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.dom f1, f2))
                   (λf2. τ (A1.dom f1, f2)) ∧
        «curry F G τ (A1.cod f1) :
          curry F F F (A1.cod f1) →[A2,B] curry G G G (A1.cod f1)»
    using assms f1 curry-in-hom [of A1.cod f1] curry-def A1.arr-cod-iff-arr by simp
  ultimately show ?thesis
    using f1 curry-def by fastforce
 qed
 also have ... = A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2))
                            (λf2. τ (f1, f2))
 proof (intro A2-B.MkArr-eqI)
  show (λf2. F (A1.dom f1, f2)) = (λf2. F (A1.dom f1, f2)) by simp
  show (λf2. G (A1.cod f1, f2)) = (λf2. G (A1.cod f1, f2)) by simp
  show τ-cod-f1oF-f1.map = (λf2. τ (f1, f2))
  proof
    fix f2
```

```
      have ¬A2.arr f2 ⟹ τ-cod-f1oF-f1.map f2 = (λf2. τ (f1, f2)) f2
        using f1 by (simp add: τ.extensionality τ-cod-f1oF-f1.extensionality)
      moreover have A2.arr f2 ⟹ τ-cod-f1oF-f1.map f2 = (λf2. τ (f1, f2)) f2
      proof −
        interpret τ-f1: natural-transformation A2 B ‹λf2. F (A1.dom f1, f2)›
                        ‹λf2. G (A1.cod f1, f2)› ‹λf2. τ (f1, f2)›
          using assms f1 curry-in-hom [of f1] curry-simp by auto
        fix f2
        assume f2: A2.arr f2
        show τ-cod-f1oF-f1.map f2 = (λf2. τ (f1, f2)) f2
          using f1 f2 τ-cod-f1oF-f1.map-simp-1 B.comp-assoc τ.naturality2
          by fastforce
      qed
      ultimately show τ-cod-f1oF-f1.map f2 = (λf2. τ (f1, f2)) f2 by blast
    qed
  qed
  also have ... = curry F G τ f1 using f1 curry-def by simp
  finally show ?thesis by blast
 qed
qed

lemma uncurry-preserves-functors:
assumes functor A1 A2-B.comp F
shows functor A1xA2.comp B (uncurry F)
proof −
 interpret F: functor A1 A2-B.comp F using assms by auto
 show ?thesis
  using uncurry-def
  apply (unfold-locales)
     apply auto[4]
 proof −
  fix f g :: 'a1 * 'a2
  let ?f1 = fst f
  let ?f2 = snd f
  let ?g1 = fst g
  let ?g2 = snd g
  assume fg: A1xA2.seq g f
  have f: A1xA2.arr f using fg A1xA2.seqE by blast
  have f1: A1.arr ?f1 using f by auto
  have f2: A2.arr ?f2 using f by auto
  have g: «g : A1xA2.cod f →A1xA2 A1xA2.cod g»
    using fg A1xA2.dom-char A1xA2.cod-char
    by (elim A1xA2.seqE, intro A1xA2.in-homI, auto)
  let ?g1 = fst g
  let ?g2 = snd g
  have g1: «?g1 : A1.cod ?f1 →A1 A1.cod ?g1»
    using f g by (intro A1.in-homI, auto)
  have g2: «?g2 : A2.cod ?f2 →A2 A2.cod ?g2»
```

```
    using f g by (intro A2.in-homI, auto)
  interpret Ff1: natural-transformation A2 B ‹A2-B.Dom (F ?f1)› ‹A2-B.Cod (F ?f1)›
                                             ‹A2-B.Map (F ?f1)›
    using f A2-B.arr-char [of F ?f1] by auto
  interpret Fg1: natural-transformation A2 B ‹A2-B.Cod (F ?f1)› ‹A2-B.Cod (F ?g1)›
                                             ‹A2-B.Map (F ?g1)›
    using f1 g1 A2-B.arr-char F.preserves-arr
          A2-B.Map-dom [of F ?g1] A2-B.Map-cod [of F ?f1]
    by fastforce
  interpret Fg1oFf1: vertical-composite A2 B
                       ‹A2-B.Dom (F ?f1)› ‹A2-B.Cod (F ?f1)› ‹A2-B.Cod (F ?g1)›
                       ‹A2-B.Map (F ?f1)› ‹A2-B.Map (F ?g1)› ..
  show uncurry F (g ·A1xA2 f) = uncurry F g ·B uncurry F f
    using f1 g1 g2 g2 f g fg E.map-simp uncurry-def by auto
 qed
qed

lemma uncurry-preserves-transformations:
assumes natural-transformation A1 A2-B.comp F G τ
shows natural-transformation A1xA2.comp B (uncurry F) (uncurry G) (uncurry τ)
proof −
 interpret τ: natural-transformation A1 A2-B.comp F G τ using assms by auto
 interpret functor A1xA2.comp B ‹uncurry F›
   using τ.F.functor-axioms uncurry-preserves-functors by blast
 interpret functor A1xA2.comp B ‹uncurry G›
   using τ.G.functor-axioms uncurry-preserves-functors by blast
 show ?thesis
 proof
   fix f
   show ¬ A1xA2.arr f ⟹ uncurry τ f = B.null
     using uncurry-def by auto
   assume f: A1xA2.arr f
   let ?f1 = fst f
   let ?f2 = snd f
   show B.arr (uncurry τ f)
     using f uncurry-def by simp
   show uncurry G f ·B uncurry τ (A1xA2.dom f) = uncurry τ f
     using f uncurry-def τ.naturality1 A2-BxA2.seq-char A2.comp-arr-dom
           E.preserves-comp [of (G (fst f), snd f) (τ (A1.dom (fst f)), A2.dom (snd f))]
     by auto
   show uncurry τ (A1xA2.cod f) ·B uncurry F f = uncurry τ f
   proof −
     have 1: A1.arr ?f1 ∧ A1.arr (fst (A1.cod ?f1, A2.cod ?f2)) ∧
              A1.cod ?f1 = A1.dom (fst (A1.cod ?f1, A2.cod ?f2)) ∧
              A2.seq (snd (A1.cod ?f1, A2.cod ?f2)) ?f2
       using f A1.arr-cod-iff-arr A2.arr-cod-iff-arr by auto
     hence 2:
         ?f2 = A2 (snd (τ (fst (A1xA2.cod f)), snd (A1xA2.cod f))) (snd (F ?f1, ?f2))
       using f A2.comp-cod-arr by simp
```

```
      have A2-B.arr (τ ?f1) using 1 by force
      thus ?thesis
        unfolding uncurry-def E.map-def
        using f 1 2
        apply simp
        by (metis (no-types, lifting) A2-B.Map-comp ‹A2-B.arr (τ (fst f))› τ.naturality2)

    qed
  qed
qed

lemma uncurry-curry:
assumes natural-transformation A1xA2.comp B F G τ
shows uncurry (curry F G τ) = τ
proof
  interpret τ: natural-transformation A1xA2.comp B F G τ using assms by auto
  interpret curry-τ: natural-transformation A1 A2-B.comp ‹curry F F F› ‹curry G G G›
                                                         ‹curry F G τ›
    using assms curry-preserves-transformations by auto
  fix f
  have ¬A1xA2.arr f ⟹ uncurry (curry F G τ) f = τ f
    using curry-def uncurry-def τ.extensionality by auto
  moreover have A1xA2.arr f ⟹ uncurry (curry F G τ) f = τ f
  proof −
    assume f: A1xA2.arr f
    have 1: A2-B.Map (curry F G τ (fst f)) (snd f) = τ (fst f, snd f)
      using f A1xA2.arr-char curry-def by simp
    thus uncurry (curry F G τ) f = τ f
      unfolding uncurry-def E.map-def
      using f 1 A1xA2.arr-char [of f] by simp
  qed
  ultimately show uncurry (curry F G τ) f = τ f by blast
qed

lemma curry-uncurry:
assumes functor A1 A2-B.comp F and functor A1 A2-B.comp G
and natural-transformation A1 A2-B.comp F G τ
shows curry (uncurry F) (uncurry G) (uncurry τ) = τ
proof
  interpret F: functor A1 A2-B.comp F using assms(1) by auto
  interpret G: functor A1 A2-B.comp G using assms(2) by auto
  interpret τ: natural-transformation A1 A2-B.comp F G τ using assms(3) by auto
  interpret uncurry-F: functor A1xA2.comp B ‹uncurry F›
    using F.functor-axioms uncurry-preserves-functors by auto
  interpret uncurry-G: functor A1xA2.comp B ‹uncurry G›
    using G.functor-axioms uncurry-preserves-functors by auto
  fix f1
  have ¬A1.arr f1 ⟹ curry (uncurry F) (uncurry G) (uncurry τ) f1 = τ f1
    using curry-def uncurry-def τ.extensionality by simp
```

```
moreover have A1.arr f1 ⟹ curry (uncurry F) (uncurry G) (uncurry τ) f1 = τ f1
proof −
  assume f1: A1.arr f1
  interpret uncurry-τ:
      natural-transformation A1xA2.comp B ‹uncurry F› ‹uncurry G› ‹uncurry τ›
    using τ.natural-transformation-axioms uncurry-preserves-transformations [of F G τ]
    by simp
  have curry (uncurry F) (uncurry G) (uncurry τ) f1 =
        A2-B.MkArr (λf2. uncurry F (A1.dom f1, f2)) (λf2. uncurry G (A1.cod f1, f2))
                   (λf2. uncurry τ (f1, f2))
    using f1 curry-def by simp
  also have ... = A2-B.MkArr (λf2. uncurry F (A1.dom f1, f2))
                             (λf2. uncurry G (A1.cod f1, f2))
                             (λf2. E.map (τ f1, f2))
  proof −
    have (λf2. uncurry τ (f1, f2)) = (λf2. E.map (τ f1, f2))
      using f1 uncurry-def E.extensionality by auto
    thus ?thesis by simp
  qed
  also have ... = τ f1
  proof −
    have A2-B.Dom (τ f1) = (λf2. uncurry F (A1.dom f1, f2))
    proof −
      have A2-B.Dom (τ f1) = A2-B.Map (A2-B.dom (τ f1))
        using f1 A2-B.ide-char A2-B.Map-dom A2-B.dom-char by auto
      also have ... = A2-B.Map (F (A1.dom f1))
        using f1 by simp
      also have ... = (λf2. uncurry F (A1.dom f1, f2))
      proof
        fix f2
        interpret F-dom-f1: functor A2 B ‹A2-B.Map (F (A1.dom f1))›
          using f1 A2-B.ide-char F.preserves-ide by simp
        show A2-B.Map (F (A1.dom f1)) f2 = uncurry F (A1.dom f1, f2)
          using f1 uncurry-def E.map-simp F-dom-f1.extensionality by auto
      qed
      finally show ?thesis by auto
    qed
    moreover have A2-B.Cod (τ f1) = (λf2. uncurry G (A1.cod f1, f2))
    proof −
      have A2-B.Cod (τ f1) = A2-B.Map (A2-B.cod (τ f1))
        using f1 A2-B.ide-char A2-B.Map-cod A2-B.cod-char by auto
      also have ... = A2-B.Map (G (A1.cod f1))
        using f1 by simp
      also have ... = (λf2. uncurry G (A1.cod f1, f2))
      proof
        fix f2
        interpret G-cod-f1: functor A2 B ‹A2-B.Map (G (A1.cod f1))›
          using f1 A2-B.ide-char G.preserves-ide by simp
        show A2-B.Map (G (A1.cod f1)) f2 = uncurry G (A1.cod f1, f2)
```

```
          using f1 uncurry-def E.map-simp G-cod-f1.extensionality by auto
        qed
        finally show ?thesis by auto
      qed
      moreover have A2-B.Map (τ f1) = (λf2. E.map (τ f1, f2))
      proof
        fix f2
        have ¬A2.arr f2 ⟹ A2-B.Map (τ f1) f2 = (λf2. E.map (τ f1, f2)) f2
          using f1 A2-B.arrE τ.preserves-reflects-arr natural-transformation.extensionality
          by (metis (no-types, lifting) E.fixing-arr-gives-natural-transformation-1)
        moreover have A2.arr f2 ⟹ A2-B.Map (τ f1) f2 = (λf2. E.map (τ f1, f2)) f2
          using f1 E.map-simp by fastforce
        ultimately show A2-B.Map (τ f1) f2 = (λf2. E.map (τ f1, f2)) f2 by blast
      qed
      ultimately show ?thesis
        using f1 A2-B.MkArr-Map τ.preserves-reflects-arr by metis
    qed
    finally show ?thesis by auto
  qed
  ultimately show curry (uncurry F) (uncurry G) (uncurry τ) f1 = τ f1 by blast
qed

end

locale curried-functor =
  currying A1 A2 B +
  A1xA2: product-category A1 A2 +
  A2-B: functor-category A2 B +
  F: binary-functor A1 A2 B F
for A1 :: 'a1 comp          (infixr ‹·A1› 55)
and A2 :: 'a2 comp          (infixr ‹·A2› 55)
and B :: 'b comp            (infixr ‹·B› 55)
and F :: 'a1 * 'a2 ⇒ 'b
begin

  notation A1xA2.comp       (infixr ‹·A1xA2› 55)
  notation A2-B.comp        (infixr ‹·[A2,B]› 55)
  notation A1xA2.in-hom     (‹«- : - →A1xA2 -»›)
  notation A2-B.in-hom      (‹«- : - →[A2,B] -»›)

  definition map
  where map ≡ curry F F F

  lemma map-simp [simp]:
  assumes A1.arr f1
  shows map f1 =
        A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. F (A1.cod f1, f2)) (λf2. F (f1, f2))
    using assms map-def curry-simp by auto
```

lemma *is-functor*:
shows *functor A1 A2-B.comp map*
using *F.functor-axioms map-def curry-preserves-functors* **by** *simp*

end

sublocale *curried-functor* ⊆ *functor A1 A2-B.comp map*
using *is-functor* **by** *auto*

locale *curried-functor′* =
A1: *category A1* +
A2: *category A2* +
A1xA2: *product-category A1 A2* +
currying A2 A1 B +
F: *binary-functor A1 A2 B F* +
A1-B: *functor-category A1 B*
for *A1* :: *'a1 comp* (**infixr** ‹·$_{A1}$› *55*)
and *A2* :: *'a2 comp* (**infixr** ‹·$_{A2}$› *55*)
and *B* :: *'b comp* (**infixr** ‹·$_{B}$› *55*)
and *F* :: *'a1 ∗ 'a2 ⇒ 'b*
begin

notation *A1xA2.comp* (**infixr** ‹·$_{A1xA2}$› *55*)
notation *A1-B.comp* (**infixr** ‹·$_{[A1,B]}$› *55*)
notation *A1xA2.in-hom* (‹«- : - →$_{A1xA2}$ -»›)
notation *A1-B.in-hom* (‹«- : - →$_{[A1,B]}$ -»›)

definition *map*
where *map* ≡ *curry F.sym F.sym F.sym*

lemma *map-simp* [*simp*]:
assumes *A2.arr f2*
shows *map f2* =
A1-B.MkArr (λ*f1*. *F* (*f1*, *A2.dom f2*)) (λ*f1*. *F* (*f1*, *A2.cod f2*)) (λ*f1*. *F* (*f1*, *f2*))
using *assms map-def curry-simp* **by** *simp*

lemma *is-functor*:
shows *functor A2 A1-B.comp map*
proof −
interpret *A2xA1*: *product-category A2 A1* ..
interpret *F′*: *binary-functor A2 A1 B F.sym*
using *F.sym-is-binary-functor* **by** *simp*
have *functor A2xA1.comp B F.sym* ..
thus *?thesis* **using** *map-def curry-preserves-functors* **by** *simp*
qed

end

sublocale *curried-functor′* ⊆ *functor A2 A1-B.comp map*

using *is-functor* **by** *auto*

end

Chapter 15

Yoneda

theory *Yoneda*
imports *DualCategory SetCat FunctorCategory*
begin

This theory defines the notion of a "hom-functor" and gives a proof of the Yoneda Lemma. In traditional developments of category theory based on set theories such as ZFC, hom-functors are normally defined to be functors into the large category **Set** whose objects are of *all* sets and whose arrows are functions between sets. However, in HOL there does not exist a single "type of all sets", so the notion of the category of *all* sets and functions does not make sense. To work around this, we consider a more general setting consisting of a category C together with a set category S and a function φ such that whenever b and a are objects of C then $\varphi\ (b,\ a)$ maps $C.hom\ b\ a$ injectively to $S.Univ$. We show that these data induce a binary functor Hom from $Cop \times C$ to S in such a way that φ is rendered natural in $(b,\ a)$. The Yoneda lemma is then proved for the Yoneda functor determined by Hom.

15.1 Hom-Functors

A hom-functor for a category C allows us to regard the hom-sets of C as objects of a category S of sets and functions. Any description of a hom-functor for C must therefore specify the category S and provide some sort of correspondence between arrows of C and elements of objects of S. If we are to think of each hom-set $C.hom\ b\ a$ of C as corresponding to an object $Hom\ (b,\ a)$ of S then at a minimum it ought to be the case that the correspondence between arrows and elements is bijective between $C.hom\ b\ a$ and $Hom\ (b,\ a)$. The *hom-functor* locale defined below captures this idea by assuming a set category S and a function φ taking arrows of C to elements of $S.Univ$, such that φ is injective on each set $C.hom\ b\ a$. We show that these data induce a functor Hom from $Cop \times C$ to S in such a way that φ becomes a natural bijection between $C.hom\ b\ a$ and $Hom\ (b,\ a)$.

locale *hom-functor* =
  C: *category* C +

S: *set-category S setp*
for *C* :: *'c comp* (**infixr** ‹·› *55*)
and *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *setp* :: *'s set ⇒ bool*
and *φ* :: *'c ∗ 'c ⇒ 'c ⇒ 's +*
assumes *maps-arr-to-Univ*: *C.arr f ⟹ φ (C.dom f, C.cod f) f ∈ S.Univ*
and *local-inj*: *⟦ C.ide b; C.ide a ⟧ ⟹ inj-on (φ (b, a)) (C.hom b a)*
and *small-homs*: *⟦ C.ide b; C.ide a ⟧ ⟹ setp (φ (b, a) ‘ C.hom b a)*
begin

sublocale *Cop*: *dual-category C* **..**
sublocale *CopxC*: *product-category Cop.comp C* **..**

notation *S.in-hom* (‹«- : - →$_S$ -»›)
notation *CopxC.comp* (**infixr** ‹⊙› *55*)
notation *CopxC.in-hom* (‹«- : - ⇌ -»›)

definition *set*
where *set ba ≡ φ (fst ba, snd ba) ‘ C.hom (fst ba) (snd ba)*

lemma *set-subset-Univ*:
assumes *C.ide b* **and** *C.ide a*
shows *set (b, a) ⊆ S.Univ*
using *assms set-def maps-arr-to-Univ CopxC.ide-char* **by** *auto*

definition *ψ* :: *'c ∗ 'c ⇒ 's ⇒ 'c*
where *ψ ba = inv-into (C.hom (fst ba) (snd ba)) (φ ba)*

lemma *φ-mapsto*:
assumes *C.ide b* **and** *C.ide a*
shows *φ (b, a) ∈ C.hom b a → set (b, a)*
using *assms set-def maps-arr-to-Univ* **by** *auto*

lemma *ψ-mapsto*:
assumes *C.ide b* **and** *C.ide a*
shows *ψ (b, a) ∈ set (b, a) → C.hom b a*
using *assms set-def ψ-def local-inj* **by** *auto*

lemma *ψ-φ* [*simp*]:
assumes *«f : b → a»*
shows *ψ (b, a) (φ (b, a) f) = f*
using *assms local-inj* [*of b a*] *ψ-def* **by** *fastforce*

lemma *φ-ψ* [*simp*]:
assumes *C.ide b* **and** *C.ide a*
and *x ∈ set (b, a)*
shows *φ (b, a) (ψ (b, a) x) = x*
using *assms set-def local-inj ψ-def* **by** *auto*

lemma *ψ-img-set*:
assumes *C.ide b* **and** *C.ide a*
shows *ψ (b, a) ‘ set (b, a) = C.hom b a*
using *assms ψ-def set-def local-inj* **by** *auto*

A hom-functor maps each arrow (g, f) of *CopxC* to the arrow of the set category S corresponding to the function that takes an arrow h of $(\cdot)$ to the arrow $f \cdot h \cdot g$ of $(\cdot)$ obtained by precomposing with g and postcomposing with f.

definition *map*
where *map gf =*
(if CopxC.arr gf then
S.mkArr (set (CopxC.dom gf)) (set (CopxC.cod gf))
(φ (CopxC.cod gf) o (λh. snd gf · h · fst gf) o ψ (CopxC.dom gf))
else S.null)

lemma *arr-map*:
assumes *CopxC.arr gf*
shows *S.arr (map gf)*
proof −
have *φ (CopxC.cod gf) o (λh. snd gf · h · fst gf) o ψ (CopxC.dom gf)*
∈ set (CopxC.dom gf) → set (CopxC.cod gf)
using *assms φ-mapsto [of fst (CopxC.cod gf) snd (CopxC.cod gf)]*
ψ-mapsto [of fst (CopxC.dom gf) snd (CopxC.dom gf)]
by *fastforce*
thus *?thesis*
using *assms map-def set-subset-Univ small-homs*
by *(simp add: set-def)*
qed

lemma *map-ide [simp]*:
assumes *C.ide b* **and** *C.ide a*
shows *map (b, a) = S.mkIde (set (b, a))*
proof −
have *map (b, a) = S.mkArr (set (b, a)) (set (b, a))*
(φ (b, a) o (λh. a · h · b) o ψ (b, a))
using *assms map-def* **by** *auto*
also have *... = S.mkArr (set (b, a)) (set (b, a)) (λh. h)*
proof −
have *S.mkArr (set (b, a)) (set (b, a)) (λh. h) = ...*
using *assms set-subset-Univ set-def C.comp-arr-dom C.comp-cod-arr*
S.arr-mkIde small-homs
by *(intro S.mkArr-eqI′, simp, fastforce)*
thus *?thesis* **by** *auto*
qed
also have *... = S.mkIde (set (b, a))*
using *assms S.mkIde-as-mkArr set-subset-Univ small-homs set-def* **by** *simp*
finally show *?thesis* **by** *blast*
qed

lemma *set-map*:
assumes *C.ide a* **and** *C.ide b*
shows *S.set* (*map* (*b*, *a*)) = *set* (*b*, *a*)
using *assms map-ide set-subset-Univ small-homs set-def* **by** *simp*

The definition does in fact yield a functor.

sublocale *functor CopxC.comp S map*
proof
show $\bigwedge gf.\ \neg$ *CopxC.arr gf* $\Longrightarrow$ *map gf* = *S.null*
using *map-def* **by** *auto*
fix *gf*
assume *gf*: *CopxC.arr gf*
thus *arr*: *S.arr* (*map gf*) **using** *gf arr-map* **by** *blast*
show *S.dom* (*map gf*) = *map* (*CopxC.dom gf*)
using *arr gf local.map-def map-ide* **by** *auto*
show *S.cod* (*map gf*) = *map* (*CopxC.cod gf*)
using *gf set-subset-Univ ψ-mapsto map-def set-def S.arr-mkIde arr map-ide* **by** *auto*
next
fix *gf gf′*
assume *gf′*: *CopxC.seq gf′ gf*
hence *seq*: *C.arr* (*fst gf*) ∧ *C.arr* (*snd gf*) ∧ *C.dom* (*snd gf′*) = *C.cod* (*snd gf*) ∧
C.arr (*fst gf′*) ∧ *C.arr* (*snd gf′*) ∧ *C.dom* (*fst gf*) = *C.cod* (*fst gf′*)
by (*elim CopxC.seqE C.seqE*, *auto*)
have *0*: *S.arr* (*map* (*CopxC.comp gf′ gf*))
using *gf′ arr-map* **by** *blast*
have *1*: *map* (*gf′* ⊙ *gf*) =
S.mkArr (*set* (*CopxC.dom gf*)) (*set* (*CopxC.cod gf′*))
(φ (*CopxC.cod gf′*) *o* (λh. *snd* (*gf′* ⊙ *gf*) · *h* · *fst* (*gf′* ⊙ *gf*))
o ψ (*CopxC.dom gf*))
using *gf′ map-def* **using** *CopxC.cod-comp CopxC.dom-comp* **by** *auto*
also have ... = *S.mkArr* (*set* (*CopxC.dom gf*)) (*set* (*CopxC.cod gf′*))
(φ (*CopxC.cod gf′*) ∘ (λh. *snd gf′* · *h* · *fst gf′*) ∘ ψ (*CopxC.dom gf′*)
∘
(φ (*CopxC.cod gf*) ∘ (λh. *snd gf* · *h* · *fst gf*) ∘ ψ (*CopxC.dom gf*)))
proof (*intro S.mkArr-eqI′*)
show *S.arr* (*S.mkArr* (*set* (*CopxC.dom gf*)) (*set* (*CopxC.cod gf′*))
(φ (*CopxC.cod gf′*) ∘ (λh. *snd* (*gf′* ⊙ *gf*) · *h* · *fst* (*gf′* ⊙ *gf*))
∘ ψ (*CopxC.dom gf*)))
using *0 1* **by** *simp*
show $\bigwedge x.\ x \in$ *set* (*CopxC.dom gf*) $\Longrightarrow$
(φ (*CopxC.cod gf′*) ∘ (λh. *snd* (*gf′* ⊙ *gf*) · *h* · *fst* (*gf′* ⊙ *gf*)) ∘
ψ (*CopxC.dom gf*)) *x* =
(φ (*CopxC.cod gf′*) ∘ (λh. *snd gf′* · *h* · *fst gf′*) ∘ ψ (*CopxC.dom gf′*) ∘
(φ (*CopxC.cod gf*) ∘ (λh. *snd gf* · *h* · *fst gf*) ∘ ψ (*CopxC.dom gf*))) *x*
using *gf′ ψ-mapsto set-def ψ-φ C.comp-assoc* **by** *fastforce*
qed
also have ... = *map gf′* ·$_S$ *map gf*
using *seq gf′ map-def arr-map* [*of gf*] *arr-map* [*of gf′*] *S.comp-mkArr* **by** *auto*
finally show *map* (*gf′* ⊙ *gf*) = *map gf′* ·$_S$ *map gf*

using *seq gf'* **by** *auto*
qed

lemma *is-functor*:
shows *functor CopxC.comp S map* **..**

sublocale *binary-functor Cop.comp C S map* **..**

lemma *is-binary-functor*:
shows *binary-functor Cop.comp C S map* **..**

The map φ determines a bijection between $C.hom\ b\ a$ and $set\ (b,\ a)$ which is natural in $(b,\ a)$.

lemma φ*-local-bij*:
assumes $C.ide\ b$ **and** $C.ide\ a$
shows *bij-betw* $(\varphi\ (b,\ a))\ (C.hom\ b\ a)\ (set\ (b,\ a))$
using *assms local-inj inj-on-imp-bij-betw set-def* **by** *auto*

lemma φ*-natural*:
assumes $C.arr\ g$ **and** $C.arr\ f$ **and** $h \in C.hom\ (C.cod\ g)\ (C.dom\ f)$
shows $\varphi\ (C.dom\ g,\ C.cod\ f)\ (f \cdot h \cdot g) = S.Fun\ (map\ (g,\ f))\ (\varphi\ (C.cod\ g,\ C.dom\ f)\ h)$
proof −
let $?\varphi h = \varphi\ (C.cod\ g,\ C.dom\ f)\ h$
have φh: $?\varphi h \in set\ (C.cod\ g,\ C.dom\ f)$
using *assms* φ*-mapsto set-def* **by** *simp*
have *gf*: $CopxC.arr\ (g,\ f)$ **using** *assms* **by** *simp*
have $S.Fun\ (map\ (g,\ f))\ ?\varphi h =$
$(\varphi\ (C.dom\ g,\ C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g,\ C.dom\ f))\ ?\varphi h$
proof −
have $S.Fun\ (map\ (g,\ f)) =$
$restrict\ (\varphi\ (C.dom\ g,\ C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g,\ C.dom\ f))$
$(set\ (C.cod\ g,\ C.dom\ f))$
proof −
have $map\ (g,\ f) =$
$S.mkArr\ (set\ (C.cod\ g,\ C.dom\ f))\ (set\ (C.dom\ g,\ C.cod\ f))$
$(\varphi\ (C.dom\ g,\ C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g,\ C.dom\ f))$
using *assms map-def* **by** *simp*
moreover have $S.arr\ (map\ (g,\ f))$ **using** *gf* **by** *simp*
ultimately show *?thesis*
using *S.Fun-mkArr* **by** *simp*
qed
thus *?thesis*
using φh **by** *simp*
qed
also have ... $= \varphi\ (C.dom\ g,\ C.cod\ f)\ (f \cdot h \cdot g)$
using *assms*(*3*) **by** *simp*
finally show *?thesis* **by** *auto*
qed

```
lemma Dom-map:
assumes C.arr g and C.arr f
shows S.Dom (map (g, f)) = set (C.cod g, C.dom f)
  using assms map-def preserves-arr by auto

lemma Cod-map:
assumes C.arr g and C.arr f
shows S.Cod (map (g, f)) = set (C.dom g, C.cod f)
  using assms map-def preserves-arr by auto

lemma Fun-map:
assumes C.arr g and C.arr f
shows S.Fun (map (g, f)) =
        restrict (φ (C.dom g, C.cod f) o (λh. f · h · g) o ψ (C.cod g, C.dom f))
                 (set (C.cod g, C.dom f))
  using assms map-def preserves-arr by force

lemma map-simp-1:
assumes C.arr g and C.ide a
shows map (g, a) = S.mkArr (set (C.cod g, a)) (set (C.dom g, a))
                           (φ (C.dom g, a) o Cop.comp g o ψ (C.cod g, a))
proof −
  have 1: map (g, a) = S.mkArr (set (C.cod g, a)) (set (C.dom g, a))
                               (φ (C.dom g, a) o (λh. a · h · g) o ψ (C.cod g, a))
    using assms map-def by force
  also have ... = S.mkArr (set (C.cod g, a)) (set (C.dom g, a))
                          (φ (C.dom g, a) o Cop.comp g o ψ (C.cod g, a))
    using assms 1 preserves-arr [of (g, a)] set-def C.in-homI C.comp-cod-arr
    by (intro S.mkArr-eqI) auto
 finally show ?thesis by blast
qed

lemma map-simp-2:
assumes C.ide b and C.arr f
shows map (b, f) = S.mkArr (set (b, C.dom f)) (set (b, C.cod f))
                           (φ (b, C.cod f) o C f o ψ (b, C.dom f))
proof −
  have 1: map (b, f) = S.mkArr (set (b, C.dom f)) (set (b, C.cod f))
                               (φ (b, C.cod f) o (λh. f · h · b) o ψ (b, C.dom f))
    using assms map-def by force
  also have ... = S.mkArr (set (b, C.dom f)) (set (b, C.cod f))
                          (φ (b, C.cod f) o C f o ψ (b, C.dom f))
    using assms 1 preserves-arr [of (b, f)] set-def C.in-homI C.comp-arr-dom
    by (intro S.mkArr-eqI) auto
  finally show ?thesis by blast
qed

end
```

Every category C has a hom-functor: take S to be the replete set category generated

by the arrow type $'a$ of C and take $\varphi\ (b,\ a)$ to be the map $S.UP :: 'a \Rightarrow 'a\ SC.arr$.

context *category*
begin

interpretation S: *replete-setcat* ‹*TYPE*('*a*)› **.**

lemma *has-hom-functor*:
shows *hom-functor C S.comp S.setp* (λ-. *S.UP*)
using *S.UP-mapsto S.inj-UP injD inj-onI*
by *unfold-locales* (*auto simp add*: *inj-def inj-onI*)

end

The locales *set-valued-functor* and *set-valued-transformation* provide some abbreviations that are convenient when working with functors and natural transformations into a set category.

locale *set-valued-functor* =
C: *category C* +
S: *set-category S setp* +
functor C S F
for $C :: 'c\ comp$
and $S :: 's\ comp$
and $setp :: 's\ set \Rightarrow bool$
and $F :: 'c \Rightarrow 's$
begin

abbreviation $SET :: 'c \Rightarrow 's\ set$
where $SET\ a \equiv S.set\ (F\ a)$

abbreviation $DOM :: 'c \Rightarrow 's\ set$
where $DOM\ f \equiv S.Dom\ (F\ f)$

abbreviation $COD :: 'c \Rightarrow 's\ set$
where $COD\ f \equiv S.Cod\ (F\ f)$

abbreviation $FUN :: 'c \Rightarrow 's \Rightarrow 's$
where $FUN\ f \equiv S.Fun\ (F\ f)$

end

locale *set-valued-transformation* =
C: *category C* +
S: *set-category S setp* +
F: *set-valued-functor C S setp F* +
G: *set-valued-functor C S setp G* +
natural-transformation C S F G τ
for $C :: 'c\ comp$
and $S :: 's\ comp$
and $setp :: 's\ set \Rightarrow bool$

and $F :: 'c \Rightarrow 's$
and $G :: 'c \Rightarrow 's$
and $\tau :: 'c \Rightarrow 's$
begin

abbreviation $DOM :: 'c \Rightarrow 's\ set$
where $DOM\ f \equiv S.Dom\ (\tau\ f)$

abbreviation $COD :: 'c \Rightarrow 's\ set$
where $COD\ f \equiv S.Cod\ (\tau\ f)$

abbreviation $FUN :: 'c \Rightarrow 's \Rightarrow 's$
where $FUN\ f \equiv S.Fun\ (\tau\ f)$

end

15.2 Yoneda Functors

A Yoneda functor is the functor from C to $[Cop, S]$ obtained by "currying" a hom-functor in its first argument.

locale *yoneda-functor* =
C: *category C* +
Cop: *dual-category C* +
CopxC: *product-category Cop.comp C* +
S: *set-category S setp* +
Hom: *hom-functor C S setp* φ
for $C :: 'c\ comp$ (**infixr** $\langle\cdot\rangle$ *55*)
and $S :: 's\ comp$ (**infixr** $\langle\cdot_S\rangle$ *55*)
and $setp :: 's\ set \Rightarrow bool$
and $\varphi :: 'c * 'c \Rightarrow 'c \Rightarrow 's$
begin

sublocale *Cop-S*: *functor-category Cop.comp S* ..
sublocale *curried-functor′ Cop.comp C S Hom.map* ..

notation *Cop-S.in-hom* ($\langle\langle$- : - $\rightarrow_{[Cop,S]}$ -$\rangle\rangle$)

abbreviation ψ
where $\psi \equiv Hom.\psi$

An arrow of the functor category $[Cop, S]$ consists of a natural transformation bundled together with its domain and codomain functors. However, when considering a Yoneda functor from C to $[Cop, S]$ we generally are only interested in the mapping Y that takes each arrow f of C to the corresponding natural transformation $Y\ f$. The domain and codomain functors are then the identity transformations $Y\ (C.dom\ f)$ and $Y\ (C.cod\ f)$.

definition *Y*

where $Y\ f \equiv Cop\text{-}S.Map\ (map\ f)$

lemma *Y-simp* [*simp*]:
assumes $C.arr\ f$
shows $Y\ f = (\lambda g.\ Hom.map\ (g,\ f))$
using *assms preserves-arr Y-def* **by** *simp*

lemma *Y-ide-is-functor*:
assumes $C.ide\ a$
shows *functor Cop.comp S* $(Y\ a)$
using *assms Y-def Hom.fixing-ide-gives-functor-2* **by** *force*

lemma *Y-arr-is-transformation*:
assumes $C.arr\ f$
shows *natural-transformation Cop.comp S* $(Y\ (C.dom\ f))\ (Y\ (C.cod\ f))\ (Y\ f)$
using *assms Y-def* [*of f*] *map-def Hom.fixing-arr-gives-natural-transformation-2*
preserves-dom preserves-cod **by** *fastforce*

lemma *Y-ide-arr* [*simp*]:
assumes *a*: $C.ide\ a$ **and** $«g : b' \to b»$
shows $«Y\ a\ g : Hom.map\ (b,\ a) \to_S Hom.map\ (b',\ a)»$
and $Y\ a\ g = S.mkArr\ (Hom.set\ (b,\ a))\ (Hom.set\ (b',\ a))\ (\varphi\ (b',\ a)\ o\ Cop.comp\ g\ o\ \psi\ (b,\ a))$
using *assms Hom.map-simp-1* **by** (*fastforce*, *auto*)

lemma *Y-arr-ide* [*simp*]:
assumes $C.ide\ b$ **and** $«f : a \to a'»$
shows $«Y\ f\ b : Hom.map\ (b,\ a) \to_S Hom.map\ (b,\ a')»$
and $Y\ f\ b = S.mkArr\ (Hom.set\ (b,\ a))\ (Hom.set\ (b,\ a'))\ (\varphi\ (b,\ a')\ o\ C\ f\ o\ \psi\ (b,\ a))$
using *assms* **apply** *fastforce*
using *assms Hom.map-simp-2* **by** *auto*

end

locale *yoneda-functor-fixed-object* =
yoneda-functor +
fixes *a*
assumes *ide-a*: $C.ide\ a$
begin

sublocale *functor Cop.comp S* $‹Y\ a›$
using *ide-a Y-ide-is-functor* **by** *auto*
sublocale *set-valued-functor Cop.comp S setp* $‹Y\ a›$ **..**

end

The Yoneda lemma states that, given a category C and a functor F from Cop to a set category S, for each object a of C, the set of natural transformations from the contravariant functor $Y\ a$ to F is in bijective correspondence with the set $F.SET\ a$ of

elements of $F\ a$.

Explicitly, if e is an arbitrary element of the set $F.SET\ a$, then the functions $\lambda x.\ F.FUN\ (\psi\ (b,\ a)\ x)\ e$ are the components of a natural transformation from $Y\ a$ to F. Conversely, if τ is a natural transformation from $Y\ a$ to F, then the component $\tau\ b$ of τ at an arbitrary object b is completely determined by the single arrow $\tau.FUN\ a\ (\varphi\ (a,\ a)\ a)))$, which is the the element of $F.SET\ a$ that corresponds to the image of the identity a under the function $\tau.FUN\ a$. Then $\tau\ b$ is the arrow from $Y\ a\ b$ to $F\ b$ corresponding to the function $\lambda x.\ (F.FUN\ (\psi\ (b,\ a)\ x)\ (\tau.FUN\ a\ (\varphi\ (a,\ a)\ a)))$ from $S.set\ (Y\ a\ b)$ to $F.SET\ b$.

The above expressions look somewhat more complicated than the usual versions due to the need to account for the coercions φ and ψ.

```
locale yoneda-lemma =
  yoneda-functor-fixed-object C S setp φ a +
  F: set-valued-functor Cop.comp S setp F
for C :: 'c comp (infixr ‹·› 55)
and S :: 's comp (infixr ‹·S› 55)
and setp :: 's set ⇒ bool
and φ :: 'c * 'c ⇒ 'c ⇒ 's
and F :: 'c ⇒ 's
and a :: 'c
begin
```

The mapping that evaluates the component $\tau\ a$ at a of a natural transformation τ from Y to F on the element $\varphi\ (a,\ a)\ a$ of $SET\ a$, yielding an element of $F.SET\ a$.

```
  definition ℰ :: ('c ⇒ 's) ⇒ 's
  where ℰ τ = S.Fun (τ a) (φ (a, a) a)
```

The mapping that takes an element e of $F.SET\ a$ and produces a map on objects of C whose value at b is the arrow of S corresponding to the function $(\lambda x.\ F.FUN\ (\psi\ (b,\ a)\ x)\ e) \in Hom.set\ (b,\ a) \to F.SET\ b$.

```
  definition 𝒯o :: 's ⇒ 'c ⇒ 's
  where 𝒯o e b = S.mkArr (Hom.set (b, a)) (F.SET b) (λx. F.FUN (ψ (b, a) x) e)

  lemma 𝒯o-in-hom:
  assumes e: e ∈ S.set (F a) and b: C.ide b
  shows «𝒯o e b : Y a b →S F b»
  proof −
    have (λx. F.FUN (ψ (b, a) x) e) ∈ Hom.set (b, a) → F.SET b
    proof
      fix x
      assume x: x ∈ Hom.set (b, a)
      thus F.FUN (ψ (b, a) x) e ∈ F.SET b
        using assms e ide-a Hom.ψ-mapsto S.Fun-mapsto [of F (ψ (b, a) x)] by force
    qed
    thus ?thesis
      using ide-a b S.mkArr-in-hom Hom.set-subset-Univ S.mkIde-set 𝒯o-def
      by (metis C.ideD(1) Cop.ide-char F.preserves-ide Hom.set-map S.setp-set-ide
```

preserves-ide Y-simp)
qed

For each $e \in F.SET\ a$, the mapping $\mathcal{T}_o\ e$ gives the components of a natural transformation $\mathcal{T}$ from $Y\ a$ to F.

lemma $\mathcal{T}_o$*-induces-transformation*:
assumes *e*: $e \in S.set\ (F\ a)$
shows *transformation-by-components Cop.comp S* $(Y\ a)\ F\ (\mathcal{T}_o\ e)$
proof
show $\bigwedge b.\ Cop.ide\ b \Longrightarrow \langle\langle \mathcal{T}_o\ e\ b : Y\ a\ b \rightarrow_S F\ b \rangle\rangle$
using *ide-a e* $\mathcal{T}_o$*-in-hom* **by** *simp*
fix $g :: 'c$
assume *g*: *Cop.arr g*
let $?b = Cop.dom\ g$
let $?b' = Cop.cod\ g$
show $\mathcal{T}_o\ e\ (Cop.cod\ g) \cdot_S Y\ a\ g = F\ g \cdot_S \mathcal{T}_o\ e\ (Cop.dom\ g)$
proof −
have *1*: $\mathcal{T}_o\ e\ (Cop.cod\ g) \cdot_S Y\ a\ g =$
$S.mkArr\ (Hom.set\ (?b,\ a))\ (F.SET\ ?b')$
$((\lambda x.\ F.FUN\ (\psi\ (?b',\ a)\ x)\ e)\ o\ (\varphi\ (?b',\ a)\ o\ Cop.comp\ g\ o\ \psi\ (?b,\ a)))$
proof −
have $S.arr\ (S.mkArr\ (Hom.set\ (?b',\ a))\ (F.SET\ ?b')\ (\lambda s.\ F.FUN\ (\psi\ (?b',\ a)\ s)\ e)) \wedge$
$S.dom\ (S.mkArr\ (Hom.set\ (?b',\ a))\ (F.SET\ ?b')\ (\lambda s.\ F.FUN\ (\psi\ (?b',\ a)\ s)\ e))$
$= Y\ a\ ?b' \wedge$
$S.cod\ (S.mkArr\ (Hom.set\ (?b',\ a))\ (F.SET\ ?b')\ (\lambda s.\ F.FUN\ (\psi\ (?b',\ a)\ s)\ e))$
$= F\ ?b'$
using *Cop.cod-char* $\mathcal{T}_o$*-def* $\mathcal{T}_o$*-in-hom e g*
by (*metis Cop.ide-char Cop.ide-cod S.in-homE*)
moreover have $Y\ a\ g = S.mkArr\ (Hom.set\ (?b,\ a))\ (Hom.set\ (?b',\ a))$
$(\varphi\ (?b',\ a) \circ Cop.comp\ g \circ \psi\ (?b,\ a))$
using *Y-ide-arr* [*of a g ?b' ?b*] *ide-a g* **by** *auto*
ultimately show *?thesis*
using *ide-a e g Y-ide-arr Cop.cod-char* $\mathcal{T}_o$*-def S.comp-mkArr preserves-arr*
by *metis*
qed
also have ... $= S.mkArr\ (Hom.set\ (?b,\ a))\ (F.SET\ ?b')$
$(F.FUN\ g\ o\ (\lambda x.\ F.FUN\ (\psi\ (?b,\ a)\ x)\ e))$
proof (*intro S.mkArr-eqI'*)
show $S.arr\ (S.mkArr\ (Hom.set\ (?b,\ a))\ (F.SET\ ?b')$
$((\lambda x.\ F.FUN\ (\psi\ (?b',\ a)\ x)\ e)$
$o\ (\varphi\ (?b',\ a)\ o\ Cop.comp\ g\ o\ \psi\ (?b,\ a))))$
proof (*intro S.arr-mkArrI*)
show *setp* $(Hom.set\ (Cop.dom\ g,\ a))$
by (*metis C.ideD*(*1*) *Cop.arr-dom Cop.ide-char CopxC.arrI*$_{PC}$ *Hom.arr-map*
S.arr-mkIde Cop.ide-dom g Hom.map-ide ide-a)
show *setp* $(F.SET\ (Cop.cod\ g))$
using *g* **by** *force*
show $(\lambda x.\ F.FUN\ (\psi\ (?b',\ a)\ x)\ e)\ o\ (\varphi\ (?b',\ a)\ o\ Cop.comp\ g\ o\ \psi\ (?b,\ a))$
$\in Hom.set\ (?b,\ a) \rightarrow F.SET\ ?b'$

```
      proof −
        have S.arr (S (𝒯_o e ?b′) (Y a g))
          using ide-a e g 𝒯_o-in-hom Y-ide-arr(1) Cop.ide-char Cop.ide-cod by blast
        thus ?thesis using 1 by simp
      qed
    qed
    show ⋀x. x ∈ Hom.set (?b, a) ⟹
                ((λx. F.FUN (ψ (?b′, a) x) e) o (φ (?b′, a) o Cop.comp g o ψ (?b, a))) x
                = (F.FUN g o (λx. F.FUN (ψ (?b, a) x) e)) x
    proof −
      fix x
      assume x: x ∈ Hom.set (?b, a)
      have ((λx. (F.FUN o ψ (?b′, a)) x e)
                o (φ (?b′, a) o Cop.comp g o ψ (?b, a))) x
              = F.FUN (ψ (?b′, a)) (φ (?b′, a)) (C (ψ (?b, a) x) g))) e
        by simp
      also have ... = (F.FUN g o (F.FUN o ψ (?b, a)) x) e
      proof −
        have «ψ (?b, a) x : ?b → a»
          using ide-a x g Hom.ψ-mapsto [of ?b a] by auto
        thus ?thesis
          using assms g Hom.ψ-φ F.preserves-comp by fastforce
      qed
      also have ... = (F.FUN g o (λx. F.FUN (ψ (?b, a) x) e)) x by fastforce
     finally show ((λx. F.FUN (ψ (?b′, a) x) e) o (φ (?b′, a) o Cop.comp g o ψ (?b, a))) x
                    = (F.FUN g o (λx. F.FUN (ψ (?b, a) x) e)) x
        by simp
    qed
  qed
  also have ... = F g ·_S 𝒯_o e ?b
  proof −
    have S.arr (F g) ∧ F g = S.mkArr (F.SET ?b) (F.SET ?b′) (F.FUN g)
      using g S.mkArr-Fun [of F g] by simp
    moreover have
       S.arr (𝒯_o e ?b) ∧
        𝒯_o e ?b = S.mkArr (Hom.set (?b, a)) (F.SET ?b) (λx. F.FUN (ψ (?b, a) x) e)
      using e g 𝒯_o-def 𝒯_o-in-hom
      by (metis C.ide-cod Cop.arr-char Cop.dom-char S.in-homE)
    ultimately show ?thesis
      using S.comp-mkArr by metis
  qed
  finally show ?thesis by blast
 qed
qed

definition 𝒯 :: 's ⇒ 'c ⇒ 's
where 𝒯 e ≡ transformation-by-components.map Cop.comp S (Y a) (𝒯_o e)

end
```

locale *yoneda-lemma-fixed-e* =
yoneda-lemma +
fixes *e*
assumes *E*: $e \in F.SET\ a$
begin

interpretation $\mathcal{T}e$: *transformation-by-components Cop.comp S* ‹*Y a*› *F* ‹$\mathcal{T}_o$ *e*›
using *E* $\mathcal{T}_o$*-induces-transformation* **by** *auto*
sublocale $\mathcal{T}e$: *natural-transformation Cop.comp S* ‹*Y a*› *F* ‹$\mathcal{T}$ *e*›
unfolding $\mathcal{T}$*-def* **..**

lemma $\mathcal{T}e$*-ide*:
assumes *Cop.ide b*
shows *S.arr* ($\mathcal{T}$ *e b*)
and $\mathcal{T}$ *e b* = *S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*) ($\lambda x.$ *F.FUN* (ψ (*b*, *a*) *x*) *e*)
using *assms* **apply** *auto*[*1*]
using *assms* $\mathcal{T}_o$*-def* $\mathcal{T}$*-def* **by** *auto*

end

locale *yoneda-lemma-fixed-*τ =
yoneda-lemma +
τ: *natural-transformation Cop.comp S* ‹*Y a*› *F* τ
for τ
begin

sublocale τ: *set-valued-transformation Cop.comp S setp* ‹*Y a*› *F* τ **..**

The key lemma: The component $\tau\ b$ of τ at an arbitrary object b is completely determined by the single element $\tau.FUN\ a\ (\varphi\ (a,\ a)\ a) \in F.SET\ a$.

lemma τ*-ide*:
assumes *b*: *Cop.ide b*
shows $\tau\ b$ = *S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*)
($\lambda x.$ (*F.FUN* (ψ (*b*, *a*) *x*) (τ*.FUN a* (φ (*a*, *a*) *a*))))
proof −
let *?*φa = φ (*a*, *a*) *a*
have φa: φ (*a*, *a*) *a* ∈ *Hom.set* (*a*, *a*) **using** *ide-a Hom.*φ*-mapsto* **by** *fastforce*
have *1*: $\tau\ b$ = *S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*) (τ*.FUN b*)
using *ide-a b S.mkArr-Fun* [*of* τ *b*] *Hom.set-map* **by** *auto*
also have
... = *S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*) ($\lambda x.$ (*F.FUN* (ψ (*b*, *a*) *x*) (τ*.FUN a ?*φa)))
proof (*intro S.mkArr-eqI'*)
show *2*: *S.arr* (*S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*) (τ*.FUN b*))
using *ide-a b 1 S.mkArr-Fun* [*of* τ *b*] *Hom.set-map* **by** *auto*
show $\bigwedge x.\ x \in$ *Hom.set* (*b*, *a*) $\Longrightarrow$ τ*.FUN b x* = (*F.FUN* (ψ (*b*, *a*) *x*) (τ*.FUN a ?*φa))

```
proof −
  fix x
  assume x: x ∈ Hom.set (b, a)
  let ?ψx = ψ (b, a) x
  have ψx: «?ψx : b → a»
    using ide-a b x Hom.ψ-mapsto [of b a] by auto
  show τ.FUN b x = (F.FUN (ψ (b, a) x) (τ.FUN a ?φa))
  proof −
    have τ.FUN b x = S.Fun (τ b ·S Y a ?ψx) ?φa
    proof −
      have τ.FUN b x = τ.FUN b ((φ (b, a) o Cop.comp ?ψx) a)
        using ide-a b x ψx Hom.φ-ψ
        by (metis C.comp-cod-arr C.in-homE C.ide-dom Cop.comp-def comp-apply)
      also have ... = (τ.FUN b o (φ (b, a) o Cop.comp ?ψx o ψ (a, a))) ?φa
        using ide-a b C.ide-in-hom by simp
      also have ... = S.Fun (τ b ·S Y a ?ψx) ?φa
      proof −
        have S.seq (τ b) (Y a ?ψx) ∧
             τ b ·S Y a ?ψx =
               S.mkArr (Hom.set (a, a)) (F.SET b)
                       (τ.FUN b o (φ (b, a) ∘ Cop.comp ?ψx ∘ ψ (a, a)))
        proof
          show S.seq (τ b) (Y a ?ψx)
            using ψx τ.naturality2 by fastforce
          show τ b ·S Y a ?ψx =
                 S.mkArr (Hom.set (a, a)) (F.SET b)
                         (τ.FUN b o (φ (b, a) ∘ Cop.comp ?ψx ∘ ψ (a, a)))
            by (metis 1 2 Cop.arrI Cop.hom-char S.comp-mkArr Y-ide-arr(2)
                ψx ide-a preserves-arr)
        qed
        thus ?thesis
          using ide-a b x Hom.φ-mapsto S.Fun-mkArr by force
      qed
      finally show ?thesis by auto
    qed
    also have ... = S.Fun (F ?ψx ·S τ a) ?φa
      using ide-a b ψx τ.naturality by force
    also have ... = F.FUN ?ψx (τ.FUN a ?φa)
    proof −
      have restrict (S.Fun (F ?ψx ·S τ a)) (Hom.set (a, a))
                    = restrict (F.FUN (ψ (b, a) x) o τ.FUN a) (Hom.set (a, a))
      proof −
        have S.arr (F ?ψx ·S τ a) ∧
             F ?ψx ·S τ a = S.mkArr (Hom.set (a, a)) (F.SET b) (F.FUN ?ψx o τ.FUN a)
        proof
          show 1: S.seq (F ?ψx) (τ a)
            using ψx ide-a τ.preserves-cod F.preserves-dom
            by (elim C.in-homE, auto)
          show F ?ψx ·S τ a = S.mkArr (Hom.set (a, a)) (F.SET b) (F.FUN ?ψx o τ.FUN
```

```
a)
            proof −
              have τ a = S.mkArr (Hom.set (a, a)) (F.SET a) (τ.FUN a)
                using ide-a 1 S.mkArr-Fun [of τ a] Hom.set-map by auto
              moreover have F ?ψx = S.mkArr (F.SET a) (F.SET b) (F.FUN ?ψx)
                using x ψx 1 S.mkArr-Fun [of F ?ψx] by fastforce
              ultimately show ?thesis
                using 1 S.comp-mkArr [of Hom.set (a, a) F.SET a τ.FUN a
                                         F.SET b F.FUN ?ψx]
                by (elim S.seqE, auto)
            qed
          qed
          thus ?thesis by force
        qed
        thus S.Fun (F (ψ (b, a) x) ·S τ a) ?φa = F.FUN ?ψx (τ.FUN a ?φa)
          using ide-a φa restr-eqE [of S.Fun (F ?ψx ·S τ a)
                                       Hom.set (a, a) F.FUN ?ψx o τ.FUN a]
          by simp
      qed
      finally show ?thesis by simp
    qed
  qed
  finally show ?thesis by auto
qed
```

Consequently, if τ' is any natural transformation from $Y\ a$ to F that agrees with τ at a, then $\tau' = \tau$.

```
lemma eqI:
assumes natural-transformation Cop.comp S (Y a) F τ' and τ' a = τ a
shows τ' = τ
proof (intro natural-transformation-eqI)
  interpret τ': natural-transformation Cop.comp S ‹Y a› F τ' using assms by auto
  interpret T': yoneda-lemma-fixed-τ C S setp φ F a τ' ..
  show natural-transformation Cop.comp S (Y a) F τ ..
  show natural-transformation Cop.comp S (Y a) F τ' ..
  show ⋀b. Cop.ide b ⟹ τ' b = τ b
    using assms(2) τ-ide T'.τ-ide by simp
qed

end

context yoneda-lemma
begin
```

One half of the Yoneda lemma: The mapping $\mathcal{T}$ is an injection, with left inverse $\mathcal{E}$, from the set $F.SET\ a$ to the set of natural transformations from $Y\ a$ to F.

```
lemma T-is-injection:
assumes e ∈ F.SET a
```

shows *natural-transformation Cop.comp S* (*Y a*) *F* (*𝒯 e*) **and** *ℰ* (*𝒯 e*) = *e*
proof −
interpret *yoneda-lemma-fixed-e C S setp φ F a e*
using *assms* **by** (*unfold-locales*, *auto*)
show *natural-transformation Cop.comp S* (*Y a*) *F* (*𝒯 e*) **..**
show *ℰ* (*𝒯 e*) = *e*
unfolding *ℰ-def*
using *assms 𝒯e-ide S.Fun-mkArr Hom.φ-mapsto Hom.ψ-φ ide-a*
F.preserves-ide S.Fun-ide restrict-apply C.ide-in-hom
by (*auto simp add*: *Pi-iff*)
qed

lemma *ℰτ-mapsto*:
assumes *natural-transformation Cop.comp S* (*Y a*) *F τ*
shows *ℰ τ* ∈ *F.SET a*
proof −
interpret *τ*: *natural-transformation Cop.comp S* ‹*Y a*› *F τ*
using *assms* **by** *auto*
interpret *yoneda-lemma-fixed-τ C S setp φ F a τ* **..**
show *?thesis*
proof (*unfold ℰ-def*)
have *τ.FUN a* ∈ *Hom.set* (*a*, *a*) → *F.SET a*
proof −
have *S.arr* (*τ a*) ∧ *S.Dom* (*τ a*) = *Hom.set* (*a*, *a*) ∧ *S.Cod* (*τ a*) = *F.SET a*
using *ide-a Hom.set-map* **by** *auto*
thus *?thesis*
using *S.Fun-mapsto* **by** *blast*
qed
thus *τ.FUN a* (*φ* (*a*, *a*) *a*) ∈ *F.SET a*
using *ide-a Hom.φ-mapsto* **by** *fastforce*
qed
qed

The other half of the Yoneda lemma: The mapping *𝒯* is a surjection, with right inverse *ℰ*, taking natural transformations from *Y a* to *F* to elements of *F.SET a*.

lemma *𝒯-is-surjection*:
assumes *natural-transformation Cop.comp S* (*Y a*) *F τ*
shows *𝒯* (*ℰ τ*) = *τ*
proof −
interpret *natural-transformation Cop.comp S* ‹*Y a*› *F τ*
using *assms* **by** *auto*
interpret *yoneda-lemma-fixed-τ C S setp φ F a τ* **..**
interpret *yoneda-lemma-fixed-e C S setp φ F a* ‹*ℰ τ*›
using *assms ℰτ-mapsto* **by** *unfold-locales auto*
show *𝒯* (*ℰ τ*) = *τ*
using *ide-a τ-ide* [*of a*] *𝒯e-ide ℰ-def natural-transformation-𝒯e*
by (*intro eqI*) *auto*
qed

The main result.

theorem *yoneda-lemma*:
shows *bij-betw* $\mathcal{T}$ (*F.SET a*) {τ. *natural-transformation Cop.comp S* (*Y a*) *F* τ}
using $\mathcal{E}\tau$*-mapsto* $\mathcal{T}$*-is-injection* $\mathcal{T}$*-is-surjection*
by (*intro bij-betwI*) *auto*

end

We now consider the special case in which F is the contravariant functor $Y\ a'$. Then for any e in *Hom.set* (a, a') we have $\mathcal{T}\ e = Y\ (\psi\ (a,\ a')\ e)$, and $\mathcal{T}$ is a bijection from *Hom.set* (a, a') to the set of natural transformations from $Y\ a$ to $Y\ a'$. It then follows that that the Yoneda functor Y is a fully faithful functor from C to the functor category [*Cop*, *S*].

locale *yoneda-lemma-for-hom* =
yoneda-functor-fixed-object C S setp φ *a* +
Ya': *yoneda-functor-fixed-object C S setp* φ *a'* +
yoneda-lemma C S setp φ *Y a' a*
for *C* :: *'c comp* (**infixr** ‹·› *55*)
and *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *setp* :: *'s set* ⇒ *bool*
and φ :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
and *a* :: *'c*
and *a'* :: *'c* +
assumes *ide-a'*: *C.ide a'*
begin

In case F is the functor $Y\ a'$, for any $e \in$ *Hom.set* (a, a') the induced natural transformation $\mathcal{T}\ e$ from $Y\ a$ to $Y\ a'$ is just $Y\ (\psi\ (a,\ a')\ e)$.

lemma *app-*$\mathcal{T}$*-equals*:
assumes *e*: *e* ∈ *Hom.set* (*a*, *a'*)
shows $\mathcal{T}\ e = Y\ (\psi\ (a,\ a')\ e)$
proof −
let *?ψe* = ψ (*a*, *a'*) *e*
have *ψe*: «*?ψe* : *a* → *a'*» **using** *ide-a ide-a' e Hom.ψ-mapsto* **by** *auto*
interpret *Ye*: *natural-transformation Cop.comp S* ‹*Y a*› ‹*Y a'*› ‹*Y ?ψe*›
using *Y-arr-is-transformation* [*of ?ψe*] *ψe* **by** (*elim C.in-homE*, *auto*)
interpret *yoneda-lemma-fixed-e C S setp* φ ‹*Y a'*› *a e*
using *ide-a ide-a' e Hom.set-map*
by (*unfold-locales*, *simp-all*)
interpret *yoneda-lemma-fixed-*τ *C S setp* φ ‹*Y a'*› *a* ‹$\mathcal{T}$ *e*› **..**
have *natural-transformation Cop.comp S* (*Y a*) (*Y a'*) (*Y ?ψe*) **..**
moreover have *natural-transformation Cop.comp S* (*Y a*) (*Y a'*) ($\mathcal{T}$ *e*) **..**
moreover have $\mathcal{T}\ e\ a = Y\ ?\psi e\ a$
proof −
have *1*: $\mathcal{T}\ e\ a$ = *S.mkArr* (*Hom.set* (*a*, *a*)) (*Ya'.SET a*) (λ*x*. *Ya'.FUN* (ψ (*a*, *a*) *x*) *e*)
using *ide-a* $\mathcal{T}_o$*-def* $\mathcal{T}$*e-ide* **by** *simp*
also have
... = *S.mkArr* (*Hom.set* (*a*, *a*)) (*Hom.set* (*a*, *a'*)) (φ (*a*, *a'*) *o C ?ψe o ψ* (*a*, *a*))
proof (*intro S.mkArr-eqI*)
show *S.arr* (*S.mkArr* (*Hom.set* (*a*, *a*)) (*Ya'.SET a*) (λ*x*. *Ya'.FUN* (ψ (*a*, *a*) *x*) *e*))

```
      using ide-a e 1 𝒯e.preserves-reflects-arr
      by (metis Cop.ide-char 𝒯e-ide(1))
    show Hom.set (a, a) = Hom.set (a, a) ..
    show 2: Ya'.SET a = Hom.set (a, a')
      using ide-a ide-a' Y-simp Hom.set-map by simp
    show ⋀x. x ∈ Hom.set (a, a) ⟹
              Ya'.FUN (ψ (a, a) x) e = (φ (a, a') o C ?ψe o ψ (a, a)) x
    proof −
      fix x
      assume x: x ∈ Hom.set (a, a)
      have ψx: «ψ (a, a) x : a → a»
        using ide-a x Hom.ψ-mapsto [of a a] by auto
      have S.arr (Y a' (ψ (a, a) x)) ∧
           Y a' (ψ (a, a) x) = S.mkArr (Hom.set (a, a')) (Hom.set (a, a'))
                                 (φ (a, a') ∘ Cop.comp (ψ (a, a) x) ∘ ψ (a, a'))
        using Y-ide-arr ide-a ide-a' ψx by blast
      hence Ya'.FUN (ψ (a, a) x) e = (φ (a, a') ∘ Cop.comp (ψ (a, a) x) ∘ ψ (a, a')) e
        using e 2 S.Fun-mkArr Ya'.preserves-reflects-arr [of ψ (a, a) x] by simp
      also have ... = (φ (a, a') o C ?ψe o ψ (a, a)) x by simp
      finally show Ya'.FUN (ψ (a, a) x) e = (φ (a, a') o C ?ψe o ψ (a, a)) x by auto
    qed
  qed
  also have ... = Y ?ψe a
    using ide-a ide-a' Y-arr-ide ψe by simp
  finally show 𝒯 e a = Y ?ψe a by auto
 qed
 ultimately show ?thesis using eqI by auto
qed

lemma is-injective-on-homs:
shows inj-on map (C.hom a a')
proof (intro inj-onI)
 fix f f'
 assume f: f ∈ C.hom a a' and f': f' ∈ C.hom a a'
 assume eq: map f = map f'
 show f = f'
 proof −
  have f = ψ (a, a') (ℰ (Y (ψ (a, a') (φ (a, a') f))))
    by (metis (no-types, lifting) C.comp-arr-dom C.ide-in-hom Hom.φ-natural
        Hom.ψ-φ ℰ-def category.in-homE f ide-a mem-Collect-eq
        Y-simp yoneda-functor-axioms yoneda-functor-def)
  also have ... = ψ (a, a') (ℰ (𝒯 (φ (a, a') f')))
    using f f' eq Hom.φ-mapsto [of a a'] ide-a Hom.ψ-φ Y-def
          app-𝒯-equals [of φ (a, a') f']
    by fastforce
  also have ... = f'
    by (metis C.ideD(1) Hom.φ-mapsto Hom.ψ-φ Hom.set-map PiE Y-simp
        𝒯-is-injection(2) f' ide-a ide-a' mem-Collect-eq)
  finally show ?thesis by auto
```

```
    qed
  qed

end

context yoneda-functor
begin

  sublocale faithful-functor C Cop-S.comp map
  proof
    fix f :: 'c and f' :: 'c
    assume par: C.par f f' and ff': map f = map f'
    show f = f'
    proof −
      interpret Ya': yoneda-functor-fixed-object C S setp φ ‹C.cod f›
        using par by (unfold-locales, auto)
      interpret yoneda-lemma-for-hom C S setp φ ‹C.dom f› ‹C.cod f›
        using par by (unfold-locales, auto)
      show f = f'
        using par ff' is-injective-on-homs inj-on-def [of map C.hom (C.dom f) (C.cod f)]
        by force
    qed
  qed

  lemma is-faithful-functor:
  shows faithful-functor C Cop-S.comp map
    ..

  sublocale full-functor C Cop-S.comp map
  proof
    fix a :: 'c and a' :: 'c and t
    assume a: C.ide a and a': C.ide a'
    assume t: «t : map a →[Cop,S] map a'»
    show ∃ e. «e : a → a'» ∧ map e = t
    proof
      interpret Ya': yoneda-functor-fixed-object C S setp φ a'
        using a' by (unfold-locales, auto)
      interpret yoneda-lemma-for-hom C S setp φ a a'
        using a a' by (unfold-locales, auto)
      have NT: natural-transformation Cop.comp S (Y a) (Y a') (Cop-S.Map t)
        using t a' Y-def Cop-S.Map-dom Cop-S.Map-cod Cop-S.dom-char Cop-S.cod-char
              Cop-S.in-homE Cop-S.arrE
        by metis
      hence 1: ℰ (Cop-S.Map t) ∈ Hom.set (a, a')
        using ℰτ-mapsto ide-a ide-a' Hom.set-map by simp
      moreover have map (ψ (a, a') (ℰ (Cop-S.Map t))) = t
      proof (intro Cop-S.arr-eqI)
        have 2: «map (ψ (a, a') (ℰ (Cop-S.Map t))) : map a →[Cop,S] map a'»
          using 1 ide-a ide-a' Hom.ψ-mapsto [of a a'] by blast
```

```
        show Cop-S.arr t using t by blast
        show Cop-S.arr (map (ψ (a, a') (ℰ (Cop-S.Map t)))) using 2 by blast
        show 3: Cop-S.Map (map (ψ (a, a') (ℰ (Cop-S.Map t)))) = Cop-S.Map t
          using NT 1 Y-def 𝒯-is-surjection app-𝒯-equals ℰτ-mapsto by metis
        show 4: Cop-S.Dom (map (ψ (a, a') (ℰ (Cop-S.Map t)))) = Cop-S.Dom t
          using t 2 functor-axioms Cop-S.Map-dom by (metis Cop-S.in-homE)
        show Cop-S.Cod (map (ψ (a, a') (ℰ (Cop-S.Map t)))) = Cop-S.Cod t
          using 2 3 4 t Cop-S.Map-cod by (metis Cop-S.in-homE)
      qed
      ultimately show «ψ (a, a') (ℰ (Cop-S.Map t)) : a → a'» ∧
                      map (ψ (a, a') (ℰ (Cop-S.Map t))) = t
        using ide-a ide-a' Hom.ψ-mapsto by auto
    qed
  qed

  lemma is-full-functor:
  shows full-functor C Cop-S.comp map
    ..

  sublocale fully-faithful-functor C Cop-S.comp map ..

 end

end
```

Chapter 16

Adjunction

theory *Adjunction*
imports *Yoneda*
begin

This theory defines the notions of adjoint functor and adjunction in various ways and establishes their equivalence. The notions "left adjoint functor" and "right adjoint functor" are defined in terms of universal arrows. "Meta-adjunctions" are defined in terms of natural bijections between hom-sets, where the notion of naturality is axiomatized directly. "Hom-adjunctions" formalize the notion of adjunction in terms of natural isomorphisms of hom-functors. "Unit-counit adjunctions" define adjunctions in terms of functors equipped with unit and counit natural transformations that satisfy the usual "triangle identities." The *adjunction* locale is defined as the grand unification of all the definitions, and includes formulas that connect the data from each of them. It is shown that each of the definitions induces an interpretation of the *adjunction* locale, so that all the definitions are essentially equivalent. Finally, it is shown that right adjoint functors are unique up to natural isomorphism.

The reference [7] was useful in constructing this theory.

16.1 Left Adjoint Functor

"e is an arrow from $F\ x$ to y."

locale *arrow-from-functor* =
  C: *category* C +
  D: *category* D +
  F: *functor* $D\ C\ F$
  for $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ *55*)
  and $C :: 'c\ comp$ (**infixr** $\langle\cdot_C\rangle$ *55*)
  and $F :: 'd \Rightarrow 'c$
  and $x :: 'd$
  and $y :: 'c$
  and $e :: 'c$ +
  assumes *arrow*: $D.ide\ x \wedge C.in\text{-}hom\ e\ (F\ x)\ y$

begin

notation *C.in-hom* (‹«- : - →$_C$ -»›)
notation *D.in-hom* (‹«- : - →$_D$ -»›)

"g is a D-coextension of f along e."

definition *is-coext* :: $'d \Rightarrow 'c \Rightarrow 'd \Rightarrow$ *bool*
where *is-coext* $x'\ f\ g \equiv$ «$g : x' \to_D x$» $\wedge$ $f = e \cdot_C F\ g$

end

"e is a terminal arrow from $F\ x$ to y."

locale *terminal-arrow-from-functor* =
arrow-from-functor D C F x y e
for D :: $'d$ *comp* (**infixr** ‹·$_D$› *55*)
and C :: $'c$ *comp* (**infixr** ‹·$_C$› *55*)
and F :: $'d \Rightarrow 'c$
and x :: $'d$
and y :: $'c$
and e :: $'c$ +
assumes *is-terminal*: *arrow-from-functor D C F x' y f* $\Longrightarrow$ ($\exists!g$. *is-coext x' f g*)
begin

definition *the-coext* :: $'d \Rightarrow 'c \Rightarrow 'd$
where *the-coext* $x'\ f$ = (*THE g. is-coext* $x'\ f\ g$)

lemma *the-coext-prop*:
assumes *arrow-from-functor D C F x' y f*
shows «*the-coext* $x'\ f : x' \to_D x$» **and** $f = e \cdot_C F$ (*the-coext* $x'\ f$)
by (*metis assms is-coext-def is-terminal the-coext-def the-equality*)+

lemma *the-coext-unique*:
assumes *arrow-from-functor D C F x' y f* **and** *is-coext* $x'\ f\ g$
shows g = *the-coext* $x'\ f$
using *assms is-terminal the-coext-def the-equality* **by** *metis*

end

A left adjoint functor is a functor F: $D \to C$ that enjoys the following universal coextension property: for each object y of C there exists an object x of D and an arrow $e \in C.hom\ (F\ x)\ y$ such that for any arrow $f \in C.hom\ (F\ x')\ y$ there exists a unique $g \in D.hom\ x'\ x$ such that $f = C\ e\ (F\ g)$.

locale *left-adjoint-functor* =
C: *category C* +
D: *category D* +
functor D C F
for D :: $'d$ *comp* (**infixr** ‹·$_D$› *55*)
and C :: $'c$ *comp* (**infixr** ‹·$_C$› *55*)
and F :: $'d \Rightarrow 'c$ +

assumes *ex-terminal-arrow*: $C.ide\ y \Longrightarrow (\exists x\ e.\ terminal\text{-}arrow\text{-}from\text{-}functor\ D\ C\ F\ x\ y\ e)$
begin

notation $C.in\text{-}hom$ $(\langle\langle\text{- : - } \to_C \text{ -}\rangle\rangle)$
notation $D.in\text{-}hom$ $(\langle\langle\text{- : - } \to_D \text{ -}\rangle\rangle)$

end

16.2 Right Adjoint Functor

"e is an arrow from x to $G\ y$."

locale *arrow-to-functor* =
C: *category* C +
D: *category* D +
G: *functor* $C\ D\ G$
for $C :: 'c\ comp$ (**infixr** $\langle\cdot_C\rangle$ 55)
and $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ 55)
and $G :: 'c \Rightarrow 'd$
and $x :: 'd$
and $y :: 'c$
and $e :: 'd$ +
assumes *arrow*: $C.ide\ y \wedge D.in\text{-}hom\ e\ x\ (G\ y)$
begin

notation $C.in\text{-}hom$ $(\langle\langle\text{- : - } \to_C \text{ -}\rangle\rangle)$
notation $D.in\text{-}hom$ $(\langle\langle\text{- : - } \to_D \text{ -}\rangle\rangle)$

"f is a C-extension of g along e."

definition *is-ext* $:: 'c \Rightarrow 'd \Rightarrow 'c \Rightarrow bool$
where $is\text{-}ext\ y'\ g\ f \equiv \langle\langle f : y \to_C y' \rangle\rangle \wedge g = G\ f \cdot_D e$

end

"e is an initial arrow from x to $G\ y$."

locale *initial-arrow-to-functor* =
arrow-to-functor $C\ D\ G\ x\ y\ e$
for $C :: 'c\ comp$ (**infixr** $\langle\cdot_C\rangle$ 55)
and $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ 55)
and $G :: 'c \Rightarrow 'd$
and $x :: 'd$
and $y :: 'c$
and $e :: 'd$ +
assumes *is-initial*: $arrow\text{-}to\text{-}functor\ C\ D\ G\ x\ y' \Longrightarrow (\exists! f.\ is\text{-}ext\ y'\ g\ f)$
begin

definition *the-ext* $:: 'c \Rightarrow 'd \Rightarrow 'c$
where $the\text{-}ext\ y'\ g = (THE\ f.\ is\text{-}ext\ y'\ g\ f)$

lemma *the-ext-prop*:
assumes *arrow-to-functor C D G x y′ g*
shows «*the-ext y′ g* : *y* →$_C$ *y′*» **and** *g* = *G* (*the-ext y′ g*) ·$_D$ *e*
by (*metis assms is-initial is-ext-def the-equality the-ext-def*)+

lemma *the-ext-unique*:
assumes *arrow-to-functor C D G x y′ g* **and** *is-ext y′ g f*
shows *f* = *the-ext y′ g*
using *assms is-initial the-ext-def the-equality* **by** *metis*

end

A right adjoint functor is a functor G: $C \to D$ that enjoys the following universal extension property: for each object x of D there exists an object y of C and an arrow $e \in D.hom\ x\ (G\ y)$ such that for any arrow $g \in D.hom\ x\ (G\ y')$ there exists a unique $f \in C.hom\ y\ y'$ such that $h = D\ e\ (G\ f)$.

locale *right-adjoint-functor* =
C: *category C* +
D: *category D* +
functor C D G
for *C* :: *′c comp* (**infixr** ‹·$_C$› *55*)
and *D* :: *′d comp* (**infixr** ‹·$_D$› *55*)
and *G* :: *′c* ⇒ *′d* +
assumes *ex-initial-arrow*: *D.ide x* ⟹ (∃ *y e*. *initial-arrow-to-functor C D G x y e*)
begin

notation *C.in-hom* (‹«- : - →$_C$ -»›)
notation *D.in-hom* (‹«- : - →$_D$ -»›)

end

16.3 Various Definitions of Adjunction

16.3.1 Meta-Adjunction

A "meta-adjunction" consists of a functor F: $D \to C$, a functor G: $C \to D$, and for each object x of C and y of D a bijection between $C.hom\ (F\ y)\ x$ to $D.hom\ y\ (G\ x)$ which is natural in x and y. The naturality is easy to express at the meta-level without having to resort to the formal baggage of "set category," "hom-functor," and "natural isomorphism," hence the name.

locale *meta-adjunction* =
C: *category C* +
D: *category D* +
F: *functor D C F* +
G: *functor C D G*
for *C* :: *′c comp* (**infixr** ‹·$_C$› *55*)
and *D* :: *′d comp* (**infixr** ‹·$_D$› *55*)
and *F* :: *′d* ⇒ *′c*

and $G :: 'c \Rightarrow 'd$
and $\varphi :: 'd \Rightarrow 'c \Rightarrow 'd$
and $\psi :: 'c \Rightarrow 'd \Rightarrow 'c +$
assumes *φ-in-hom*: $[\![D.ide\ y;\ C.in\text{-}hom\ f\ (F\ y)\ x]\!] \Longrightarrow D.in\text{-}hom\ (\varphi\ y\ f)\ y\ (G\ x)$
and *ψ-in-hom*: $[\![C.ide\ x;\ D.in\text{-}hom\ g\ y\ (G\ x)]\!] \Longrightarrow C.in\text{-}hom\ (\psi\ x\ g)\ (F\ y)\ x$
and *ψ-φ*: $[\![D.ide\ y;\ C.in\text{-}hom\ f\ (F\ y)\ x]\!] \Longrightarrow \psi\ x\ (\varphi\ y\ f) = f$
and *φ-ψ*: $[\![C.ide\ x;\ D.in\text{-}hom\ g\ y\ (G\ x)]\!] \Longrightarrow \varphi\ y\ (\psi\ x\ g) = g$
and *φ-naturality*: $[\![C.in\text{-}hom\ f\ x\ x';\ D.in\text{-}hom\ g\ y'\ y;\ C.in\text{-}hom\ h\ (F\ y)\ x]\!] \Longrightarrow$
$\varphi\ y'\ (f\ \cdot_C\ h\ \cdot_C\ F\ g) = G\ f\ \cdot_D\ \varphi\ y\ h\ \cdot_D\ g$
begin

notation $C.in\text{-}hom$ ($\langle\langle$- : - $\rightarrow_C$ -$\rangle\rangle$)
notation $D.in\text{-}hom$ ($\langle\langle$- : - $\rightarrow_D$ -$\rangle\rangle$)

The naturality of ψ is a consequence of the naturality of φ and the other assumptions.

lemma *ψ-naturality*:
assumes *f*: «$f : x \rightarrow_C x'$» **and** *g*: «$g : y' \rightarrow_D y$» **and** *h*: «$h : y \rightarrow_D G\ x$»
shows $f\ \cdot_C\ \psi\ x\ h\ \cdot_C\ F\ g = \psi\ x'\ (G\ f\ \cdot_D\ h\ \cdot_D\ g)$
using *f g h φ-naturality ψ-in-hom C.ide-dom D.ide-dom D.in-homE φ-ψ ψ-φ*
by (*metis C.comp-in-homI' F.preserves-hom C.in-homE D.in-homE*)

lemma *respects-natural-isomorphism*:
assumes *natural-isomorphism D C F' F τ* **and** *natural-isomorphism C D G G' μ*
shows *meta-adjunction C D F' G'*
($\lambda y\ f.\ \mu\ (C.cod\ f)\ \cdot_D\ \varphi\ y\ (f\ \cdot_C$ *inverse-transformation.map D C F τ y*))
($\lambda x\ g.\ \psi\ x\ ((inverse\text{-}transformation.map\ C\ D\ G'\ \mu\ x)\ \cdot_D\ g)\ \cdot_C\ \tau\ (D.dom\ g)$)
proof −
interpret *τ*: *natural-isomorphism D C F' F τ*
using *assms*(*1*) **by** *simp*
interpret *τ'*: *inverse-transformation D C F' F τ*
..
interpret *μ*: *natural-isomorphism C D G G' μ*
using *assms*(*2*) **by** *simp*
interpret *μ'*: *inverse-transformation C D G G' μ*
..
let *?φ'* = $\lambda y\ f.\ \mu\ (C.cod\ f)\ \cdot_D\ \varphi\ y\ (f\ \cdot_C\ \tau'.map\ y)$
let *?ψ'* = $\lambda x\ g.\ \psi\ x\ (\mu'.map\ x\ \cdot_D\ g)\ \cdot_C\ \tau\ (D.dom\ g)$
show *meta-adjunction C D F' G' ?φ' ?ψ'*
proof
show $\bigwedge y\ f\ x.\ [\![D.ide\ y;$ «$f : F'\ y \rightarrow_C x$»$]\!]$
$\Longrightarrow$ «$\mu\ (C.cod\ f)\ \cdot_D\ \varphi\ y\ (f\ \cdot_C\ \tau'.map\ y) : y \rightarrow_D G'\ x$»
proof −
fix *x y f*
assume *y*: *D.ide y* **and** *f*: «$f : F'\ y \rightarrow_C x$»
show «$\mu\ (C.cod\ f)\ \cdot_D\ \varphi\ y\ (f\ \cdot_C\ \tau'.map\ y) : y \rightarrow_D G'\ x$»
proof (*intro D.comp-in-homI*)
show «$\mu\ (C.cod\ f) : G\ x \rightarrow_D G'\ x$»
using *f* **by** *fastforce*
show «$\varphi\ y\ (f\ \cdot_C\ \tau'.map\ y) : y \rightarrow_D G\ x$»

```
    using f y φ-in-hom by auto
  qed
qed
show ⋀x g y. ⟦C.ide x; «g : y →_D G' x»⟧
               ⟹ «ψ x (μ'.map x ·_D g) ·_C τ (D.dom g) : F' y →_C x»
proof −
  fix x y g
  assume x: C.ide x and g: «g : y →_D G' x»
  show «ψ x (μ'.map x ·_D g) ·_C τ (D.dom g) : F' y →_C x»
  proof (intro C.comp-in-homI)
    show «τ (D.dom g) : F' y →_C F y»
      using g by fastforce
    show «ψ x (μ'.map x ·_D g) : F y →_C x»
      using x g ψ-in-hom by auto
  qed
qed
show ⋀y f x. ⟦D.ide y; «f : F' y →_C x»⟧
                ⟹ ψ x (μ'.map x ·_D μ (C.cod f) ·_D φ y (f ·_C τ'.map y)) ·_C
                     τ (D.dom (μ (C.cod f) ·_D φ y (f ·_C τ'.map y))) =
                   f
proof −
  fix x y f
  assume y: D.ide y and f: «f : F' y →_C x»
  have 1: «φ y (f ·_C τ'.map y) : y →_D G x»
    using f y φ-in-hom by auto
  show ψ x (μ'.map x ·_D μ (C.cod f) ·_D φ y (f ·_C τ'.map y)) ·_C
         τ (D.dom (μ (C.cod f) ·_D φ y (f ·_C τ'.map y))) =
       f
  proof −
    have ψ x (μ'.map x ·_D μ (C.cod f) ·_D φ y (f ·_C τ'.map y)) ·_C
           τ (D.dom (μ (C.cod f) ·_D φ y (f ·_C τ'.map y))) =
         ψ x ((μ'.map x ·_D μ (C.cod f)) ·_D φ y (f ·_C τ'.map y)) ·_C
           τ (D.dom (μ (C.cod f) ·_D φ y (f ·_C τ'.map y)))
      using D.comp-assoc by simp
    also have ... = ψ x (φ y (f ·_C τ'.map y)) ·_C τ y
     by (metis 1 C.arr-cod C.dom-cod C.ide-cod C.in-homE D.comp-ide-arr D.dom-comp
       D.ide-compE D.in-homE D.inverse-arrowsE μ'.inverts-components μ.preserves-dom
         μ.preserves-reflects-arr category.seqI f meta-adjunction-axioms
         meta-adjunction-def)
    also have ... = f
      using f y ψ-φ C.comp-assoc τ'.inverts-components [of y] C.comp-arr-dom
      by fastforce
    finally show ?thesis by blast
  qed
qed
show ⋀x g y. ⟦C.ide x; «g : y →_D G' x»⟧
                ⟹ μ (C.cod (ψ x (μ'.map x ·_D g) ·_C τ (D.dom g))) ·_D
                     φ y ((ψ x (μ'.map x ·_D g) ·_C τ (D.dom g)) ·_C τ'.map y) =
                   g
```

```
    proof −
      fix x y g
      assume x: C.ide x and g: «g : y →_D G′ x»
      have 1: «ψ x (μ′.map x ·_D g) : F y →_C x»
        using x g ψ-in-hom by auto
      show μ (C.cod (ψ x (μ′.map x ·_D g) ·_C τ (D.dom g))) ·_D
             φ y ((ψ x (μ′.map x ·_D g) ·_C τ (D.dom g)) ·_C τ′.map y) =
           g
      proof −
        have μ (C.cod (ψ x (μ′.map x ·_D g) ·_C τ (D.dom g))) ·_D
               φ y ((ψ x (μ′.map x ·_D g) ·_C τ (D.dom g)) ·_C τ′.map y) =
             μ (C.cod (ψ x (μ′.map x ·_D g) ·_C τ (D.dom g))) ·_D
               φ y (ψ x (μ′.map x ·_D g) ·_C τ (D.dom g) ·_C τ′.map y)
          using C.comp-assoc by simp
        also have ... = μ x ·_D φ y (ψ x (μ′.map x ·_D g))
          using 1 C.comp-arr-dom C.comp-arr-inv′ g by fastforce
        also have ... = (μ x ·_D μ′.map x) ·_D g
          using x g φ-ψ D.comp-assoc by auto
        also have ... = g
          using x g μ′.inverts-components [of x] D.comp-cod-arr by fastforce
        finally show ?thesis by blast
      qed
    qed
    show ⋀f x x′ g y′ y h. ⟦«f : x →_C x′»; «g : y′ →_D y»; «h : F′ y →_C x»⟧
           ⟹ μ (C.cod (f ·_C h ·_C F′ g)) ·_D φ y′ ((f ·_C h ·_C F′ g) ·_C τ′.map y′) =
              G′ f ·_D (μ (C.cod h) ·_D φ y (h ·_C τ′.map y)) ·_D g
    proof −
      fix x y x′ y′ f g h
      assume f: «f : x →_C x′» and g: «g : y′ →_D y» and h: «h : F′ y →_C x»
      show μ (C.cod (f ·_C h ·_C F′ g)) ·_D φ y′ ((f ·_C h ·_C F′ g) ·_C τ′.map y′) =
           G′ f ·_D (μ (C.cod h) ·_D φ y (h ·_C τ′.map y)) ·_D g
      proof −
        have μ (C.cod (f ·_C h ·_C F′ g)) ·_D φ y′ ((f ·_C h ·_C F′ g) ·_C τ′.map y′) =
             μ x′ ·_D φ y′ ((f ·_C h ·_C F′ g) ·_C τ′.map y′)
          using f g h by fastforce
        also have ... = μ x′ ·_D φ y′ (f ·_C (h ·_C τ′.map y) ·_C F g)
          using g τ′.naturality C.comp-assoc by auto
        also have ... = (μ x′ ·_D G f) ·_D φ y (h ·_C τ′.map y) ·_D g
          using f g h φ-naturality [of f x x′ g y′ y h ·_C τ′.map y] D.comp-assoc
          by fastforce
        also have ... = (G′ f ·_D μ x) ·_D φ y (h ·_C τ′.map y) ·_D g
          using f μ.naturality by auto
        also have ... = G′ f ·_D (μ (C.cod h) ·_D φ y (h ·_C τ′.map y)) ·_D g
          using h D.comp-assoc by auto
        finally show ?thesis by blast
      qed
    qed
  qed
qed
```

end

16.3.2 Hom-Adjunction

The bijection between hom-sets that defines an adjunction can be represented formally as a natural isomorphism of hom-functors. However, stating the definition this way is more complex than was the case for *meta-adjunction*. One reason is that we need to have a "set category" that is suitable as a target category for the hom-functors, and since the arrows of the categories C and D will in general have distinct types, we need a set category that simultaneously embeds both. Another reason is that we simply have to formally construct the various categories and functors required to express the definition.

This is a good place to point out that I have often included more sublocales in a locale than are strictly required. The main reason for this is the fact that the locale system in Isabelle only gives one name to each entity introduced by a locale: the name that it has in the first locale in which it occurs. This means that entities that make their first appearance deeply nested in sublocales will have to be referred to by long qualified names that can be difficult to understand, or even to discover. To counteract this, I have typically introduced sublocales before the superlocales that contain them to ensure that the entities in the sublocales can be referred to by short meaningful (and predictable) names. In my opinion, though, it would be better if the locale system would make entities that occur in multiple locales accessible by *all* possible qualified names, so that the most perspicuous name could be used in any particular context.

locale *hom-adjunction* =
  C: *category C* +
  D: *category D* +
  S: *set-category S setp* +
  Cop: *dual-category C* +
  Dop: *dual-category D* +
  CopxC: *product-category Cop.comp C* +
  DopxD: *product-category Dop.comp D* +
  DopxC: *product-category Dop.comp C* +
  F: *functor D C F* +
  G: *functor C D G* +
  HomC: *hom-functor C S setp* φC +
  HomD: *hom-functor D S setp* φD +
  Fop: *dual-functor Dop.comp Cop.comp F* +
  FopxC: *product-functor Dop.comp C Cop.comp C Fop.map C.map* +
  DopxG: *product-functor Dop.comp C Dop.comp D Dop.map G* +
  Hom-FopxC: *composite-functor DopxC.comp CopxC.comp S FopxC.map HomC.map* +
  Hom-DopxG: *composite-functor DopxC.comp DopxD.comp S DopxG.map HomD.map* +
  Hom-FopxC: *set-valued-functor DopxC.comp S setp Hom-FopxC.map* +
  Hom-DopxG: *set-valued-functor DopxC.comp S setp Hom-DopxG.map* +
  Φ: *set-valued-transformation DopxC.comp S setp Hom-FopxC.map Hom-DopxG.map* Φ +
  Ψ: *set-valued-transformation DopxC.comp S setp Hom-DopxG.map Hom-FopxC.map* Ψ +
  $\Phi\Psi$: *inverse-transformations DopxC.comp S Hom-FopxC.map Hom-DopxG.map* Φ Ψ
  for *C* :: *'c comp* (**infixr** $\langle\cdot_C\rangle$ *55*)

and $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ *55*)
and $S :: 's\ comp$ (**infixr** $\langle\cdot_S\rangle$ *55*)
and $setp :: 's\ set \Rightarrow bool$
and $\varphi C :: 'c * 'c \Rightarrow 'c \Rightarrow 's$
and $\varphi D :: 'd * 'd \Rightarrow 'd \Rightarrow 's$
and $F :: 'd \Rightarrow 'c$
and $G :: 'c \Rightarrow 'd$
and $\Phi :: 'd * 'c \Rightarrow 's$
and $\Psi :: 'd * 'c \Rightarrow 's$
begin

notation *C.in-hom* ($\langle\langle$- : - $\rightarrow_C$ -$\rangle\rangle$)
notation *D.in-hom* ($\langle\langle$- : - $\rightarrow_D$ -$\rangle\rangle$)

abbreviation $\psi C :: 'c * 'c \Rightarrow 's \Rightarrow 'c$
where $\psi C \equiv HomC.\psi$

abbreviation $\psi D :: 'd * 'd \Rightarrow 's \Rightarrow 'd$
where $\psi D \equiv HomD.\psi$

end

16.3.3 Unit/Counit Adjunction

Expressed in unit/counit terms, an adjunction consists of functors F: $D \to C$ and G: $C \to D$, equipped with natural transformations η: $1 \to GF$ and ε: $FG \to 1$ satisfying certain "triangle identities".

locale *unit-counit-adjunction* =
C: *category C* +
D: *category D* +
F: *functor D C F* +
G: *functor C D G* +
GF: *composite-functor D C D F G* +
FG: *composite-functor C D C G F* +
FGF: *composite-functor D C C F* ‹*F o G*› +
GFG: *composite-functor C D D G* ‹*G o F*› +
η: *natural-transformation D D D.map* ‹*G o F*› η +
ε: *natural-transformation C C* ‹*F o G*› *C.map* ε +
$F\eta$: *natural-transformation D C F* ‹*F o G o F*› ‹*F o* η› +
ηG: *natural-transformation C D G* ‹*G o F o G*› ‹η *o G*› +
εF: *natural-transformation D C* ‹*F o G o F*› *F* ‹ε *o F*› +
$G\varepsilon$: *natural-transformation C D* ‹*G o F o G*› *G* ‹*G o* ε› +
$\varepsilon FoF\eta$: *vertical-composite D C F* ‹*F o G o F*› *F* ‹*F o* η› ‹ε *o F*› +
$G\varepsilon o\eta G$: *vertical-composite C D G* ‹*G o F o G*› *G* ‹η *o G*› ‹*G o* ε›
for $C :: 'c\ comp$ (**infixr** $\langle\cdot_C\rangle$ *55*)
and $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ *55*)
and $F :: 'd \Rightarrow 'c$
and $G :: 'c \Rightarrow 'd$
and $\eta :: 'd \Rightarrow 'd$

```
  and ε :: 'c ⇒ 'c +
  assumes triangle-F: εFoFη.map = F
  and triangle-G: GεoηG.map = G
begin

  notation C.in-hom      (‹«- : - →C -»›)
  notation D.in-hom      (‹«- : - →D -»›)

end

lemma unit-determines-counit:
assumes unit-counit-adjunction C D F G η ε
and unit-counit-adjunction C D F G η ε'
shows ε = ε'
proof −

  interpret Adj: unit-counit-adjunction C D F G η ε using assms(1) by auto
  interpret Adj': unit-counit-adjunction C D F G η ε' using assms(2) by auto
  interpret FGFG: composite-functor C D C G ‹F o G o F› ..
  interpret FGε: natural-transformation C C ‹(F o G) o (F o G)› ‹F o G› ‹(F o G) o ε›
  using Adj.ε.natural-transformation-axioms Adj.FG.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by fastforce
  interpret FηG: natural-transformation C C ‹F o G› ‹F o G o F o G› ‹F o η o G›
    using Adj.η.natural-transformation-axioms Adj.Fη.natural-transformation-axioms
        Adj.G.as-nat-trans.natural-transformation-axioms horizontal-composite
    by blast
  interpret ε'ε: natural-transformation C C ‹F o G o F o G› Adj.C.map ‹ε' o ε›
  proof −
    have natural-transformation C C ((F o G) o (F o G)) Adj.C.map (ε' o ε)
      using Adj.ε.natural-transformation-axioms Adj'.ε.natural-transformation-axioms
          horizontal-composite Adj.C.is-functor comp-functor-identity
      by (metis (no-types, lifting))
    thus natural-transformation C C (F o G o F o G) Adj.C.map (ε' o ε)
      using o-assoc by metis
  qed
  interpret ε'εoFηG: vertical-composite
                  C C ‹F o G› ‹F o G o F o G› Adj.C.map ‹F o η o G› ‹ε' o ε› ..
  have ε' = vertical-composite.map C C (F o Adj.GεoηG.map) ε'
    using vcomp-ide-dom [of C C F o G Adj.C.map ε'] Adj.triangle-G
    by (simp add: Adj'.ε.natural-transformation-axioms)
  also have ... = vertical-composite.map C C
                  (vertical-composite.map C C (F o η o G) (F o G o ε)) ε'
    using whisker-left Adj.F.functor-axioms Adj.Gε.natural-transformation-axioms
        Adj.ηG.natural-transformation-axioms o-assoc
    by (metis (no-types, lifting))
  also have ... = vertical-composite.map C C
                  (vertical-composite.map C C (F o η o G) (ε' o F o G)) ε
  proof −
```

```
    have vertical-composite.map C C
          (vertical-composite.map C C (F o η o G) (F o G o ε)) ε'
          = vertical-composite.map C C (F o η o G)
              (vertical-composite.map C C (F o G o ε) ε')
      using vcomp-assoc
      by (metis (no-types, lifting) Adj'.ε.natural-transformation-axioms
          FGε.natural-transformation-axioms FηG.natural-transformation-axioms o-assoc)
    also have ... = vertical-composite.map C C (F o η o G)
                      (vertical-composite.map C C (ε' o F o G) ε)
      using Adj'.ε.natural-transformation-axioms Adj.ε.natural-transformation-axioms
          interchange-spc [of C C F o G Adj.C.map ε C F o G Adj.C.map ε']
      by (metis hcomp-ide-cod hcomp-ide-dom o-assoc)
    also have ... = vertical-composite.map C C
                      (vertical-composite.map C C (F o η o G) (ε' o F o G)) ε
      using vcomp-assoc
      by (metis Adj'.εF.natural-transformation-axioms
          Adj.G.as-nat-trans.natural-transformation-axioms
          Adj.ε.natural-transformation-axioms FηG.natural-transformation-axioms
          horizontal-composite)
    finally show ?thesis by simp
  qed
  also have ... = vertical-composite.map C C
                    (vertical-composite.map D C (F o η) (ε' o F) o G) ε
    using whisker-right Adj'.εF.natural-transformation-axioms
        Adj.Fη.natural-transformation-axioms Adj.G.functor-axioms
    by metis
  also have ... = ε
    using Adj'.triangle-F vcomp-ide-cod Adj.ε.natural-transformation-axioms by simp
  finally show ?thesis by simp
qed

lemma counit-determines-unit:
assumes unit-counit-adjunction C D F G η ε
and unit-counit-adjunction C D F G η' ε
shows η = η'
proof −
  interpret Adj: unit-counit-adjunction C D F G η ε using assms(1) by auto
  interpret Adj': unit-counit-adjunction C D F G η' ε using assms(2) by auto
  interpret GFGF: composite-functor D C D F ‹G o F o G› ..
  interpret GFη: natural-transformation D D ‹G o F› ‹(G o F) o (G o F)› ‹(G o F) o η›
    using Adj.η.natural-transformation-axioms Adj.GF.functor-axioms
        Adj.GF.as-nat-trans.natural-transformation-axioms comp-functor-identity
        horizontal-composite
    by (metis (no-types, lifting))
  interpret η'GF: natural-transformation D D ‹G o F› ‹(G o F) o (G o F)› ‹η' o (G o F)›
    using Adj'.η.natural-transformation-axioms Adj.GF.functor-axioms
        Adj.GF.as-nat-trans.natural-transformation-axioms comp-identity-functor
        horizontal-composite
    by (metis (no-types, lifting))
```

interpret $G\varepsilon F$: *natural-transformation D D* ‹*G o F o G o F*› ‹*G o F*› ‹*G o ε o F*›
using *Adj.ε.natural-transformation-axioms Adj.F.as-nat-trans.natural-transformation-axioms*
Adj.Gε.natural-transformation-axioms horizontal-composite
by *blast*
interpret $\eta'\eta$: *natural-transformation D D Adj.D.map* ‹*G o F o G o F*› ‹η' *o* η›
proof −
have *natural-transformation D D Adj.D.map* ((*G o F*) *o* (*G o F*)) (η' *o* η)
using *Adj′.η.natural-transformation-axioms Adj.D.identity-functor-axioms*
Adj.η.natural-transformation-axioms horizontal-composite identity-functor.is-functor
by *fastforce*
thus *natural-transformation D D Adj.D.map* (*G o F o G o F*) (η' *o* η)
using *o-assoc* **by** *metis*
qed
interpret $G\varepsilon Fo\eta'\eta$: *vertical-composite*
D D Adj.D.map ‹*G o F o G o F*› ‹*G o F*› ‹η' *o* η› ‹*G o ε o F*› **..**
have η' = *vertical-composite.map D D* η' (*G o Adj.εFoFη.map*)
using *vcomp-ide-cod* [*of D D Adj.D.map G o F* η'] *Adj.triangle-F*
by (*simp add*: *Adj′.η.natural-transformation-axioms*)
also have ... = *vertical-composite.map D D* η'
(*vertical-composite.map D D* (*G o* (*F o* η)) (*G o* (ε *o F*)))
using *whisker-left Adj.Fη.natural-transformation-axioms Adj.G.functor-axioms*
Adj.εF.natural-transformation-axioms
by *fastforce*
also have ... = *vertical-composite.map D D*
(*vertical-composite.map D D* η' (*G o* (*F o* η))) (*G o ε o F*)
using *vcomp-assoc Adj′.η.natural-transformation-axioms*
GFη.natural-transformation-axioms GεF.natural-transformation-axioms o-assoc
by (*metis* (*no-types*, *lifting*))
also have ... = *vertical-composite.map D D*
(*vertical-composite.map D D* η (η' *o G o F*)) (*G o ε o F*)
using *interchange-spc* [*of D D Adj.D.map G o F* η *D Adj.D.map G o F* η']
Adj.η.natural-transformation-axioms Adj′.η.natural-transformation-axioms
by (*metis hcomp-ide-cod hcomp-ide-dom o-assoc*)
also have ... = *vertical-composite.map D D* η
(*vertical-composite.map D D* (η' *o G o F*) (*G o ε o F*))
using *vcomp-assoc*
by (*metis* (*no-types*, *lifting*) *Adj.η.natural-transformation-axioms*
GεF.natural-transformation-axioms η′GF.natural-transformation-axioms o-assoc)
also have ... = *vertical-composite.map D D* η
(*vertical-composite.map C D* (η' *o G*) (*G o* ε) *o F*)
using *whisker-right Adj′.ηG.natural-transformation-axioms Adj.F.functor-axioms*
Adj.Gε.natural-transformation-axioms
by *fastforce*
also have ... = η
using *Adj′.triangle-G vcomp-ide-dom Adj.GF.functor-axioms*
Adj.η.natural-transformation-axioms
by *simp*
finally show *?thesis* **by** *simp*
qed

16.3.4 Adjunction

The grand unification of everything to do with an adjunction.

locale *adjunction* =
C: *category C* +
D: *category D* +
S: *set-category S setp* +
Cop: *dual-category C* +
Dop: *dual-category D* +
CopxC: *product-category Cop.comp C* +
DopxD: *product-category Dop.comp D* +
DopxC: *product-category Dop.comp C* +
idDop: *identity-functor Dop.comp* +
HomC: *hom-functor C S setp φC* +
HomD: *hom-functor D S setp φD* +
F: *left-adjoint-functor D C F* +
G: *right-adjoint-functor C D G* +
GF: *composite-functor D C D F G* +
FG: *composite-functor C D C G F* +
FGF: *composite-functor D C C F FG.map* +
GFG: *composite-functor C D D G GF.map* +
Fop: *dual-functor Dop.comp Cop.comp F* +
FopxC: *product-functor Dop.comp C Cop.comp C Fop.map C.map* +
DopxG: *product-functor Dop.comp C Dop.comp D Dop.map G* +
Hom-FopxC: *composite-functor DopxC.comp CopxC.comp S FopxC.map HomC.map* +
Hom-DopxG: *composite-functor DopxC.comp DopxD.comp S DopxG.map HomD.map* +
Hom-FopxC: *set-valued-functor DopxC.comp S setp Hom-FopxC.map* +
Hom-DopxG: *set-valued-functor DopxC.comp S setp Hom-DopxG.map* +
η: *natural-transformation D D D.map GF.map η* +
ε: *natural-transformation C C FG.map C.map ε* +
Fη: *natural-transformation D C F ‹F o G o F› ‹F o η›* +
ηG: *natural-transformation C D G ‹G o F o G› ‹η o G›* +
εF: *natural-transformation D C ‹F o G o F› F ‹ε o F›* +
Gε: *natural-transformation C D ‹G o F o G› G ‹G o ε›* +
εFoFη: *vertical-composite D C F FGF.map F ‹F o η› ‹ε o F›* +
Gεoηg: *vertical-composite C D G GFG.map G ‹η o G› ‹G o ε›* +
φψ: *meta-adjunction C D F G φ ψ* +
ηε: *unit-counit-adjunction C D F G η ε* +
ΦΨ: *hom-adjunction C D S setp φC φD F G Φ Ψ*
for *C* :: *'c comp* (**infixr** ‹·$_C$› *55*)
and *D* :: *'d comp* (**infixr** ‹·$_D$› *55*)
and *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *setp* :: *'s set* ⇒ *bool*
and *φC* :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
and *φD* :: *'d* ∗ *'d* ⇒ *'d* ⇒ *'s*
and *F* :: *'d* ⇒ *'c*
and *G* :: *'c* ⇒ *'d*
and *φ* :: *'d* ⇒ *'c* ⇒ *'d*
and *ψ* :: *'c* ⇒ *'d* ⇒ *'c*

and $\eta :: 'd \Rightarrow 'd$
and $\varepsilon :: 'c \Rightarrow 'c$
and $\Phi :: 'd * 'c \Rightarrow 's$
and $\Psi :: 'd * 'c \Rightarrow 's +$
assumes φ*-in-terms-of-*η: $[\![D.ide\ y;\ «f : F\ y \rightarrow_C x»]\!] \Longrightarrow \varphi\ y\ f = G\ f \cdot_D \eta\ y$
and ψ*-in-terms-of-*ε: $[\![C.ide\ x;\ «g : y \rightarrow_D G\ x»]\!] \Longrightarrow \psi\ x\ g = \varepsilon\ x \cdot_C F\ g$
and η*-in-terms-of-*φ: $D.ide\ y \Longrightarrow \eta\ y = \varphi\ y\ (F\ y)$
and ε*-in-terms-of-*ψ: $C.ide\ x \Longrightarrow \varepsilon\ x = \psi\ x\ (G\ x)$
and φ*-in-terms-of-*Φ: $[\![D.ide\ y;\ «f : F\ y \rightarrow_C x»]\!] \Longrightarrow$
$\varphi\ y\ f = (\Phi\Psi.\psi D\ (y,\ G\ x)\ o\ S.Fun\ (\Phi\ (y,\ x))\ o\ \varphi C\ (F\ y,\ x))\ f$
and ψ*-in-terms-of-*Ψ: $[\![C.ide\ x;\ «g : y \rightarrow_D G\ x»]\!] \Longrightarrow$
$\psi\ x\ g = (\Phi\Psi.\psi C\ (F\ y,\ x)\ o\ S.Fun\ (\Psi\ (y,\ x))\ o\ \varphi D\ (y,\ G\ x))\ g$
and Φ*-in-terms-of-*φ:
$[\![C.ide\ x;\ D.ide\ y]\!] \Longrightarrow$
$\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \Phi\Psi.\psi C\ (F\ y,\ x))$
and Ψ*-in-terms-of-*ψ:
$[\![C.ide\ x;\ D.ide\ y]\!] \Longrightarrow$
$\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \Phi\Psi.\psi D\ (y,\ G\ x))$

16.4 Meta-Adjunctions Induce Unit/Counit Adjunctions

context *meta-adjunction*
begin

interpretation *GF*: *composite-functor D C D F G* **..**
interpretation *FG*: *composite-functor C D C G F* **..**
interpretation *FGF*: *composite-functor D C C F FG.map* **..**
interpretation *GFG*: *composite-functor C D D G GF.map* **..**

definition $\eta o :: 'd \Rightarrow 'd$
where $\eta o\ y = \varphi\ y\ (F\ y)$

lemma ηo*-in-hom*:
assumes $D.ide\ y$
shows $«\eta o\ y : y \rightarrow_D G\ (F\ y)»$
using *assms D.ide-in-hom* ηo*-def* φ*-in-hom* **by** *force*

lemma φ*-in-terms-of-*ηo:
assumes $D.ide\ y$ **and** $«f : F\ y \rightarrow_C x»$
shows $\varphi\ y\ f = G\ f \cdot_D \eta o\ y$
proof (*unfold* ηo*-def*)
have *1*: $«F\ y : F\ y \rightarrow_C F\ y»$
using *assms*(*1*) *D.ide-in-hom* **by** *blast*
hence $\varphi\ y\ (F\ y) = \varphi\ y\ (F\ y) \cdot_D y$
by (*metis assms*(*1*) *D.in-homE* φ*-in-hom D.comp-arr-dom*)
thus $\varphi\ y\ f = G\ f \cdot_D \varphi\ y\ (F\ y)$
using *assms 1 D.ide-in-hom* **by** (*metis C.comp-arr-dom C.in-homE* φ*-naturality*)

qed

lemma *φ-F-char*:
assumes «$g : y' \to_D y$»
shows $\varphi\ y'\ (F\ g) = \eta o\ y \cdot_D g$
  using *assms η o-def φ-in-hom* [*of y F y F y*]
    D.comp-cod-arr [*of D* (*φ y* (*F y*)) *g G* (*F y*)]
    φ-naturality [*of F y F y F y g y' y F y*]
  by (*metis C.ide-in-hom D.arr-cod-iff-arr D.arr-dom D.cod-cod D.cod-dom D.comp-ide-arr*
    D.comp-ide-self D.ide-cod D.in-homE F.as-nat-trans.naturality2 F.functor-axioms
    F.preserves-section-retraction φ-in-hom functor.preserves-hom)

interpretation *η*: *transformation-by-components D D D.map GF.map η o*
proof
  show $\bigwedge a.\ D.ide\ a \Longrightarrow$ «$\eta o\ a : D.map\ a \to_D GF.map\ a$»
    using *η o-def φ-in-hom D.ide-in-hom* **by** *force*
  fix *f*
  assume *f*: *D.arr f*
  show $\eta o\ (D.cod\ f) \cdot_D D.map\ f = GF.map\ f \cdot_D \eta o\ (D.dom\ f)$
    using *f φ-F-char* [*of D.map f D.dom f D.cod f*]
      φ-in-terms-of-η o [*of D.dom f F f F* (*D.cod f*)]
    by *force*
qed

lemma *η-map-simp*:
assumes *D.ide y*
shows $\eta.map\ y = \varphi\ y\ (F\ y)$
  using *assms η.map-simp-ide η o-def* **by** *simp*

definition $\varepsilon o :: 'c \Rightarrow 'c$
where $\varepsilon o\ x = \psi\ x\ (G\ x)$

lemma *ε o-in-hom*:
assumes *C.ide x*
shows «$\varepsilon o\ x : F\ (G\ x) \to_C x$»
  using *assms C.ide-in-hom ε o-def ψ-in-hom* **by** *force*

lemma *ψ-in-terms-of-ε o*:
assumes *C.ide x* **and** «$g : y \to_D G\ x$»
shows $\psi\ x\ g = \varepsilon o\ x \cdot_C F\ g$
proof −
  have $\varepsilon o\ x \cdot_C F\ g = x \cdot_C \psi\ x\ (G\ x) \cdot_C F\ g$
    using *assms ε o-def ψ-in-hom* [*of x G x G x*]
      C.comp-cod-arr [*of ψ x* (*G x*) $\cdot_C$ *F g x*]
    by *fastforce*
  also have ... $= \psi\ x\ (G\ x \cdot_D G\ x \cdot_D g)$
    using *assms ψ-naturality* [*of x x x g y G x G x*] **by** *force*
  also have ... $= \psi\ x\ g$
    using *assms D.comp-cod-arr* **by** *fastforce*

finally show *?thesis* **by** *simp*
qed

lemma *ψ-G-char*:
assumes «*f*: *x* →$_C$ *x′*»
shows *ψ x′* (*G f*) = *f* ·$_C$ *εo x*
proof (*unfold εo-def*)
have *0*: *C.ide x* ∧ *C.ide x′* **using** *assms* **by** *auto*
thus *ψ x′* (*G f*) = *f* ·$_C$ *ψ x* (*G x*)
using *0 assms ψ-naturality ψ-in-hom* [*of x G x G x*] *G.preserves-hom εo-def*
ψ-in-terms-of-εo G.as-nat-trans.naturality1 C.ide-in-hom
by (*metis C.arrI C.in-homE*)
qed

interpretation *ε*: *transformation-by-components C C FG.map C.map εo*
apply *unfold-locales*
using *εo-in-hom*
apply *simp*
using *ψ-G-char ψ-in-terms-of-εo*
by (*metis C.arr-iff-in-hom C.ide-cod C.map-simp G.preserves-hom comp-apply*)

lemma *ε-map-simp*:
assumes *C.ide x*
shows *ε.map x* = *ψ x* (*G x*)
using *assms εo-def* **by** *simp*

interpretation *FD*: *composite-functor D D C D.map F* ..
interpretation *CF*: *composite-functor D C C F C.map* ..
interpretation *GC*: *composite-functor C C D C.map G* ..
interpretation *DG*: *composite-functor C D D G D.map* ..

interpretation *Fη*: *natural-transformation D C F* ‹*F o G o F*› ‹*F o η.map*›
by (*metis* (*no-types*, *lifting*) *F.as-nat-trans.natural-transformation-axioms*
F.functor-axioms η.natural-transformation-axioms comp-functor-identity
horizontal-composite o-assoc)

interpretation *εF*: *natural-transformation D C* ‹*F o G o F*› *F* ‹*ε.map o F*›
using *ε.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms*
horizontal-composite
by *fastforce*

interpretation *ηG*: *natural-transformation C D G* ‹*G o F o G*› ‹*η.map o G*›
using *η.natural-transformation-axioms G.as-nat-trans.natural-transformation-axioms*
horizontal-composite
by *fastforce*

interpretation *Gε*: *natural-transformation C D* ‹*G o F o G*› *G* ‹*G o ε.map*›
by (*metis* (*no-types*, *lifting*) *G.as-nat-trans.natural-transformation-axioms*
G.functor-axioms ε.natural-transformation-axioms comp-functor-identity

horizontal-composite o-assoc)

interpretation *εFoFη*: *vertical-composite D C F ‹F o G o F› F ‹F o η.map› ‹ε.map o F›*
..
interpretation *GεoηG*: *vertical-composite C D G ‹G o F o G› G ‹η.map o G› ‹G o ε.map›*
..

lemma *unit-counit-F*:
assumes *D.ide y*
shows *F y = εo (F y) ·C F (ηo y)*
using *assms ψ-in-terms-of-εo ηo-def ψ-φ ηo-in-hom F.preserves-ide C.ide-in-hom* **by** *metis*

lemma *unit-counit-G*:
assumes *C.ide x*
shows *G x = G (εo x) ·D ηo (G x)*
using *assms φ-in-terms-of-ηo εo-def φ-ψ εo-in-hom G.preserves-ide D.ide-in-hom* **by** *metis*

lemma *induces-unit-counit-adjunction′*:
shows *unit-counit-adjunction C D F G η.map ε.map*
proof
show *εFoFη.map = F*
using *εFoFη.is-natural-transformation εFoFη.map-simp-ide unit-counit-F*
F.as-nat-trans.natural-transformation-axioms
by (*intro natural-transformation-eqI*) *auto*
show *GεoηG.map = G*
using *GεoηG.is-natural-transformation GεoηG.map-simp-ide unit-counit-G*
G.as-nat-trans.natural-transformation-axioms
by (*intro natural-transformation-eqI*) *auto*
qed

definition *η* :: *′d* ⇒ *′d* **where** *η* ≡ *η.map*
definition *ε* :: *′c* ⇒ *′c* **where** *ε* ≡ *ε.map*

theorem *induces-unit-counit-adjunction*:
shows *unit-counit-adjunction C D F G η ε*
unfolding *η-def ε-def*
using *induces-unit-counit-adjunction′* **by** *simp*

lemma *η-naturalitytransformation*:
shows *natural-transformation D D D.map GF.map η*
unfolding *η-def* ..

lemma *ε-naturalitytransformation*:
shows *natural-transformation C C FG.map C.map ε*
unfolding *ε-def* ..

From the defined η and ε we can recover the original φ and ψ.

lemma *φ-in-terms-of-η*:
assumes *D.ide y* **and** «*f* : *F y* →*C x*»

shows $\varphi\ y\ f = G\ f\ \cdot_D\ \eta\ y$
using *assms η-def* **by** (*simp add*: *φ-in-terms-of-ηo*)

lemma *ψ-in-terms-of-ε*:
assumes $C.ide\ x$ **and** $«g : y \rightarrow_D G\ x»$
shows $\psi\ x\ g = \varepsilon\ x\ \cdot_C\ F\ g$
using *assms ε-def* **by** (*simp add*: *ψ-in-terms-of-εo*)

end

16.5 Meta-Adjunctions Induce Left and Right Adjoint Functors

context *meta-adjunction*
begin

interpretation *unit-counit-adjunction C D F G η ε*
using *induces-unit-counit-adjunction η-def ε-def* **by** *auto*

lemma *has-terminal-arrows-from-functor*:
assumes *x*: $C.ide\ x$
shows *terminal-arrow-from-functor D C F* $(G\ x)\ x\ (\varepsilon\ x)$
and $\bigwedge y'\ f.$ *arrow-from-functor D C F y′ x f*
$\Longrightarrow$ *terminal-arrow-from-functor.the-coext D C F* $(G\ x)\ (\varepsilon\ x)\ y'\ f = \varphi\ y'\ f$
proof −
interpret *εx*: *arrow-from-functor D C F* $‹G\ x›\ x\ ‹\varepsilon\ x›$
using *x ε.preserves-hom G.preserves-ide* **by** *unfold-locales auto*
have *1*: $\bigwedge y'\ f.$ *arrow-from-functor D C F y′ x f* $\Longrightarrow$
$\varepsilon x.is\text{-}coext\ y'\ f\ (\varphi\ y'\ f) \wedge (\forall g'.\ \varepsilon x.is\text{-}coext\ y'\ f\ g' \longrightarrow g' = \varphi\ y'\ f)$
using *x*
by (*metis* (*full-types*) *εx.is-coext-def φ-ψ ψ-in-terms-of-ε arrow-from-functor.arrow φ-in-hom ψ-φ*)
interpret *εx*: *terminal-arrow-from-functor D C F* $‹G\ x›\ x\ ‹\varepsilon\ x›$
using *1* **by** *unfold-locales blast*
show *terminal-arrow-from-functor D C F* $(G\ x)\ x\ (\varepsilon\ x)$ **..**
show $\bigwedge y'\ f.$ *arrow-from-functor D C F y′ x f* $\Longrightarrow \varepsilon x.the\text{-}coext\ y'\ f = \varphi\ y'\ f$
using *1 εx.the-coext-def* **by** *auto*
qed

lemma *has-left-adjoint-functor*:
shows *left-adjoint-functor D C F*
apply *unfold-locales* **using** *has-terminal-arrows-from-functor* **by** *auto*

lemma *has-initial-arrows-to-functor*:
assumes *y*: $D.ide\ y$
shows *initial-arrow-to-functor C D G y* $(F\ y)\ (\eta\ y)$
and $\bigwedge x'\ g.$ *arrow-to-functor C D G y x′ g* $\Longrightarrow$
initial-arrow-to-functor.the-ext C D G $(F\ y)\ (\eta\ y)\ x'\ g = \psi\ x'\ g$

```
  proof −
    interpret ηy: arrow-to-functor C D G y ‹F y› ‹η y›
      using y by unfold-locales auto
    have 1: ⋀x′ g. arrow-to-functor C D G y x′ g ⟹
                   ηy.is-ext x′ g (ψ x′ g) ∧ (∀f′. ηy.is-ext x′ g f′ ⟶ f′ = ψ x′ g)
      using y
      by (metis (full-types) ηy.is-ext-def ψ-φ φ-in-terms-of-η arrow-to-functor.arrow
          ψ-in-hom φ-ψ)
    interpret ηy: initial-arrow-to-functor C D G y ‹F y› ‹η y›
      apply unfold-locales using 1 by blast
    show initial-arrow-to-functor C D G y (F y) (η y) ..
    show ⋀x′ g. arrow-to-functor C D G y x′ g ⟹ ηy.the-ext x′ g = ψ x′ g
      using 1 ηy.the-ext-def by auto
  qed

  lemma has-right-adjoint-functor:
  shows right-adjoint-functor C D G
    apply unfold-locales using has-initial-arrows-to-functor by auto

end
```

16.6 Unit/Counit Adjunctions Induce Meta-Adjunctions

```
context unit-counit-adjunction
begin

  definition φ :: 'd ⇒ 'c ⇒ 'd
  where φ y h = G h ·D η y

  definition ψ :: 'c ⇒ 'd ⇒ 'c
  where ψ x h = ε x ·C F h

  interpretation meta-adjunction C D F G φ ψ
  proof
    fix x :: 'c and y :: 'd and f :: 'c
    assume y: D.ide y and f: «f : F y →C x»
    show 0: «φ y f : y →D G x»
      using f y G.preserves-hom η.preserves-hom φ-def D.ide-in-hom by auto
    show ψ x (φ y f) = f
    proof −
      have ψ x (φ y f) = (ε x ·C F (G f)) ·C F (η y)
        using y f φ-def ψ-def C.comp-assoc by auto
      also have ... = (f ·C ε (F y)) ·C F (η y)
        using y f ε.naturality by auto
      also have ... = f
        using y f εFoFη.map-simp-2 triangle-F C.comp-arr-dom D.ide-in-hom C.comp-assoc
        by fastforce
      finally show ?thesis by auto
    qed
```

```
    next
    fix x :: 'c and y :: 'd and g :: 'd
    assume x: C.ide x and g: «g : y →D G x»
    show «ψ x g : F y →C x» using g x ψ-def by fastforce
    show φ y (ψ x g) = g
    proof −
      have φ y (ψ x g) = (G (ε x) ·D η (G x)) ·D g
        using g x φ-def ψ-def η.naturality [of g] D.comp-assoc by auto
      also have ... = g
        using x g triangle-G D.comp-ide-arr GεoηG.map-simp-ide by auto
      finally show ?thesis by auto
    qed
    next
    fix f :: 'c and g :: 'd and h :: 'c and x :: 'c and x' :: 'c and y :: 'd and y' :: 'd
    assume f: «f : x →C x'» and g: «g : y' →D y» and h: «h : F y →C x»
    show φ y' (f ·C h ·C F g) = G f ·D φ y h ·D g
      using φ-def f g h η.naturality D.comp-assoc by fastforce
  qed

  theorem induces-meta-adjunction:
  shows meta-adjunction C D F G φ ψ ..
```

From the defined φ and ψ we can recover the original η and ε.

```
  lemma η-in-terms-of-φ:
  assumes D.ide y
  shows η y = φ y (F y)
    using assms φ-def D.comp-cod-arr by auto

  lemma ε-in-terms-of-ψ:
  assumes C.ide x
  shows ε x = ψ x (G x)
    using assms ψ-def C.comp-arr-dom by auto

end
```

16.7 Left and Right Adjoint Functors Induce Meta-Adjunctions

A left adjoint functor induces a meta-adjunction, modulo the choice of a right adjoint and counit.

```
context left-adjoint-functor
begin

  definition Go :: 'c ⇒ 'd
  where Go a = (SOME b. ∃ e. terminal-arrow-from-functor D C F b a e)

  definition εo :: 'c ⇒ 'c
  where εo a = (SOME e. terminal-arrow-from-functor D C F (Go a) a e)
```

lemma *Go-εo-terminal*:
assumes ∃ *b e*. *terminal-arrow-from-functor D C F b a e*
shows *terminal-arrow-from-functor D C F* (*Go a*) *a* (*εo a*)
using *assms Go-def εo-def*
someI-ex [*of* λ*b*. ∃ *e*. *terminal-arrow-from-functor D C F b a e*]
someI-ex [*of* λ*e*. *terminal-arrow-from-functor D C F* (*Go a*) *a e*]
by *simp*

The right adjoint G to F takes each arrow f of C to the unique D-coextension of f $\cdot_C$ *εo* (*C.dom f*) along *εo* (*C.cod f*).

definition *G* :: *'c* ⇒ *'d*
where *G f* = (*if C.arr f then*
terminal-arrow-from-functor.the-coext D C F (*Go* (*C.cod f*)) (*εo* (*C.cod f*))
(*Go* (*C.dom f*)) (*f* ·*C εo* (*C.dom f*))
else D.null)

lemma *G-ide*:
assumes *C.ide x*
shows *G x* = *Go x*
proof −
interpret *terminal-arrow-from-functor D C F* ‹*Go x*› *x* ‹*εo x*›
using *assms ex-terminal-arrow Go-εo-terminal* **by** *blast*
have *1*: *arrow-from-functor D C F* (*Go x*) *x* (*εo x*) **..**
have *is-coext* (*Go x*) (*εo x*) (*Go x*)
using *arrow is-coext-def C.in-homE C.comp-arr-dom* **by** *auto*
hence *Go x* = *the-coext* (*Go x*) (*εo x*) **using** *1 the-coext-unique* **by** *blast*
moreover have *εo x* = *C x* (*εo* (*C.dom x*))
using *assms arrow C.comp-ide-arr C.seqI' C.ide-in-hom C.in-homE* **by** *metis*
ultimately show *?thesis* **using** *assms G-def C.cod-dom C.ide-in-hom C.in-homE* **by** *metis*
qed

lemma *G-is-functor*:
shows *functor C D G*
proof
fix *f* :: *'c*
assume ¬*C.arr f*
thus *G f* = *D.null* **using** *G-def* **by** *auto*
next
fix *f* :: *'c*
assume *f*: *C.arr f*
let *?x* = *C.dom f*
let *?x'* = *C.cod f*
interpret *xε*: *terminal-arrow-from-functor D C F* ‹*Go ?x*› ‹*?x*› ‹*εo ?x*›
using *f ex-terminal-arrow Go-εo-terminal* **by** *simp*
interpret *x'ε*: *terminal-arrow-from-functor D C F* ‹*Go ?x'*› ‹*?x'*› ‹*εo ?x'*›
using *f ex-terminal-arrow Go-εo-terminal* **by** *simp*
have *1*: *arrow-from-functor D C F* (*Go ?x*) *?x'* (*C f* (*εo ?x*))
using *f xε.arrow* **by** (*unfold-locales*, *auto*)
have *G f* = *x'ε.the-coext* (*Go ?x*) (*C f* (*εo ?x*)) **using** *f G-def* **by** *simp*

```
  hence Gf: «G f : Go ?x →D Go ?x'» ∧ f ·C εo ?x = εo ?x' ·C F (G f)
    using 1 x'ε.the-coext-prop by simp
  show D.arr (G f) using Gf by auto
  show D.dom (G f) = G ?x using f Gf G-ide by auto
  show D.cod (G f) = G ?x' using f Gf G-ide by auto
  next
  fix f f' :: 'c
  assume ff': C.arr (C f' f)
  have f: C.arr f using ff' by auto
  let ?x = C.dom f
  let ?x' = C.cod f
  let ?x'' = C.cod f'
  interpret xε: terminal-arrow-from-functor D C F ‹Go ?x› ‹?x› ‹εo ?x›
    using f ex-terminal-arrow Go-εo-terminal by simp
  interpret x'ε: terminal-arrow-from-functor D C F ‹Go ?x'› ‹?x'› ‹εo ?x'›
    using f ex-terminal-arrow Go-εo-terminal by simp
  interpret x''ε: terminal-arrow-from-functor D C F ‹Go ?x''› ‹?x''› ‹εo ?x''›
    using ff' ex-terminal-arrow Go-εo-terminal by auto
  have 1: arrow-from-functor D C F (Go ?x) ?x' (f ·C εo ?x)
     using f xε.arrow by (unfold-locales, auto)
  have 2: arrow-from-functor D C F (Go ?x') ?x'' (f' ·C εo ?x')
     using ff' x'ε.arrow by (unfold-locales, auto)
  have G f = x'ε.the-coext (Go ?x) (C f (εo ?x))
    using f G-def by simp
  hence Gf: D.in-hom (G f) (Go ?x) (Go ?x') ∧ f ·C εo ?x = εo ?x' ·C F (G f)
    using 1 x'ε.the-coext-prop by simp
  have G f' = x''ε.the-coext (Go ?x') (f' ·C εo ?x')
    using ff' G-def by auto
  hence Gf': «G f' : Go (C.cod f) →D Go (C.cod f')» ∧ f' ·C εo ?x' = εo ?x'' ·C F (G f')
    using 2 x''ε.the-coext-prop by simp
  show G (f' ·C f) = G f' ·D G f
  proof −
    have x''ε.is-coext (Go ?x) ((f' ·C f) ·C εo ?x) (G f' ·D G f)
    proof −
      have 3: «G f' ·D G f : Go (C.dom f) →D Go (C.cod f')» using 1 2 Gf Gf' by auto
      moreover have (f' ·C f) ·C εo ?x = εo ?x'' ·C F (G f' ·D G f)
        by (metis 3 C.comp-assoc D.in-homE Gf Gf' preserves-comp)
      ultimately show ?thesis using x''ε.is-coext-def by auto
    qed
    moreover have arrow-from-functor D C F (Go ?x) ?x'' ((f' ·C f) ·C εo ?x)
      using ff' xε.arrow by unfold-locales blast
    ultimately show ?thesis
      using ff' G-def x''ε.the-coext-unique C.seqE C.cod-comp C.dom-comp by auto
  qed
qed

interpretation G: functor C D G using G-is-functor by auto

lemma G-simp:
```

assumes *C.arr f*
shows *G f = terminal-arrow-from-functor.the-coext D C F (Go (C.cod f)) (εo (C.cod f))*
(Go (C.dom f)) (f ·C εo (C.dom f))
using *assms G-def* **by** *simp*

interpretation *idC*: *identity-functor C* **..**
interpretation *GF*: *composite-functor C D C G F* **..**

interpretation *ε*: *transformation-by-components C C GF.map C.map εo*
proof
fix *x* :: *'c*
assume *x*: *C.ide x*
show «*εo x* : *GF.map x* →*C* *C.map x*»
proof −
interpret *terminal-arrow-from-functor D C F ‹Go x› x ‹εo x›*
using *x Go-εo-terminal ex-terminal-arrow* **by** *simp*
show *?thesis* **using** *x G-ide arrow* **by** *auto*
qed
next
fix *f* :: *'c*
assume *f*: *C.arr f*
show *εo (C.cod f) ·C GF.map f = C.map f ·C εo (C.dom f)*
proof −
let *?x = C.dom f*
let *?x′ = C.cod f*
interpret *xε*: *terminal-arrow-from-functor D C F ‹Go ?x› ?x ‹εo ?x›*
using *f Go-εo-terminal ex-terminal-arrow* **by** *simp*
interpret *x′ε*: *terminal-arrow-from-functor D C F ‹Go ?x′› ?x′ ‹εo ?x′›*
using *f Go-εo-terminal ex-terminal-arrow* **by** *simp*
have *1*: *arrow-from-functor D C F (Go ?x) ?x′ (C f (εo ?x))*
using *f xε.arrow* **by** *unfold-locales auto*
have *G f = x′ε.the-coext (Go ?x) (f ·C εo ?x)*
using *f G-simp* **by** *blast*
hence *x′ε.is-coext (Go ?x) (f ·C εo ?x) (G f)*
using *1 x′ε.the-coext-prop x′ε.is-coext-def* **by** *auto*
thus *?thesis*
using *f x′ε.is-coext-def* **by** *simp*
qed
qed

definition *ψ*
where *ψ x h = C (ε.map x) (F h)*

lemma *ψ-in-hom*:
assumes *C.ide x* **and** «*g* : *y* →*D* *G x*»
shows «*ψ x g* : *F y* →*C* *x*»
unfolding *ψ-def* **using** *assms ε.maps-ide-in-hom* **by** *auto*

lemma *ψ-natural*:

assumes f: «$f : x \rightarrow_C x'$» **and** g: «$g : y' \rightarrow_D y$» **and** h: «$h : y \rightarrow_D G\ x$»
shows $f \cdot_C \psi\ x\ h \cdot_C F\ g = \psi\ x'\ ((G\ f \cdot_D h) \cdot_D g)$
proof −
  have $f \cdot_C \psi\ x\ h \cdot_C F\ g = f \cdot_C (\varepsilon.map\ x \cdot_C F\ h) \cdot_C F\ g$
    unfolding ψ-*def* **by** *auto*
  also have ... $= (f \cdot_C \varepsilon.map\ x) \cdot_C F\ h \cdot_C F\ g$
    using *C.comp-assoc* **by** *fastforce*
  also have ... $= (f \cdot_C \varepsilon.map\ x) \cdot_C F\ (h \cdot_D g)$
    using *g h* **by** *fastforce*
  also have ... $= (\varepsilon.map\ x' \cdot_C F\ (G\ f)) \cdot_C F\ (h \cdot_D g)$
    using *f* ε*.naturality* **by** *auto*
  also have ... $= \varepsilon.map\ x' \cdot_C F\ ((G\ f \cdot_D h) \cdot_D g)$
    using *f g h C.comp-assoc* **by** *fastforce*
  also have ... $= \psi\ x'\ ((G\ f \cdot_D h) \cdot_D g)$
    unfolding ψ-*def* **by** *auto*
  finally show *?thesis* **by** *auto*
qed

lemma ψ-*inverts-coext*:
assumes x: *C.ide x* **and** g: «$g : y \rightarrow_D G\ x$»
shows *arrow-from-functor.is-coext D C F* (*G x*) (*ε.map x*) *y* (ψ *x g*) *g*
proof −
  interpret $x\varepsilon$: *arrow-from-functor D C F* ‹*G x*› *x* ‹*ε.map x*›
    using *x* ε*.maps-ide-in-hom* **by** *unfold-locales auto*
  show *xε.is-coext y* (ψ *x g*) *g*
    using *x g* ψ*-def xε.is-coext-def G-ide* **by** *blast*
qed

lemma ψ-*invertible*:
assumes y: *D.ide y* **and** f: «$f : F\ y \rightarrow_C x$»
shows $\exists!g.$ «$g : y \rightarrow_D G\ x$» $\wedge$ $\psi\ x\ g = f$
proof
  have x: *C.ide x* **using** *f* **by** *auto*
  interpret $x\varepsilon$: *terminal-arrow-from-functor D C F* ‹*Go x*› *x* ‹*εo x*›
    using *x ex-terminal-arrow Go-εo-terminal* **by** *auto*
  have *1*: *arrow-from-functor D C F y x f*
    using *y f* **by** (*unfold-locales*, *auto*)
  let *?g* = *xε.the-coext y f*
  have ψ *x ?g* = *f*
    using *1 x y* ψ*-def xε.the-coext-prop G-ide* ψ*-inverts-coext xε.is-coext-def* **by** *simp*
  thus «*?g* : *y* $\rightarrow_D$ *G x*» $\wedge$ ψ *x ?g* = *f*
    using *1 x xε.the-coext-prop G-ide* **by** *simp*
  show $\bigwedge g'.$ «g' : *y* $\rightarrow_D$ *G x*» $\wedge$ ψ *x g'* = *f* $\Longrightarrow$ *g'* = *?g*
    using *1 x y* ψ*-inverts-coext G-ide xε.the-coext-unique* **by** *force*
qed

definition φ
where $\varphi\ y\ f$ = (*THE g.* «$g : y \rightarrow_D G\ (C.cod\ f)$» $\wedge$ $\psi\ (C.cod\ f)\ g = f$)

```
  lemma φ-in-hom:
  assumes D.ide y and «f : F y →C x»
  shows «φ y f : y →D G x»
    using assms ψ-invertible φ-def theI' [of λg. «g : y →D G x» ∧ ψ x g = f]
    by auto

  lemma φ-ψ:
  assumes C.ide x and «g : y →D G x»
  shows φ y (ψ x g) = g
  proof −
    have φ y (ψ x g) = (THE g'. «g' : y →D G x» ∧ ψ x g' = ψ x g)
    proof −
      have C.cod (ψ x g) = x
        using assms ψ-in-hom by auto
      thus ?thesis
        using φ-def by auto
    qed
    moreover have ∃!g'. «g' : y →D G x» ∧ ψ x g' = ψ x g
      using assms ψ-in-hom ψ-invertible D.ide-dom by blast
    ultimately show φ y (ψ x g) = g
      using assms(2) by auto
  qed

  lemma ψ-φ:
  assumes D.ide y and «f : F y →C x»
  shows ψ x (φ y f) = f
    using assms ψ-invertible φ-def theI' [of λg. «g : y →D G x» ∧ ψ x g = f]
    by auto

  lemma φ-natural:
  assumes «f : x →C x'» and «g : y' →D y» and «h : F y →C x»
  shows φ y' (f ·C h ·C F g) = (G f ·D φ y h) ·D g
  proof −
    have C.ide x' ∧ D.ide y ∧ D.in-hom (φ y h) y (G x)
      using assms φ-in-hom by auto
    thus ?thesis
      using assms D.comp-in-homI G.preserves-hom ψ-natural [of f x x' g y' y φ y h] φ-ψ ψ-φ
      by auto
  qed

  theorem induces-meta-adjunction:
  shows meta-adjunction C D F G φ ψ
    using φ-in-hom ψ-in-hom φ-ψ ψ-φ φ-natural D.comp-assoc
    by unfold-locales auto

end
```

A right adjoint functor induces a meta-adjunction, modulo the choice of a left adjoint and unit.

context *right-adjoint-functor*
begin

definition *Fo* :: *'d* ⇒ *'c*
where *Fo y* = (*SOME x*. ∃ *u*. *initial-arrow-to-functor C D G y x u*)

definition *ηo* :: *'d* ⇒ *'d*
where *ηo y* = (*SOME u*. *initial-arrow-to-functor C D G y* (*Fo y*) *u*)

lemma *Fo-ηo-initial*:
assumes ∃ *x u*. *initial-arrow-to-functor C D G y x u*
shows *initial-arrow-to-functor C D G y* (*Fo y*) (*ηo y*)
using *assms Fo-def ηo-def*
someI-ex [*of* λ*x*. ∃ *u*. *initial-arrow-to-functor C D G y x u*]
someI-ex [*of* λ*u*. *initial-arrow-to-functor C D G y* (*Fo y*) *u*]
by *simp*

The left adjoint F to g takes each arrow g of D to the unique C-extension of $\eta o\ (D.cod\ g)\ \cdot_D\ g$ along $\eta o\ (D.dom\ g)$.

definition *F* :: *'d* ⇒ *'c*
where *F g* = (*if D.arr g then*
initial-arrow-to-functor.the-ext C D G (*Fo* (*D.dom g*)) (*ηo* (*D.dom g*))
(*Fo* (*D.cod g*)) (*ηo* (*D.cod g*) ·*D g*)
else C.null)

lemma *F-ide*:
assumes *D.ide y*
shows *F y* = *Fo y*
proof −
interpret *initial-arrow-to-functor C D G y* ‹*Fo y*› ‹*ηo y*›
using *assms ex-initial-arrow Fo-ηo-initial* **by** *blast*
have *1*: *arrow-to-functor C D G y* (*Fo y*) (*ηo y*) **..**
have *is-ext* (*Fo y*) (*ηo y*) (*Fo y*)
unfolding *is-ext-def* **using** *arrow D.comp-ide-arr* [*of G* (*Fo y*) *ηo y*] **by** *force*
hence *Fo y* = *the-ext* (*Fo y*) (*ηo y*)
using *1 the-ext-unique* **by** *blast*
moreover have *ηo y* = *D* (*ηo* (*D.cod y*)) *y*
using *assms arrow D.comp-arr-ide D.comp-arr-dom* **by** *auto*
ultimately show *?thesis*
using *assms F-def D.dom-cod D.in-homE D.ide-in-hom* **by** *metis*
qed

lemma *F-is-functor*:
shows *functor D C F*
proof
fix *g* :: *'d*
assume ¬*D.arr g*
thus *F g* = *C.null* **using** *F-def* **by** *auto*
next

fix $g :: 'd$
assume g: $D.arr\ g$
let $?y = D.dom\ g$
let $?y' = D.cod\ g$
interpret $y\eta$: *initial-arrow-to-functor C D G ?y* ‹*Fo ?y*› ‹ηo *?y*›
  using *g ex-initial-arrow Fo-ηo-initial* **by** *simp*
interpret $y'\eta$: *initial-arrow-to-functor C D G ?y'* ‹*Fo ?y'*› ‹ηo *?y'*›
  using *g ex-initial-arrow Fo-ηo-initial* **by** *simp*
have *1*: *arrow-to-functor C D G ?y* (*Fo ?y'*) (*D* (ηo *?y'*) *g*)
  using *g y'η.arrow* **by** *unfold-locales auto*
have *F g* = *yη.the-ext* (*Fo ?y'*) (*D* (ηo *?y'*) *g*)
  using *g F-def* **by** *simp*
hence *Fg*: «*F g* : *Fo ?y* $\rightarrow_C$ *Fo ?y'*» $\wedge$ ηo *?y'* $\cdot_D$ *g* = *G* (*F g*) $\cdot_D$ ηo *?y*
  using *1 yη.the-ext-prop* **by** *simp*
show *C.arr* (*F g*) **using** *Fg* **by** *auto*
show *C.dom* (*F g*) = *F ?y* **using** *Fg g F-ide* **by** *auto*
show *C.cod* (*F g*) = *F ?y'* **using** *Fg g F-ide* **by** *auto*
next
fix $g :: 'd$
fix $g' :: 'd$
assume g': $D.arr\ (D\ g'\ g)$
have g: $D.arr\ g$ **using** g' **by** *auto*
let $?y = D.dom\ g$
let $?y' = D.cod\ g$
let $?y'' = D.cod\ g'$
interpret $y\eta$: *initial-arrow-to-functor C D G ?y* ‹*Fo ?y*› ‹ηo *?y*›
  using *g ex-initial-arrow Fo-ηo-initial* **by** *simp*
interpret $y'\eta$: *initial-arrow-to-functor C D G ?y'* ‹*Fo ?y'*› ‹ηo *?y'*›
  using *g ex-initial-arrow Fo-ηo-initial* **by** *simp*
interpret $y''\eta$: *initial-arrow-to-functor C D G ?y''* ‹*Fo ?y''*› ‹ηo *?y''*›
  using g' *ex-initial-arrow Fo-ηo-initial* **by** *auto*
have *1*: *arrow-to-functor C D G ?y* (*Fo ?y'*) (ηo *?y'* $\cdot_D$ *g*)
  using *g y'η.arrow* **by** *unfold-locales auto*
have *F g* = *yη.the-ext* (*Fo ?y'*) (ηo *?y'* $\cdot_D$ *g*)
  using *g F-def* **by** *simp*
hence *Fg*: «*F g* : *Fo ?y* $\rightarrow_C$ *Fo ?y'*» $\wedge$ ηo *?y'* $\cdot_D$ *g* = *G* (*F g*) $\cdot_D$ ηo *?y*
  using *1 yη.the-ext-prop* **by** *simp*
have *2*: *arrow-to-functor C D G ?y'* (*Fo ?y''*) (ηo *?y''* $\cdot_D$ g')
  using g' *y''η.arrow* **by** *unfold-locales auto*
have *F* g' = *y'η.the-ext* (*Fo ?y''*) (ηo *?y''* $\cdot_D$ g')
  using g' *F-def* **by** *auto*
hence *Fg'*: «*F* g' : *Fo ?y'* $\rightarrow_C$ *Fo ?y''*» $\wedge$ ηo *?y''* $\cdot_D$ g' = *G* (*F* g') $\cdot_D$ ηo *?y'*
  using *2 y'η.the-ext-prop* **by** *simp*
show *F* ($g' \cdot_D$ *g*) = *F* $g' \cdot_C$ *F g*
proof −
  have *yη.is-ext* (*Fo ?y''*) (ηo *?y''* $\cdot_D$ $g' \cdot_D$ *g*) (*F* $g' \cdot_C$ *F g*)
  proof −
    have *3*: «*F* $g' \cdot_C$ *F g* : *Fo ?y* $\rightarrow_C$ *Fo ?y''*» **using** *1 2 Fg Fg'* **by** *auto*
    moreover have ηo *?y''* $\cdot_D$ $g' \cdot_D$ *g* = *G* (*F* $g' \cdot_C$ *F g*) $\cdot_D$ ηo *?y*

```
      using Fg Fg' g g' 3 y''η.arrow
      by (metis C.arrI D.comp-assoc preserves-comp)
    ultimately show ?thesis using yη.is-ext-def by auto
  qed
  moreover have arrow-to-functor C D G ?y (Fo ?y'') (ηo ?y'' ·_D g' ·_D g)
    using g g' y''η.arrow by unfold-locales auto
  ultimately show ?thesis
    using g g' F-def yη.the-ext-unique D.dom-comp D.cod-comp by auto
 qed
qed

interpretation F: functor D C F using F-is-functor by auto

lemma F-simp:
assumes D.arr g
shows F g = initial-arrow-to-functor.the-ext C D G (Fo (D.dom g)) (ηo (D.dom g))
                                             (Fo (D.cod g)) (ηo (D.cod g) ·_D g)
  using assms F-def by simp

interpretation FG: composite-functor D C D F G ..

interpretation η: transformation-by-components D D D.map FG.map ηo
proof
  fix y :: 'd
  assume y: D.ide y
  show «ηo y : D.map y →_D FG.map y»
  proof −
    interpret initial-arrow-to-functor C D G y ‹Fo y› ‹ηo y›
      using y Fo-ηo-initial ex-initial-arrow by simp
    show ?thesis using y F-ide arrow by auto
  qed
  next
  fix g :: 'd
  assume g: D.arr g
  show ηo (D.cod g) ·_D D.map g = FG.map g ·_D ηo (D.dom g)
  proof −
    let ?y = D.dom g
    let ?y' = D.cod g
    interpret yη: initial-arrow-to-functor C D G ?y ‹Fo ?y› ‹ηo ?y›
      using g Fo-ηo-initial ex-initial-arrow by simp
    interpret y'η: initial-arrow-to-functor C D G ?y' ‹Fo ?y'› ‹ηo ?y'›
      using g Fo-ηo-initial ex-initial-arrow by simp
    have arrow-to-functor C D G ?y (Fo ?y') (ηo ?y' ·_D g)
      using g y'η.arrow by unfold-locales auto
    moreover have F g = yη.the-ext (Fo ?y') (ηo ?y' ·_D g)
      using g F-simp by blast
    ultimately have yη.is-ext (Fo ?y') (ηo ?y' ·_D g) (F g)
      using yη.the-ext-prop yη.is-ext-def by auto
    thus ?thesis
```

```
    using g yη.is-ext-def by simp
  qed
qed

definition φ
where φ y h = D (G h) (η.map y)

lemma φ-in-hom:
assumes y: D.ide y and f: «f : F y →C x»
shows «φ y f : y →D G x»
  unfolding φ-def using assms η.maps-ide-in-hom by auto

lemma φ-natural:
assumes f: «f : x →C x'» and g: «g : y' →D y» and h: «h : F y →C x»
shows φ y' (f ·C h ·C F g) = (G f ·D φ y h) ·D g
proof −
  have (G f ·D φ y h) ·D g = (G f ·D G h ·D η.map y) ·D g
    unfolding φ-def by auto
  also have ... = (G f ·D G h) ·D η.map y ·D g
    using D.comp-assoc by fastforce
  also have ... = G (f ·C h) ·D G (F g) ·D η.map y'
    using f g h η.naturality by fastforce
  also have ... = (G (f ·C h) ·D G (F g)) ·D η.map y'
    using D.comp-assoc by fastforce
  also have ... = G (f ·C h ·C F g) ·D η.map y'
    using f g h D.comp-assoc by fastforce
  also have ... = φ y' (f ·C h ·C F g)
    unfolding φ-def by auto
  finally show ?thesis by auto
qed

lemma φ-inverts-ext:
assumes y: D.ide y and f: «f : F y →C x»
shows arrow-to-functor.is-ext C D G (F y) (η.map y) x (φ y f) f
proof −
  interpret yη: arrow-to-functor C D G y ‹F y› ‹η.map y›
    using y η.maps-ide-in-hom by unfold-locales auto
  show yη.is-ext x (φ y f) f
    using f y φ-def yη.is-ext-def F-ide by blast
qed

lemma φ-invertible:
assumes x: C.ide x and g: «g : y →D G x»
shows ∃!f. «f : F y →C x» ∧ φ y f = g
proof
  have y: D.ide y using g by auto
  interpret yη: initial-arrow-to-functor C D G y ‹Fo y› ‹ηo y›
    using y ex-initial-arrow Fo-ηo-initial by auto
  have 1: arrow-to-functor C D G y x g
```

```
    using x g by (unfold-locales, auto)
  let ?f = yη.the-ext x g
  have φ y ?f = g
    using φ-def yη.the-ext-prop 1 F-ide x y φ-inverts-ext yη.is-ext-def by fastforce
  moreover have «?f : F y →C x»
    using 1 y yη.the-ext-prop F-ide by simp
  ultimately show «?f : F y →C x» ∧ φ y ?f = g by auto
  show ⋀f'. «f' : F y →C x» ∧ φ y f' = g ⟹ f' = ?f
    using 1 y φ-inverts-ext yη.the-ext-unique F-ide by force
qed

definition ψ
where ψ x g = (THE f. «f : F (D.dom g) →C x» ∧ φ (D.dom g) f = g)

lemma ψ-in-hom:
assumes C.ide x and «g : y →D G x»
shows C.in-hom (ψ x g) (F y) x
  using assms φ-invertible ψ-def theI' [of λf. «f : F y →C x» ∧ φ y f = g]
  by auto

lemma ψ-φ:
assumes D.ide y and «f : F y →C x»
shows ψ x (φ y f) = f
proof −
  have D.dom (φ y f) = y using assms φ-in-hom by blast
  hence ψ x (φ y f) = (THE f'. «f' : F y →C x» ∧ φ y f' = φ y f)
    using ψ-def by auto
  moreover have ∃!f'. «f' : F y →C x» ∧ φ y f' = φ y f
    using assms φ-in-hom φ-invertible C.ide-cod by blast
  ultimately show ?thesis using assms(2) by auto
qed

lemma φ-ψ:
assumes C.ide x and «g : y →D G x»
shows φ y (ψ x g) = g
  using assms φ-invertible ψ-def theI' [of λf. «f : F y →C x» ∧ φ y f = g]
  by auto

theorem induces-meta-adjunction:
shows meta-adjunction C D F G φ ψ
  using φ-in-hom ψ-in-hom φ-ψ ψ-φ φ-natural D.comp-assoc
  by (unfold-locales, auto)

end
```

16.8 Meta-Adjunctions Induce Hom-Adjunctions

To obtain a hom-adjunction from a meta-adjunction, we need to exhibit hom-functors from C and D to a common set category S, so it is necessary to apply an actual concrete construction of such a category. We use the replete set category generated by the disjoint sum $'c + 'd$ of the arrow types of C and D.

context *meta-adjunction*
begin

interpretation S: *replete-setcat* ‹$TYPE('c+'d)$› **.**

definition $inC :: 'c \Rightarrow ('c+'d)$ *setcat.arr*
where $inC \equiv S.UP\ o\ Inl$

definition $inD :: 'd \Rightarrow ('c+'d)$ *setcat.arr*
where $inD \equiv S.UP\ o\ Inr$

interpretation S: *replete-setcat* ‹$TYPE('c+'d)$› **.**
interpretation Cop: *dual-category* C **..**
interpretation Dop: *dual-category* D **..**
interpretation $CopxC$: *product-category* $Cop.comp\ C$ **..**
interpretation $DopxD$: *product-category* $Dop.comp\ D$ **..**
interpretation $DopxC$: *product-category* $Dop.comp\ C$ **..**
interpretation $HomC$: *hom-functor* $C\ S.comp\ S.setp$ ‹λ-. inC›
proof
  show $\bigwedge f.\ C.arr\ f \Longrightarrow inC\ f \in S.Univ$
    unfolding *inC-def* **using** *S.UP-mapsto* **by** *auto*
  thus $\bigwedge b\ a.\ [\![C.ide\ b;\ C.ide\ a]\!] \Longrightarrow inC\ `\ C.hom\ b\ a \subseteq S.Univ$
    by *blast*
  show $\bigwedge b\ a.\ [\![C.ide\ b;\ C.ide\ a]\!] \Longrightarrow inj\text{-}on\ inC\ (C.hom\ b\ a)$
    unfolding *inC-def*
    using *S.inj-UP*
    by (*metis injD inj-Inl inj-compose inj-on-def*)
qed
interpretation $HomD$: *hom-functor* $D\ S.comp\ S.setp$ ‹λ-. inD›
proof
  show $\bigwedge f.\ D.arr\ f \Longrightarrow inD\ f \in S.Univ$
    unfolding *inD-def* **using** *S.UP-mapsto* **by** *auto*
  thus $\bigwedge b\ a.\ [\![D.ide\ b;\ D.ide\ a]\!] \Longrightarrow inD\ `\ D.hom\ b\ a \subseteq S.Univ$
    by *blast*
  show $\bigwedge b\ a.\ [\![D.ide\ b;\ D.ide\ a]\!] \Longrightarrow inj\text{-}on\ inD\ (D.hom\ b\ a)$
    unfolding *inD-def*
    using *S.inj-UP*
    by (*metis injD inj-Inr inj-compose inj-on-def*)
qed
interpretation Fop: *dual-functor* $D\ C\ F$ **..**
interpretation $FopxC$: *product-functor* $Dop.comp\ C\ Cop.comp\ C\ Fop.map\ C.map$ **..**
interpretation $DopxG$: *product-functor* $Dop.comp\ C\ Dop.comp\ D\ Dop.map\ G$ **..**
interpretation $Hom\text{-}FopxC$: *composite-functor* $DopxC.comp\ CopxC.comp\ S.comp$

FopxC.map HomC.map ..
interpretation *Hom-DopxG*: *composite-functor DopxC.comp DopxD.comp S.comp*
DopxG.map HomD.map ..

lemma *inC-ψ* [*simp*]:
assumes *C.ide b* **and** *C.ide a* **and** $x \in$ *inC* ‘ *C.hom b a*
shows *inC* (*HomC.ψ* (*b*, *a*) *x*) = *x*
using *assms* **by** *auto*

lemma *ψ-inC* [*simp*]:
assumes *C.arr f*
shows *HomC.ψ* (*C.dom f*, *C.cod f*) (*inC f*) = *f*
using *assms HomC.ψ-φ* **by** *blast*

lemma *inD-ψ* [*simp*]:
assumes *D.ide b* **and** *D.ide a* **and** $x \in$ *inD* ‘ *D.hom b a*
shows *inD* (*HomD.ψ* (*b*, *a*) *x*) = *x*
using *assms* **by** *auto*

lemma *ψ-inD* [*simp*]:
assumes *D.arr f*
shows *HomD.ψ* (*D.dom f*, *D.cod f*) (*inD f*) = *f*
using *assms HomD.ψ-φ* **by** *blast*

lemma *Hom-FopxC-simp*:
assumes *DopxC.arr gf*
shows *Hom-FopxC.map gf* =
S.mkArr (*HomC.set* (*F* (*D.cod* (*fst gf*)), *C.dom* (*snd gf*)))
(*HomC.set* (*F* (*D.dom* (*fst gf*)), *C.cod* (*snd gf*)))
(*inC* ∘ (λ*h*. *snd gf* ·$_C$ *h* ·$_C$ *F* (*fst gf*))
∘ *HomC.ψ* (*F* (*D.cod* (*fst gf*)), *C.dom* (*snd gf*)))
using *assms HomC.map-def* **by** *simp*

lemma *Hom-DopxG-simp*:
assumes *DopxC.arr gf*
shows *Hom-DopxG.map gf* =
S.mkArr (*HomD.set* (*D.cod* (*fst gf*), *G* (*C.dom* (*snd gf*))))
(*HomD.set* (*D.dom* (*fst gf*), *G* (*C.cod* (*snd gf*))))
(*inD* ∘ (λ*h*. *G* (*snd gf*) ·$_D$ *h* ·$_D$ *fst gf*)
∘ *HomD.ψ* (*D.cod* (*fst gf*), *G* (*C.dom* (*snd gf*))))
using *assms HomD.map-def* **by** *simp*

definition Φ*o*
where Φ*o yx* = *S.mkArr* (*HomC.set* (*F* (*fst yx*), *snd yx*))
(*HomD.set* (*fst yx*, *G* (*snd yx*)))
(*inD o φ* (*fst yx*) *o HomC.ψ* (*F* (*fst yx*), *snd yx*))

lemma Φ*o-in-hom*:
assumes *yx*: *DopxC.ide yx*

```
shows «Φo yx : Hom-FopxC.map yx →S Hom-DopxG.map yx»
proof −
  have Hom-FopxC.map yx = S.mkIde (HomC.set (F (fst yx), snd yx))
    using yx HomC.map-ide by auto
  moreover have Hom-DopxG.map yx = S.mkIde (HomD.set (fst yx, G (snd yx)))
    using yx HomD.map-ide by auto
  moreover have
      «S.mkArr (HomC.set (F (fst yx), snd yx)) (HomD.set (fst yx, G (snd yx)))
            (inD ∘ φ (fst yx) ∘ HomC.ψ (F (fst yx), snd yx)) :
         S.mkIde (HomC.set (F (fst yx), snd yx))
           →S S.mkIde (HomD.set (fst yx, G (snd yx)))»
  proof (intro S.mkArr-in-hom)
    show HomC.set (F (fst yx), snd yx) ⊆ S.Univ using yx HomC.set-subset-Univ by simp
    show HomD.set (fst yx, G (snd yx)) ⊆ S.Univ using yx HomD.set-subset-Univ by simp
    show inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx)
           ∈ HomC.set (F (fst yx), snd yx) → HomD.set (fst yx, G (snd yx))
    proof
      fix x
      assume x: x ∈ HomC.set (F (fst yx), snd yx)
      show (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx)) x
             ∈ HomD.set (fst yx, G (snd yx))
        using x yx HomC.ψ-mapsto [of F (fst yx) snd yx]
              φ-in-hom [of fst yx] HomD.φ-mapsto [of fst yx G (snd yx)]
        by auto
    qed
  qed
  ultimately show ?thesis using Φo-def by auto
qed

interpretation Φ: transformation-by-components
                  DopxC.comp S.comp Hom-FopxC.map Hom-DopxG.map Φo
proof
  fix yx
  assume yx: DopxC.ide yx
  show «Φo yx : Hom-FopxC.map yx →S Hom-DopxG.map yx»
    using yx Φo-in-hom by auto
  next
  fix gf
  assume gf: DopxC.arr gf
  show S.comp (Φo (DopxC.cod gf)) (Hom-FopxC.map gf)
         = S.comp (Hom-DopxG.map gf) (Φo (DopxC.dom gf))
  proof −
    let ?g = fst gf
    let ?f = snd gf
    let ?x = C.dom ?f
    let ?x' = C.cod ?f
    let ?y = D.cod ?g
    let ?y' = D.dom ?g
    let ?Fy = F ?y
```

let *?Fy′ = F ?y′*
let *?Fg = F ?g*
let *?Gx = G ?x*
let *?Gx′ = G ?x′*
let *?Gf = G ?f*
have *1*: *S.arr (Hom-FopxC.map gf) ∧*
Hom-FopxC.map gf = S.mkArr (HomC.set (?Fy, ?x)) (HomC.set (?Fy′, ?x′))
(inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x))
using *gf Hom-FopxC.preserves-arr Hom-FopxC-simp* **by** *blast*
have *2*: *S.arr (Φo (DopxC.cod gf)) ∧*
Φo (DopxC.cod gf) = S.mkArr (HomC.set (?Fy′, ?x′)) (HomD.set (?y′, ?Gx′))
(inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
using *gf Φo-in-hom [of DopxC.cod gf] Φo-def [of DopxC.cod gf] φ-in-hom*
by *auto*
have *3*: *S.arr (Φo (DopxC.dom gf)) ∧*
Φo (DopxC.dom gf) = S.mkArr (HomC.set (?Fy, ?x)) (HomD.set (?y, ?Gx))
(inD o φ ?y o HomC.ψ (?Fy, ?x))
using *gf Φo-in-hom [of DopxC.dom gf] Φo-def [of DopxC.dom gf] φ-in-hom*
by *auto*
have *4*: *S.arr (Hom-DopxG.map gf) ∧*
Hom-DopxG.map gf = S.mkArr (HomD.set (?y, ?Gx)) (HomD.set (?y′, ?Gx′))
(inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
using *gf Hom-DopxG.preserves-arr Hom-DopxG-simp* **by** *blast*
have *5*: *S.seq (Φo (DopxC.cod gf)) (Hom-FopxC.map gf) ∧*
S.comp (Φo (DopxC.cod gf)) (Hom-FopxC.map gf)
= S.mkArr (HomC.set (?Fy, ?x)) (HomD.set (?y′, ?Gx′))
((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x)))
by (*metis gf 1 2 DopxC.arr-iff-in-hom DopxC.ide-cod Hom-FopxC.preserves-hom*
S.comp-mkArr S.seqI′ Φo-in-hom)
have *6*: *S.comp (Hom-DopxG.map gf) (Φo (DopxC.dom gf))*
= S.mkArr (HomC.set (?Fy, ?x)) (HomD.set (?y′, ?Gx′))
((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
o (inD o φ ?y o HomC.ψ (?Fy, ?x)))
by (*metis 3 4 S.comp-mkArr*)
have *7*:
restrict ((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x))) (HomC.set (?Fy, ?x))
= restrict ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
o (inD o φ ?y o HomC.ψ (?Fy, ?x))) (HomC.set (?Fy, ?x))
proof (*intro restrict-ext*)
show *⋀h. h ∈ HomC.set (?Fy, ?x) ⟹*
((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x))) h
= ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
o (inD o φ ?y o HomC.ψ (?Fy, ?x))) h
proof −
fix *h*
assume *h*: *h ∈ HomC.set (?Fy, ?x)*

```
      have ψh: «HomC.ψ (?Fy, ?x) h : ?Fy →C ?x»
        using gf h HomC.ψ-mapsto [of ?Fy ?x] CopxC.ide-char by auto
      show ((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
              o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x))) h
              = ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
                  o (inD o φ ?y o HomC.ψ (?Fy, ?x))) h
      proof −
        have
          ((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
            o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x))) h
            = inD (φ ?y′ (?f ·C HomC.ψ (?Fy, ?x) h ·C ?Fg))
          using gf ψh HomC.φ-mapsto HomC.ψ-mapsto φ-in-hom
              ψ-inC [of ?f ·C HomC.ψ (?Fy, ?x) h ·C ?Fg]
          by auto
        also have ... = inD (D ?Gf (D (φ ?y (HomC.ψ (?Fy, ?x) h)) ?g))
          by (metis (no-types, lifting) C.arr-cod C.arr-dom-iff-arr C.arr-iff-in-hom
            C.in-homE D.arr-cod-iff-arr D.arr-iff-in-hom F.preserves-reflects-arr
            φ-naturality ψh)
        also have ... = ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
                          o (inD o φ ?y o HomC.ψ (?Fy, ?x))) h
          using gf ψh φ-in-hom by simp
        finally show ?thesis by auto
      qed
    qed
  qed
  have 8: S.mkArr (HomC.set (?Fy, ?x)) (HomD.set (?y′, ?Gx′))
              ((inD o φ ?y′ o HomC.ψ (?Fy′, ?x′))
                o (inC o (λh. ?f ·C h ·C ?Fg) o HomC.ψ (?Fy, ?x)))
          = S.mkArr (HomC.set (?Fy, ?x)) (HomD.set (?y′, ?Gx′))
                ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
                  o (inD o φ ?y o HomC.ψ (?Fy, ?x)))
    using 5 7 by force
  show ?thesis using 5 6 8 by auto
 qed
qed

lemma Φ-simp:
assumes YX: DopxC.ide yx
shows Φ.map yx =
      S.mkArr (HomC.set (F (fst yx), snd yx)) (HomD.set (fst yx, G (snd yx)))
          (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
  using YX Φo-def by simp

abbreviation Ψo
where Ψo yx ≡ S.mkArr (HomD.set (fst yx, G (snd yx))) (HomC.set (F (fst yx), snd yx))
                  (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))

lemma Ψo-in-hom:
assumes yx: DopxC.ide yx
```

```
shows «Ψo yx : Hom-DopxG.map yx →S Hom-FopxC.map yx»
proof −
  have Hom-FopxC.map yx = S.mkIde (HomC.set (F (fst yx), snd yx))
    using yx HomC.map-ide by auto
  moreover have Hom-DopxG.map yx = S.mkIde (HomD.set (fst yx, G (snd yx)))
    using yx HomD.map-ide by auto
  moreover have «Ψo yx : S.mkIde (HomD.set (fst yx, G (snd yx)))
                       →S S.mkIde (HomC.set (F (fst yx), snd yx))»
  proof (intro S.mkArr-in-hom)
    show HomC.set (F (fst yx), snd yx) ⊆ S.Univ using yx HomC.set-subset-Univ by simp
    show HomD.set (fst yx, G (snd yx)) ⊆ S.Univ using yx HomD.set-subset-Univ by simp
    show inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx))
           ∈ HomD.set (fst yx, G (snd yx)) → HomC.set (F (fst yx), snd yx)
    proof
      fix x
      assume x: x ∈ HomD.set (fst yx, G (snd yx))
      show (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx))) x
             ∈ HomC.set (F (fst yx), snd yx)
        using x yx HomD.ψ-mapsto [of fst yx G (snd yx)] ψ-in-hom [of snd yx]
              HomC.φ-mapsto [of F (fst yx) snd yx]
        by auto
    qed
  qed
  ultimately show ?thesis by auto
qed

lemma Φ-inv:
assumes yx: DopxC.ide yx
shows S.inverse-arrows (Φ.map yx) (Ψo yx)
proof −
  have 1: «Φ.map yx : Hom-FopxC.map yx →S Hom-DopxG.map yx»
    using yx Φ.preserves-hom [of yx yx yx] DopxC.ide-in-hom by blast
  have 2: «Ψo yx : Hom-DopxG.map yx →S Hom-FopxC.map yx»
    using yx Ψo-in-hom by simp
  have 3: Φ.map yx = S.mkArr (HomC.set (F (fst yx), snd yx))
                             (HomD.set (fst yx, G (snd yx)))
                             (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
    using yx Φ-simp by blast
  have antipar: S.antipar (Φ.map yx) (Ψo yx)
    using 1 2 by blast
  moreover have S.ide (S.comp (Ψo yx) (Φ.map yx))
  proof −
    have S.comp (Ψo yx) (Φ.map yx) =
           S.mkArr (HomC.set (F (fst yx), snd yx)) (HomC.set (F (fst yx), snd yx))
                   ((inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))
                     o (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx)))
      using 1 2 3 antipar S.comp-mkArr by auto
    also have
      ... = S.mkArr (HomC.set (F (fst yx), snd yx)) (HomC.set (F (fst yx), snd yx))
```

```
                (λx. x)
  proof −
    have
      S.mkArr (HomC.set (F (fst yx), snd yx)) (HomC.set (F (fst yx), snd yx)) (λx. x)
        = ...
    proof
      show
        S.arr (S.mkArr (HomC.set (F (fst yx), snd yx)) (HomC.set (F (fst yx), snd yx))
              (λx. x))
        using yx HomC.set-subset-Univ by simp
      show ⋀x. x ∈ HomC.set (F (fst yx), snd yx) ⟹
                 x = ((inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))
                      o (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))) x
      proof −
        fix x
        assume x: x ∈ HomC.set (F (fst yx), snd yx)
        have ((inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))
                    o (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))) x
              = inC (ψ (snd yx)) (HomD.ψ (fst yx, G (snd yx))
                    (inD (φ (fst yx)) (HomC.ψ (F (fst yx), snd yx) x)))))
          by simp
        also have ... = inC (ψ (snd yx)) (φ (fst yx)) (HomC.ψ (F (fst yx), snd yx) x)))
          using x yx HomC.ψ-mapsto [of F (fst yx) snd yx] φ-in-hom by force
        also have ... = inC (HomC.ψ (F (fst yx), snd yx) x)
          using x yx HomC.ψ-mapsto [of F (fst yx) snd yx] ψ-φ by force
        also have ... = x using x yx inC-ψ by simp
        finally show x = ((inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))
                          o (inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))) x
          by auto
      qed
    qed
    thus ?thesis by auto
  qed
  also have ... = S.mkIde (HomC.set (F (fst yx), snd yx))
    using yx S.mkIde-as-mkArr HomC.set-subset-Univ by force
  finally have
      S.comp (Ψo yx) (Φ.map yx) = S.mkIde (HomC.set (F (fst yx), snd yx))
    by auto
  thus ?thesis using yx HomC.set-subset-Univ S.ide-mkIde by simp
qed
moreover have S.ide (S.comp (Φ.map yx) (Ψo yx))
proof −
  have S.comp (Φ.map yx) (Ψo yx) =
          S.mkArr (HomD.set (fst yx, G (snd yx))) (HomD.set (fst yx, G (snd yx)))
                  ((inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
                    o (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx))))
    using 1 2 3 S.comp-mkArr antipar by fastforce
  also
    have ... = S.mkArr (HomD.set (fst yx, G (snd yx))) (HomD.set (fst yx, G (snd yx)))
```

```
                    (λx. x)
  proof −
    have
      S.mkArr (HomD.set (fst yx, G (snd yx))) (HomD.set (fst yx, G (snd yx))) (λx. x)
        = ...
    proof
      show
        S.arr (S.mkArr (HomD.set (fst yx, G (snd yx))) (HomD.set (fst yx, G (snd yx)))
          (λx. x))
        using yx HomD.set-subset-Univ by simp
      show ⋀x. x ∈ (HomD.set (fst yx, G (snd yx))) ⟹
            x = ((inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
              o (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))) x
      proof −
        fix x
        assume x: x ∈ HomD.set (fst yx, G (snd yx))
        have ((inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
                o (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))) x
              = inD (φ (fst yx)) (HomC.ψ (F (fst yx), snd yx)
                (inC (ψ (snd yx)) (HomD.ψ (fst yx, G (snd yx)) x))))
          by simp
        also have ... = inD (φ (fst yx)) (ψ (snd yx) (HomD.ψ (fst yx, G (snd yx)) x)))
        proof −
          have «ψ (snd yx) (HomD.ψ (fst yx, G (snd yx)) x) : F (fst yx) → snd yx»
            using x yx HomD.ψ-mapsto [of fst yx G (snd yx)] ψ-in-hom by auto
          thus ?thesis by simp
        qed
        also have ... = inD (HomD.ψ (fst yx, G (snd yx)) x)
          using x yx HomD.ψ-mapsto [of fst yx G (snd yx)] φ-ψ by force
        also have ... = x using x yx inD-ψ by simp
        finally show x = ((inD o φ (fst yx) o HomC.ψ (F (fst yx), snd yx))
                        o (inC o ψ (snd yx) o HomD.ψ (fst yx, G (snd yx)))) x
          by auto
      qed
    qed
    thus ?thesis by auto
  qed
  also have ... = S.mkIde (HomD.set (fst yx, G (snd yx)))
    using yx S.mkIde-as-mkArr HomD.set-subset-Univ by force
  finally have
      S.comp (Φ.map yx) (Ψ o yx) = S.mkIde (HomD.set (fst yx, G (snd yx)))
    by auto
  thus ?thesis using yx HomD.set-subset-Univ S.ide-mkIde by simp
qed
ultimately show ?thesis by auto
qed

interpretation Φ: natural-isomorphism DopxC.comp S.comp
                                Hom-FopxC.map Hom-DopxG.map Φ.map
```

using Φ-*inv* **by** *unfold-locales blast*

interpretation Ψ: *inverse-transformation DopxC.comp S.comp*
Hom-FopxC.map Hom-DopxG.map Φ.*map* **..**

interpretation ΦΨ: *inverse-transformations DopxC.comp S.comp*
Hom-FopxC.map Hom-DopxG.map Φ.*map* Ψ.*map*
using Ψ.*inverts-components* **by** *unfold-locales simp*

abbreviation Φ **where** Φ ≡ Φ.*map*
abbreviation Ψ **where** Ψ ≡ Ψ.*map*

abbreviation *HomC* **where** *HomC* ≡ *HomC.map*
abbreviation φ*C* **where** φ*C* ≡ λ-. *inC*
abbreviation *HomD* **where** *HomD* ≡ *HomD.map*
abbreviation φ*D* **where** φ*D* ≡ λ-. *inD*

theorem *induces-hom-adjunction*: *hom-adjunction C D S.comp S.setp* φ*C* φ*D F G* Φ Ψ **..**

lemma Ψ-*simp*:
assumes *yx*: *DopxC.ide yx*
shows Ψ *yx* = *S.mkArr* (*HomD.set* (*fst yx*, *G* (*snd yx*))) (*HomC.set* (*F* (*fst yx*), *snd yx*))
(*inC o* ψ (*snd yx*) *o HomD.*ψ (*fst yx*, *G* (*snd yx*)))
using *assms* Φ*o-def* Φ-*inv S.inverse-unique* **by** *simp*

The original φ and ψ can be recovered from Φ and Ψ.

interpretation Φ: *set-valued-transformation DopxC.comp S.comp S.setp*
Hom-FopxC.map Hom-DopxG.map Φ.*map* **..**

interpretation Ψ: *set-valued-transformation DopxC.comp S.comp S.setp*
Hom-DopxG.map Hom-FopxC.map Ψ.*map* **..**

lemma φ-*in-terms-of*-Φ′:
assumes *y*: *D.ide y* **and** *f*: «*f*: *F y* →*C x*»
shows φ *y f* = (*HomD.*ψ (*y*, *G x*) *o* Φ.*FUN* (*y*, *x*) *o inC*) *f*
proof −
have *x*: *C.ide x* **using** *f* **by** *auto*
have (*HomD.*ψ (*y*, *G x*) *o* Φ.*FUN* (*y*, *x*) *o inC*) *f* =
*HomD.*ψ (*y*, *G x*)
(*restrict* (*inD o* φ *y o HomC.*ψ (*F y*, *x*)) (*HomC.set* (*F y*, *x*)) (*inC f*))
proof −
have *S.arr* (Φ (*y*, *x*)) **using** *x y* **by** *fastforce*
thus *?thesis*
using *x y* Φ*o-def* **by** *simp*
qed
also have ... = φ *y f*
using *x y f HomC.*φ-*mapsto* φ-*in-hom HomC.*ψ-*mapsto C.ide-in-hom D.ide-in-hom*
by *auto*
finally show *?thesis* **by** *auto*

qed

lemma *ψ-in-terms-of-Ψ′*:
assumes *x*: *C.ide x* **and** *g*: «*g* : *y* →$_D$ *G x*»
shows *ψ x g* = (*HomC.ψ* (*F y*, *x*) *o* Ψ.*FUN* (*y*, *x*) *o inD*) *g*
proof −
have *y*: *D.ide y* **using** *g* **by** *auto*
have (*HomC.ψ* (*F y*, *x*) *o* Ψ.*FUN* (*y*, *x*) *o inD*) *g* =
HomC.ψ (*F y*, *x*)
(*restrict* (*inC o ψ x o HomD.ψ* (*y*, *G x*)) (*HomD.set* (*y*, *G x*)) (*inD g*))
proof −
have *S.arr* (Ψ (*y*, *x*))
using *x y* Ψ.*preserves-reflects-arr* [*of* (*y*, *x*)] **by** *simp*
thus *?thesis*
using *x y* Ψ-*simp* **by** *simp*
qed
also have ... = *ψ x g*
using *x y g HomD.φ-mapsto ψ-in-hom HomD.ψ-mapsto C.ide-in-hom D.ide-in-hom*
by *auto*
finally show *?thesis* **by** *auto*
qed

end

16.9 Hom-Adjunctions Induce Meta-Adjunctions

context *hom-adjunction*
begin

definition *φ* :: *′d* ⇒ *′c* ⇒ *′d*
where
φ y h = (*HomD.ψ* (*y*, *G* (*C.cod h*)) *o* Φ.*FUN* (*y*, *C.cod h*) *o φC* (*F y*, *C.cod h*)) *h*

definition *ψ* :: *′c* ⇒ *′d* ⇒ *′c*
where
ψ x h = (*HomC.ψ* (*F* (*D.dom h*), *x*) *o* Ψ.*FUN* (*D.dom h*, *x*) *o φD* (*D.dom h*, *G x*)) *h*

lemma *Hom-FopxC-map-simp*:
assumes *DopxC.arr gf*
shows *Hom-FopxC.map gf* =
S.mkArr (*HomC.set* (*F* (*D.cod* (*fst gf*)), *C.dom* (*snd gf*)))
(*HomC.set* (*F* (*D.dom* (*fst gf*)), *C.cod* (*snd gf*)))
(*φC* (*F* (*D.dom* (*fst gf*)), *C.cod* (*snd gf*))
o (λ*h*. *snd gf* ·$_C$ *h* ·$_C$ *F* (*fst gf*))
o HomC.ψ (*F* (*D.cod* (*fst gf*)), *C.dom* (*snd gf*)))
using *assms HomC.map-def* **by** *simp*

lemma *Hom-DopxG-map-simp*:
assumes *DopxC.arr gf*

shows *Hom-DopxG.map gf* =
  S.mkArr (*HomD.set* (*D.cod* (*fst gf*), *G* (*C.dom* (*snd gf*))))
    (*HomD.set* (*D.dom* (*fst gf*), *G* (*C.cod* (*snd gf*))))
    (*φD* (*D.dom* (*fst gf*), *G* (*C.cod* (*snd gf*)))
      o (λ*h*. *G* (*snd gf*) ·$_D$ *h* ·$_D$ *fst gf*)
      o HomD.ψ (*D.cod* (*fst gf*), *G* (*C.dom* (*snd gf*))))
 using *assms HomD.map-def* **by** *simp*

lemma Φ-*Fun-mapsto*:
assumes *D.ide y* **and** «*f* : *F y* →$_C$ *x*»
shows Φ.*FUN* (*y*, *x*) ∈ *HomC.set* (*F y*, *x*) → *HomD.set* (*y*, *G x*)
proof −
 have *S.arr* (Φ (*y*, *x*)) ∧ Φ.*DOM* (*y*, *x*) = *HomC.set* (*F y*, *x*) ∧
                    Φ.*COD* (*y*, *x*) = *HomD.set* (*y*, *G x*)
   using *assms HomC.set-map HomD.set-map* **by** *auto*
 thus *?thesis* **using** *S.Fun-mapsto* **by** *blast*
qed

lemma *φ-mapsto*:
assumes *y*: *D.ide y*
shows *φ y* ∈ *C.hom* (*F y*) *x* → *D.hom y* (*G x*)
proof
 fix *h*
 assume *h*: *h* ∈ *C.hom* (*F y*) *x*
 hence *1*: «*h* : *F y* →$_C$ *x*» **by** *simp*
 show *φ y h* ∈ *D.hom y* (*G x*)
 proof −
   have *φC* (*F y*, *x*) *h* ∈ *HomC.set* (*F y*, *x*)
     using *y h 1 HomC.φ-mapsto* [*of F y x*] **by** *fastforce*
   hence Φ.*FUN* (*y*, *x*) (*φC* (*F y*, *x*) *h*) ∈ *HomD.set* (*y*, *G x*)
     using *h y* Φ-*Fun-mapsto* **by** *auto*
   thus *?thesis*
     using *y h 1 φ-def HomC.φ-mapsto HomD.ψ-mapsto* [*of y G x*] **by** *fastforce*
 qed
qed

lemma Φ-*simp*:
assumes *D.ide y* **and** *C.ide x*
shows *S.arr* (Φ (*y*, *x*))
and Φ (*y*, *x*) = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*))
                  (*φD* (*y*, *G x*) *o φ y o ψC* (*F y*, *x*))
proof −
 show *1*: *S.arr* (Φ (*y*, *x*)) **using** *assms* **by** *auto*
 hence Φ (*y*, *x*) = *S.mkArr* (Φ.*DOM* (*y*, *x*)) (Φ.*COD* (*y*, *x*)) (Φ.*FUN* (*y*, *x*))
   using *S.mkArr-Fun* **by** *metis*
 also have ... = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*)) (Φ.*FUN* (*y*, *x*))
   using *assms HomC.set-map HomD.set-map* **by** *fastforce*
 also have ... = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*))
                  (*φD* (*y*, *G x*) *o φ y o ψC* (*F y*, *x*))

```
  proof (intro S.mkArr-eqI')
    show 2: S.arr (S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x)) (Φ.FUN (y, x)))
      using 1 calculation by argo
    show ⋀h. h ∈ HomC.set (F y, x) ⟹
              Φ.FUN (y, x) h = (φD (y, G x) o φ y o ψC (F y, x)) h
    proof −
      fix h
      assume h: h ∈ HomC.set (F y, x)
      have (φD (y, G x) o φ y o HomC.ψ (F y, x)) h =
            φD (y, G x) (ψD (y, G x) (Φ.FUN (y, x) (φC (F y, x) (ψC (F y, x) h))))
      proof −
        have «ψC (F y, x) h : F y →C x»
          using assms h HomC.ψ-mapsto [of F y x] by auto
        thus ?thesis
          using h φ-def by auto
      qed
      also have ... = φD (y, G x) (ψD (y, G x) (Φ.FUN (y, x) h))
        using assms h HomC.φ-ψ Φ-Fun-mapsto by simp
      also have ... = Φ.FUN (y, x) h
        using assms h Φ-Fun-mapsto [of y ψC (F y, x) h] HomC.ψ-mapsto
              HomD.φ-ψ [of y G x] C.ide-in-hom D.ide-in-hom
        by blast
      finally show Φ.FUN (y, x) h = (φD (y, G x) o φ y o ψC (F y, x)) h by auto
    qed
  qed
  finally show Φ (y, x) = S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                                  (φD (y, G x) o φ y o ψC (F y, x))
    by force
qed

lemma Ψ-Fun-mapsto:
assumes C.ide x and «g : y →D G x»
shows Ψ.FUN (y, x) ∈ HomD.set (y, G x) → HomC.set (F y, x)
proof −
  have S.arr (Ψ (y, x)) ∧ Ψ.COD (y, x) = HomC.set (F y, x) ∧
                          Ψ.DOM (y, x) = HomD.set (y, G x)
    using assms HomC.set-map HomD.set-map by auto
  thus ?thesis using S.Fun-mapsto by fast
qed

lemma ψ-mapsto:
assumes x: C.ide x
shows ψ x ∈ D.hom y (G x) → C.hom (F y) x
proof
  fix h
  assume h: h ∈ D.hom y (G x)
  hence 1: «h : y →D G x» by auto
  show ψ x h ∈ C.hom (F y) x
  proof −
```

```
    have Ψ.FUN (y, x) (φD (y, G x) h) ∈ HomC.set (F y, x)
    proof −
      have φD (y, G x) h ∈ HomD.set (y, G x)
        using x h 1 HomD.φ-mapsto [of y G x] by fastforce
      thus ?thesis
        using h x Ψ-Fun-mapsto by auto
    qed
    thus ?thesis
      using x h 1 ψ-def HomD.φ-mapsto HomC.ψ-mapsto [of F y x] by fastforce
  qed
qed

lemma Ψ-simp:
assumes D.ide y and C.ide x
shows S.arr (Ψ (y, x))
and Ψ (y, x) = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x))
                       (φC (F y, x) o ψ x o ψD (y, G x))
proof −
  show 1: S.arr (Ψ (y, x)) using assms by auto
  hence Ψ (y, x) = S.mkArr (Ψ.DOM (y, x)) (Ψ.COD (y, x)) (Ψ.FUN (y, x))
    using S.mkArr-Fun by metis
  also have ... = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x)) (Ψ.FUN (y, x))
    using assms HomC.set-map HomD.set-map by auto
  also have ... = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x))
                          (φC (F y, x) o ψ x o ψD (y, G x))
  proof (intro S.mkArr-eqI′)
    show S.arr (S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x)) (Ψ.FUN (y, x)))
      using 1 calculation by argo
    show ⋀h. h ∈ HomD.set (y, G x) ⟹
             Ψ.FUN (y, x) h = (φC (F y, x) o ψ x o ψD (y, G x)) h
    proof −
      fix h
      assume h: h ∈ HomD.set (y, G x)
      have (φC (F y, x) o ψ x o HomD.ψ (y, G x)) h =
           φC (F y, x) (ψC (F y, x) (Ψ.FUN (y, x) (φD (y, G x) (ψD (y, G x) h))))
      proof −
        have «ψD (y, G x) h : y →D G x»
          using assms h HomD.ψ-mapsto [of y G x] by auto
        thus ?thesis
          using h ψ-def by auto
      qed
      also have ... = φC (F y, x) (ψC (F y, x) (Ψ.FUN (y, x) h))
        using assms h HomD.φ-ψ Ψ-Fun-mapsto by simp
      also have ... = Ψ.FUN (y, x) h
        using assms h Ψ-Fun-mapsto HomD.ψ-mapsto [of y G x] HomC.φ-ψ [of F y x]
              C.ide-in-hom D.ide-in-hom
        by blast
      finally show Ψ.FUN (y, x) h = (φC (F y, x) o ψ x o HomD.ψ (y, G x)) h by auto
    qed
```

```
  qed
  finally show Ψ (y, x) = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x))
                                  (φC (F y, x) o ψ x o ψD (y, G x))
    by force
qed
```

The length of the next proof stems from having to use properties of composition of arrows in S to infer properties of the composition of the corresponding functions.

```
interpretation φψ: meta-adjunction C D F G φ ψ
proof
  fix y :: 'd and x :: 'c and h :: 'c
  assume y: D.ide y and h: «h : F y →C x»
  have x: C.ide x using h by auto
  show «φ y h : y →D G x»
  proof −
    have Φ.FUN (y, x) ∈ HomC.set (F y, x) → HomD.set (y, G x)
      using y h Φ-Fun-mapsto by blast
    thus ?thesis
      using x y h φ-def HomD.ψ-mapsto [of y G x] HomC.φ-mapsto [of F y x] by auto
  qed
  show ψ x (φ y h) = h
  proof −
    have 0: restrict (λh. h) (HomC.set (F y, x))
              = restrict (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) (HomC.set (F y, x))
    proof −
      have 1: S.ide (Ψ (y, x) ·S Φ (y, x))
        using x y ΦΨ.inv [of (y, x)] by auto
      hence 6: S.seq (Ψ (y, x)) (Φ (y, x)) by auto
      have 2: Φ (y, x) = S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                                 (φD (y, G x) o φ y o ψC (F y, x)) ∧
             Ψ (y, x) = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x))
                                 (φC (F y, x) o ψ x o ψD (y, G x))
        using x y Φ-simp Ψ-simp by force
      have 3: S (Ψ (y, x)) (Φ (y, x))
                = S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                          (φC (F y, x) o (ψ x o φ y) o ψC (F y, x))
      proof −
        have 4: S.arr (Ψ (y, x) ·S Φ (y, x)) using 1 by auto
        hence S (Ψ (y, x)) (Φ (y, x))
                = S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                          ((φC (F y, x) o ψ x o ψD (y, G x))
                            o (φD (y, G x) o φ y o ψC (F y, x)))
          using 1 2 S.ide-in-hom S.comp-mkArr by fastforce
        also have ... = S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                                (φC (F y, x) o (ψ x o φ y) o ψC (F y, x))
        proof (intro S.mkArr-eqI')
          show S.arr (S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                              ((φC (F y, x) o ψ x o ψD (y, G x))
                                o (φD (y, G x) o φ y o ψC (F y, x))))
```

```
      using 4 calculation by simp
    show ⋀h. h ∈ HomC.set (F y, x) ⟹
              ((φC (F y, x) o ψ x o ψD (y, G x))
                o (φD (y, G x) o φ y o ψC (F y, x))) h =
              (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) h
    proof −
      fix h
      assume h: h ∈ HomC.set (F y, x)
      hence «φ y (ψC (F y, x) h) : y →D G x»
        using x y h HomC.ψ-mapsto [of F y x] φ-mapsto by auto
      thus ((φC (F y, x) o ψ x o ψD (y, G x))
                o (φD (y, G x) o φ y o ψC (F y, x))) h =
           (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) h
        using x y 1 φ-mapsto HomD.ψ-φ by simp
    qed
  qed
  finally show ?thesis by simp
qed
moreover have Ψ (y, x) ·S Φ (y, x)
                = S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x)) (λh. h)
  using 1 2 6 calculation S.mkIde-as-mkArr S.arr-mkArr S.dom-mkArr S.ideD(2)
  by metis
ultimately have 4: S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                       (φC (F y, x) o (ψ x o φ y) o ψC (F y, x))
                 = S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x)) (λh. h)
  by auto
have 5: S.arr (S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
                  (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)))
  using 1 3 6 by presburger
hence restrict (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) (HomC.set (F y, x))
      = S.Fun (S.mkArr (HomC.set (F y, x)) (HomC.set (F y, x))
          (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)))
  by auto
also have ... = restrict (λh. h) (HomC.set (F y, x))
  using 4 5 by auto
finally show ?thesis by auto
qed
moreover have φC (F y, x) h ∈ HomC.set (F y, x)
  using x y h HomC.φ-mapsto [of F y x] by auto
ultimately have
    φC (F y, x) h = (φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) (φC (F y, x) h)
  using x y h HomC.φ-mapsto [of F y x] by fast
hence ψC (F y, x) (φC (F y, x) h) =
      ψC (F y, x) ((φC (F y, x) o (ψ x o φ y) o ψC (F y, x)) (φC (F y, x) h))
  by simp
hence h = ψC (F y, x) (φC (F y, x) (ψ x (φ y (ψC (F y, x) (φC (F y, x) h)))))
  using x y h HomC.ψ-φ [of F y x] by simp
also have ... = ψ x (φ y h)
  using x y h HomC.ψ-φ HomC.ψ-φ φ-mapsto ψ-mapsto
```

```
    by (metis PiE mem-Collect-eq)
  finally show ?thesis by auto
qed
next
fix x :: 'c and h :: 'd and y :: 'd
assume x: C.ide x and h: «h : y →D G x»
have y: D.ide y using h by auto
show «ψ x h : F y →C x» using x y h ψ-mapsto [of x y] by auto
show φ y (ψ x h) = h
proof −
  have 0: restrict (λh. h) (HomD.set (y, G x))
            = restrict (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) (HomD.set (y, G x))
  proof −
    have 1: S.ide (S (Φ (y, x)) (Ψ (y, x)))
      using x y ΦΨ.inv by force
    hence 6: S.seq (Φ (y, x)) (Ψ (y, x)) by auto
    have 2: Φ (y, x) = S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                         (φD (y, G x) o φ y o ψC (F y, x)) ∧
            Ψ (y, x) = S.mkArr (HomD.set (y, G x)) (HomC.set (F y, x))
                         (φC (F y, x) o ψ x o ψD (y, G x))
      using x h Φ-simp Ψ-simp by auto
    have 3: S (Φ (y, x)) (Ψ (y, x))
              = S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                  (φD (y, G x) o (φ y o ψ x) o ψD (y, G x))
    proof −
      have 4: S.seq (Φ (y, x)) (Ψ (y, x)) using 1 by auto
      hence S (Φ (y, x)) (Ψ (y, x))
              = S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                  ((φD (y, G x) o φ y o ψC (F y, x))
                    o (φC (F y, x) o ψ x o ψD (y, G x)))
        using 1 2 6 S.ide-in-hom S.comp-mkArr by fastforce
      also have ... = S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                        (φD (y, G x) o (φ y o ψ x) o ψD (y, G x))
      proof
        show S.arr (S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                      ((φD (y, G x) o φ y o ψC (F y, x))
                        o (φC (F y, x) o ψ x o ψD (y, G x))))
          using 4 calculation by simp
        show ⋀h. h ∈ HomD.set (y, G x) ⟹
                   ((φD (y, G x) o φ y o ψC (F y, x))
                     o (φC (F y, x) o ψ x o ψD (y, G x))) h =
                   (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) h
        proof −
          fix h
          assume h: h ∈ HomD.set (y, G x)
          hence «ψ x (ψD (y, G x) h) : F y →C x»
            using x y HomD.ψ-mapsto [of y G x] ψ-mapsto by auto
          thus ((φD (y, G x) o φ y o ψC (F y, x))
                  o (φC (F y, x) o ψ x o ψD (y, G x))) h =
```

```
            (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) h
          using x y HomC.ψ-φ by simp
        qed
      qed
      finally show ?thesis by auto
    qed
    moreover have Φ (y, x) ·S Ψ (y, x) =
                  S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x)) (λh. h)
      using 1 2 6 calculation
      by (metis S.arr-mkArr S.cod-mkArr S.ide-in-hom S.mkIde-as-mkArr S.in-homE)
    ultimately have 4: S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                                 (φD (y, G x) o (φ y o ψ x) o ψD (y, G x))
                       = S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x)) (λh. h)
      by auto
    have 5: S.arr (S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                           (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)))
      using 1 3 by fastforce
    hence restrict (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) (HomD.set (y, G x))
          = S.Fun (S.mkArr (HomD.set (y, G x)) (HomD.set (y, G x))
                    (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)))
      by auto
    also have ... = restrict (λh. h) (HomD.set (y, G x))
      using 4 5 by auto
    finally show ?thesis by auto
  qed
  moreover have φD (y, G x) h ∈ HomD.set (y, G x)
    using x y h HomD.φ-mapsto [of y G x] by auto
  ultimately have
      φD (y, G x) h = (φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) (φD (y, G x) h)
    by fast
  hence ψD (y, G x) (φD (y, G x) h) =
        ψD (y, G x) ((φD (y, G x) o (φ y o ψ x) o ψD (y, G x)) (φD (y, G x) h))
    by simp
  hence h = ψD (y, G x) (φD (y, G x) (φ y (ψ x (ψD (y, G x) (φD (y, G x) h)))))
    using x y h HomD.ψ-φ by simp
  also have ... = φ y (ψ x h)
    using x y h HomD.ψ-φ HomD.ψ-φ [of φ y (ψ x h) y G x] φ-mapsto ψ-mapsto
    by fastforce
  finally show ?thesis by auto
qed
next
fix x :: 'c and x' :: 'c and y :: 'd and y' :: 'd
and f :: 'c and g :: 'd and h :: 'c
assume f: «f : x →C x'» and g: «g : y' →D y» and h: «h : F y →C x»
have x: C.ide x using f by auto
have y: D.ide y using g by auto
have x': C.ide x' using f by auto
have y': D.ide y' using g by auto
show φ y' (f ·C h ·C F g) = G f ·D φ y h ·D g
```

```
proof −
  have 0: restrict ((φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x))
                    o (φD (y, G x) o φ y o ψC (F y, x)))
                   (HomC.set (F y, x))
        = restrict ((φD (y', G x') o φ y' o ψC (F y', x'))
                    o (φC (F y', x') o (λh. f ·_C h ·_C F g)) o ψC (F y, x))
                   (HomC.set (F y, x))
  proof −
    have 1: S.arr (Φ (y, x)) ∧
          Φ (y, x) = S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                             (φD (y, G x) o φ y o ψC (F y, x))
      using x y Φ-simp [of y x] by auto
    have 2: S.arr (Φ (y', x')) ∧
          Φ (y', x') = S.mkArr (HomC.set (F y', x')) (HomD.set (y', G x'))
                               (φD (y', G x') o φ y' o ψC (F y', x'))
      using x' y' Φ-simp [of y' x'] by auto
    have 3: S.arr (S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                           ((φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x))
                            o (φD (y, G x) o φ y o ψC (F y, x))))
          ∧ S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                    ((φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x))
                     o (φD (y, G x) o φ y o ψC (F y, x)))
            = S (S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
                         (φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x)))
                (S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                         (φD (y, G x) o φ y o ψC (F y, x)))
    proof −
      have 1: S.seq (S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
                             (φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x)))
                    (S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                             (φD (y, G x) o φ y o ψC (F y, x)))
      proof −
        have S.arr (Hom-DopxG.map (g, f)) ∧
             Hom-DopxG.map (g, f)
               = S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
                         (φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x))
          using f g Hom-DopxG.preserves-arr Hom-DopxG-map-simp by fastforce
        thus ?thesis
          using 1 S.cod-mkArr S.dom-mkArr S.seqI by metis
      qed
      have S.seq (S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
                          (φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x)))
                 (S.mkArr (HomC.set (F y, x)) (HomD.set (y, G x))
                          (φD (y, G x) o φ y o ψC (F y, x)))
        using 1 by (intro S.seqI', auto)
      moreover have S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                            ((φD (y', G x') o (λh. G f ·_D h ·_D g) o ψD (y, G x))
                             o (φD (y, G x) o φ y o ψC (F y, x)))
                    = S (S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
```

$(\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f\ \cdot_D\ h\ \cdot_D\ g)\ o\ \psi D\ (y,\ G\ x)))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
using *1 S.comp-mkArr* **by** *fastforce*
ultimately show *?thesis* **by** *auto*
qed
moreover have
4: $S.arr\ (S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
$o\ (\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x))))$
$\wedge\ S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
$o\ (\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
$= S\ (S.mkArr\ (HomC.set\ (F\ y',\ x'))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x')))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y',\ x'))$
$(\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
proof −
have *5*: $S.seq\ (S.mkArr\ (HomC.set\ (F\ y',\ x'))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x')))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y',\ x'))$
$(\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
proof −
have $S.arr\ (Hom\text{-}FopxC.map\ (g,\ f))\ \wedge$
$Hom\text{-}FopxC.map\ (g,\ f)$
$= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y',\ x'))$
$(\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x))$
using *f g Hom-FopxC.preserves-arr Hom-FopxC-map-simp* **by** *fastforce*
thus *?thesis* **using** *2 S.cod-mkArr S.dom-mkArr S.seqI* **by** *metis*
qed
have $S.seq\ (S.mkArr\ (HomC.set\ (F\ y',\ x'))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x')))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y',\ x'))$
$(\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
using *5* **by** (*intro S.seqI'*, *auto*)
moreover have $S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
$o\ (\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
$= S\ (S.mkArr\ (HomC.set\ (F\ y',\ x'))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x')))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y',\ x'))$
$(\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
using *5 S.comp-mkArr* **by** *fastforce*
ultimately show *?thesis* **by** *argo*
qed
moreover have *2*:
$S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f\ \cdot_D\ h\ \cdot_D\ g)\ o\ \psi D\ (y,\ G\ x))$
$o\ (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$

```
        = S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                ((φD (y', G x') o φ y' o ψC (F y', x'))
                  o (φC (F y', x') o (λh. f ·C h ·C F g) o ψC (F y, x)))
  proof −
    have
      S (Hom-DopxG.map (g, f)) (Φ (y, x)) = S (Φ (y', x')) (Hom-FopxC.map (g, f))
      using f g Φ.naturality1 Φ.naturality2 by fastforce
    moreover have Hom-DopxG.map (g, f)
                  = S.mkArr (HomD.set (y, G x)) (HomD.set (y', G x'))
                        (φD (y', G x') o (λh. G f ·D h ·D g) o ψD (y, G x))
      using f g Hom-DopxG-map-simp [of (g, f)] by fastforce
    moreover have Hom-FopxC.map (g, f)
                  = S.mkArr (HomC.set (F y, x)) (HomC.set (F y', x'))
                        (φC (F y', x') o (λh. f ·C h ·C F g) o ψC (F y, x))
      using f g Hom-FopxC-map-simp [of (g, f)] by fastforce
    ultimately show ?thesis using 1 2 3 4 by simp
  qed
  ultimately have 6: S.arr (S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                                ((φD (y', G x') o (λh. G f ·D h ·D g) o ψD (y, G x))
                                  o (φD (y, G x) o φ y o ψC (F y, x))))
    by fast
  hence restrict ((φD (y', G x') o (λh. D (G f) (D h g)) o ψD (y, G x))
                    o (φD (y, G x) o φ y o ψC (F y, x)))
                  (HomC.set (F y, x))
        = S.Fun (S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                      ((φD (y', G x') o (λh. G f ·D h ·D g) o ψD (y, G x))
                        o (φD (y, G x) o φ y o ψC (F y, x))))
    by simp
  also have ... = S.Fun (S.mkArr (HomC.set (F y, x)) (HomD.set (y', G x'))
                            ((φD (y', G x') o φ y' o ψC (F y', x'))
                              o (φC (F y', x') o (λh. f ·C h ·C F g) o ψC (F y, x))))
    using 2 by argo
  also have ... = restrict ((φD (y', G x') o φ y' o ψC (F y', x'))
                              o (φC (F y', x') o (λh. f ·C h ·C F g) o ψC (F y, x)))
                            (HomC.set (F y, x))
    using 4 S.Fun-mkArr by meson
  finally show ?thesis by auto
qed
hence 5: ((φD (y', G x') ∘ (λh. G f ·D h ·D g) ∘ ψD (y, G x))
            ∘ (φD (y, G x) ∘ φ y ∘ ψC (F y, x))) (φC (F y, x) h) =
          (φD (y', G x') ∘ φ y' ∘ ψC (F y', x')
            ∘ (φC (F y', x') ∘ (λh. f ·C h ·C F g)) ∘ ψC (F y, x)) (φC (F y, x) h)
proof −
  have φC (F y, x) h ∈ HomC.set (F y, x)
    using x y h HomC.φ-mapsto [of F y x] by auto
  thus ?thesis
    using 0 h restr-eqE [of (φD (y', G x') ∘ (λh. G f ·D h ·D g) ∘ ψD (y, G x))
                            ∘ (φD (y, G x) ∘ φ y ∘ ψC (F y, x))
                          HomC.set (F y, x)
```

```
                    (φD (y', G x') ∘ φ y' ∘ ψC (F y', x'))
                      ∘ (φC (F y', x') ∘ (λh. f ·C h ·C F g) o ψC (F y, x))]
    by fast
qed
show ?thesis
proof −
  have φ y' (C f (C h (F g))) =
        ψD (y', G x') (φD (y', G x') (φ y' (ψC (F y', x') (φC (F y', x')
          (C f (C (ψC (F y, x) (φC (F y, x) h)) (F g)))))))
  proof −
    have ψD (y', G x') (φD (y', G x') (φ y' (ψC (F y', x') (φC (F y', x')
           (C f (C (ψC (F y, x) (φC (F y, x) h)) (F g)))))))
         = ψD (y', G x') (φD (y', G x') (φ y' (ψC (F y', x') (φC (F y', x')
             (C f (C h (F g)))))))
      using x y h HomC.ψ-φ by simp
    also have ... = ψD (y', G x') (φD (y', G x') (φ y' (C f (C h (F g)))))
      using f g h HomC.ψ-φ [of C f (C h (F g))] by fastforce
    also have ... = φ y' (C f (C h (F g)))
    proof −
      have «φ y' (f ·C h ·C F g) : y' →D G x'»
        using f g h y' x' φ-mapsto [of y' x'] by auto
      thus ?thesis by simp
    qed
    finally show ?thesis by auto
  qed
  also have
    ... = ψD (y', G x')
            (φD (y', G x')
               (G f ·D ψD (y, G x) (φD (y, G x) (φ y (ψC (F y, x) (φC (F y, x) h))))
                  ·D g))
    using 5 by force
  also have ... = D (G f) (D (φ y h) g)
  proof −
    have φyh: «φ y h : y →D G x»
      using x y h φ-mapsto by auto
    have ψD (y', G x')
           (φD (y', G x')
              (G f ·D ψD (y, G x) (φD (y, G x) (φ y (ψC (F y, x) (φC (F y, x) h))))
                 ·D g)) =
         ψD (y', G x') (φD (y', G x') (G f ·D ψD (y, G x) (φD (y, G x) (φ y h)) ·D g))
      using x y f g h by auto
    also have ... = ψD (y', G x') (φD (y', G x') (G f ·D φ y h ·D g))
      using φyh x' y' f g by simp
    also have ... = G f ·D φ y h ·D g
      using φyh f g by fastforce
    finally show ?thesis by auto
  qed
  finally show ?thesis by auto
qed
```

qed
qed

theorem *induces-meta-adjunction*:
shows *meta-adjunction C D F G φ ψ* **..**

end

16.10 Putting it All Together

Combining the above results, an interpretation of any one of the locales: *left-adjoint-functor*, *right-adjoint-functor*, *meta-adjunction*, *hom-adjunction*, and *unit-counit-adjunction* extends to an interpretation of *adjunction*.

context *meta-adjunction*
begin

interpretation *S*: *replete-setcat* **.**
interpretation *F*: *left-adjoint-functor D C F* **using** *has-left-adjoint-functor* **by** *auto*
interpretation *G*: *right-adjoint-functor C D G* **using** *has-right-adjoint-functor* **by** *auto*

interpretation *ηε*: *unit-counit-adjunction C D F G η ε*
using *induces-unit-counit-adjunction η-def ε-def* **by** *auto*
interpretation ΦΨ: *hom-adjunction C D S.comp S.setp φC φD F G* Φ Ψ
using *induces-hom-adjunction* **by** *auto*

theorem *induces-adjunction*:
shows *adjunction C D S.comp S.setp φC φD F G φ ψ η ε* Φ Ψ
using *ε-map-simp η-map-simp φ-in-terms-of-η φ-in-terms-of-*Φ′ *ψ-in-terms-of-ε*
*ψ-in-terms-of-*Ψ′ *Φ-simp Ψ-simp η-def ε-def*
by *unfold-locales auto*

end

context *unit-counit-adjunction*
begin

interpretation *φψ*: *meta-adjunction C D F G φ ψ* **using** *induces-meta-adjunction* **by** *auto*

interpretation *S*: *replete-setcat* **.**
interpretation *F*: *left-adjoint-functor D C F* **using** *φψ.has-left-adjoint-functor* **by** *auto*
interpretation *G*: *right-adjoint-functor C D G* **using** *φψ.has-right-adjoint-functor* **by** *auto*

interpretation ΦΨ: *hom-adjunction C D S.comp S.setp*
*φψ.φC φψ.φD F G φψ.*Φ *φψ.*Ψ
using *φψ.induces-hom-adjunction* **by** *auto*

theorem *induces-adjunction*:
shows *adjunction C D S.comp S.setp φψ.φC φψ.φD F G φ ψ η ε φψ.*Φ *φψ.*Ψ

using *ε-in-terms-of-ψ η-in-terms-of-φ φψ.φ-in-terms-of-Φ′ ψ-def φψ.ψ-in-terms-of-Ψ′ φψ.Φ-simp φψ.Ψ-simp φ-def*
by *unfold-locales auto*

end

context *hom-adjunction*
begin

interpretation *φψ*: *meta-adjunction C D F G φ ψ*
using *induces-meta-adjunction* **by** *auto*
interpretation *F*: *left-adjoint-functor D C F* **using** *φψ.has-left-adjoint-functor* **by** *auto*
interpretation *G*: *right-adjoint-functor C D G* **using** *φψ.has-right-adjoint-functor* **by** *auto*
interpretation *ηε*: *unit-counit-adjunction C D F G φψ.η φψ.ε*
using *φψ.induces-unit-counit-adjunction φψ.η-def φψ.ε-def* **by** *auto*

theorem *induces-adjunction*:
shows *adjunction C D S setp φC φD F G φ ψ φψ.η φψ.ε Φ Ψ*
proof
fix *x*
assume *C.ide x*
thus *φψ.ε x = ψ x (G x)*
using *φψ.ε-map-simp φψ.ε-def* **by** *simp*
next
fix *y*
assume *D.ide y*
thus *φψ.η y = φ y (F y)*
using *φψ.η-map-simp φψ.η-def* **by** *simp*
fix *x y f*
assume *y*: *D.ide y* **and** *f*: «*f* : *F y* →*C x*»
show *φ y f = G f* ·*D φψ.η y*
using *y f φψ.φ-in-terms-of-η φψ.η-def* **by** *simp*
show *φ y f* = (*ψD* (*y*, *G x*) ∘ *Φ.FUN* (*y*, *x*) ∘ *φC* (*F y*, *x*)) *f*
using *y f φ-def* **by** *auto*
next
fix *x y g*
assume *x*: *C.ide x* **and** *g*: «*g* : *y* →*D G x*»
show *ψ x g = φψ.ε x* ·*C F g*
using *x g φψ.ψ-in-terms-of-ε φψ.ε-def* **by** *simp*
show *ψ x g* = (*ψC* (*F y*, *x*) ∘ *Ψ.FUN* (*y*, *x*) ∘ *φD* (*y*, *G x*)) *g*
using *x g ψ-def* **by** *fast*
next
fix *x y*
assume *x*: *C.ide x* **and** *y*: *D.ide y*
show *Φ* (*y*, *x*) = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*))
(*φD* (*y*, *G x*) *o φ y o ψC* (*F y*, *x*))
using *x y Φ-simp* **by** *simp*
show *Ψ* (*y*, *x*) = *S.mkArr* (*HomD.set* (*y*, *G x*)) (*HomC.set* (*F y*, *x*))
(*φC* (*F y*, *x*) *o ψ x o ψD* (*y*, *G x*))

```
    using x y Ψ-simp by simp
  qed

end

context left-adjoint-functor
begin

  interpretation φψ: meta-adjunction C D F G φ ψ
    using induces-meta-adjunction by auto
  interpretation S: replete-setcat .

  theorem induces-adjunction:
  shows adjunction C D S.comp S.setp φψ.φC φψ.φD F G φ ψ φψ.η φψ.ε φψ.Φ φψ.Ψ
    using φψ.induces-adjunction by auto

end

context right-adjoint-functor
begin

  interpretation φψ: meta-adjunction C D F G φ ψ
    using induces-meta-adjunction by auto
  interpretation S: replete-setcat .

  theorem induces-adjunction:
  shows adjunction C D S.comp S.setp φψ.φC φψ.φD F G φ ψ φψ.η φψ.ε φψ.Φ φψ.Ψ
    using φψ.induces-adjunction by auto

end

definition adjoint-functors
where adjoint-functors C D F G = (∃φ ψ. meta-adjunction C D F G φ ψ)

lemma adjoint-functors-respects-naturally-isomorphic:
assumes adjoint-functors C D F G
and naturally-isomorphic D C F' F and naturally-isomorphic C D G G'
shows adjoint-functors C D F' G'
proof −
  obtain φ ψ where φψ: meta-adjunction C D F G φ ψ
    using assms(1) adjoint-functors-def by blast
  interpret φψ: meta-adjunction C D F G φ ψ
    using φψ by simp
  obtain τ where τ: natural-isomorphism D C F' F τ
    using assms(2) naturally-isomorphic-def by blast
  obtain μ where μ: natural-isomorphism C D G G' μ
    using assms(3) naturally-isomorphic-def by blast
  show ?thesis
    using adjoint-functors-def τ μ φψ.respects-natural-isomorphism by blast
```

qed

lemma *left-adjoint-functor-respects-naturally-isomorphic*:
assumes *left-adjoint-functor D C F*
and *naturally-isomorphic D C F F'*
shows *left-adjoint-functor D C F'*
proof −
  interpret *F*: *left-adjoint-functor D C F*
    using *assms*(*1*) **by** *simp*
  have *1*: *meta-adjunction C D F F.G F.φ F.ψ*
    using *F.induces-meta-adjunction* **by** *simp*
  interpret *φψ*: *meta-adjunction C D F F.G F.φ F.ψ*
    using *1* **by** *simp*
  have *adjoint-functors C D F F.G*
    using *1 adjoint-functors-def* **by** *blast*
  hence *2*: *adjoint-functors C D F' F.G*
    using *assms*(*2*) *adjoint-functors-respects-naturally-isomorphic* [*of C D F F.G F' F.G*]
        naturally-isomorphic-reflexive naturally-isomorphic-symmetric
        φψ.G.functor-axioms
    by *blast*
  obtain *φ' ψ'* **where** *φ'ψ'*: *meta-adjunction C D F' F.G φ' ψ'*
    using *2 adjoint-functors-def* **by** *blast*
  interpret *φ'ψ'*: *meta-adjunction C D F' F.G φ' ψ'*
    using *φ'ψ'* **by** *simp*
  show *?thesis*
    using *φ'ψ'.has-left-adjoint-functor* **by** *simp*
qed

lemma *right-adjoint-functor-respects-naturally-isomorphic*:
assumes *right-adjoint-functor C D G*
and *naturally-isomorphic C D G G'*
shows *right-adjoint-functor C D G'*
proof −
  interpret *G*: *right-adjoint-functor C D G*
    using *assms*(*1*) **by** *simp*
  have *1*: *meta-adjunction C D G.F G G.φ G.ψ*
    using *G.induces-meta-adjunction* **by** *simp*
  interpret *φψ*: *meta-adjunction C D G.F G G.φ G.ψ*
    using *1* **by** *simp*
  have *adjoint-functors C D G.F G*
    using *1 adjoint-functors-def* **by** *blast*
  hence *2*: *adjoint-functors C D G.F G'*
    using *assms*(*2*) *adjoint-functors-respects-naturally-isomorphic*
        naturally-isomorphic-reflexive naturally-isomorphic-symmetric
        φψ.F.functor-axioms
    by *blast*
  obtain *φ' ψ'* **where** *φ'ψ'*: *meta-adjunction C D G.F G' φ' ψ'*
    using *2 adjoint-functors-def* **by** *blast*
  interpret *φ'ψ'*: *meta-adjunction C D G.F G' φ' ψ'*

```
    using φ'ψ' by simp
  show ?thesis
    using φ'ψ'.has-right-adjoint-functor by simp
qed
```

16.11 Inverse Functors are Adjoints

```
lemma inverse-functors-induce-meta-adjunction:
assumes inverse-functors C D F G
shows meta-adjunction C D F G (λx. G) (λy. F)
proof −
  interpret inverse-functors C D F G using assms by auto
  interpret meta-adjunction C D F G ‹λx. G› ‹λy. F›
  proof −
    have 1: ⋀y. B.arr y ⟹ G (F y) = y
      by (metis B.map-simp comp-apply inv)
    have 2: ⋀x. A.arr x ⟹ F (G x) = x
      by (metis A.map-simp comp-apply inv')
    show meta-adjunction C D F G (λx. G) (λy. F)
    proof
      fix y f x
      assume y: B.ide y and f: «f : F y →A x»
      show «G f : y →B G x»
        using y f 1 G.preserves-hom by (elim A.in-homE, auto)
      show F (G f) = f
        using f 2 by auto
      next
      fix x g y
      assume x: A.ide x and g: «g : y →B G x»
      show «F g : F y →A x»
        using x g 2 F.preserves-hom by (elim B.in-homE, auto)
      show G (F g) = g using g 1 A.map-def by blast
      next
      fix f x x' g y' y h
      assume f: «f : x →A x'» and g: «g : y' →B y» and h: «h : F y →A x»
      show G (C f (C h (F g))) = D (G f) (D (G h) g)
        using f g h 1 2 inv inv' A.map-def B.map-def by (elim A.in-homE B.in-homE, auto)
    qed
  qed
  show ?thesis ..
qed

lemma inverse-functors-are-adjoints:
assumes inverse-functors A B F G
shows adjoint-functors A B F G
  using assms inverse-functors-induce-meta-adjunction adjoint-functors-def by fast

context inverse-functors
begin
```

lemma *η-char*:
shows *meta-adjunction.η B F* (λ*x*. *G*) = *identity-functor.map B*
proof (*intro natural-transformation-eqI*)
interpret *meta-adjunction A B F G* ‹λ*y*. *G*› ‹λ*x*. *F*›
using *inverse-functors-induce-meta-adjunction inverse-functors-axioms* **by** *auto*
interpret *S*: *replete-setcat* **.**
interpret *adjunction A B S.comp S.setp* φ*C* φ*D F G* ‹λ*y*. *G*› ‹λ*x*. *F*› η ε Φ Ψ
using *induces-adjunction* **by** *force*
show *natural-transformation B B B.map GF.map* η
using *η.natural-transformation-axioms* **by** *auto*
show *natural-transformation B B B.map GF.map B.map*
by (*simp add*: *B.as-nat-trans.natural-transformation-axioms inv*)
show ⋀*b*. *B.ide b* ⟹ η *b* = *B.map b*
using *η-in-terms-of-*φ *ηo-def ηo-in-hom* **by** *fastforce*
qed

lemma *ε-char*:
shows *meta-adjunction.*ε *A F G* (λ*y*. *F*) = *identity-functor.map A*
proof (*intro natural-transformation-eqI*)
interpret *meta-adjunction A B F G* ‹λ*y*. *G*› ‹λ*x*. *F*›
using *inverse-functors-induce-meta-adjunction inverse-functors-axioms* **by** *auto*
interpret *S*: *replete-setcat* **.**
interpret *adjunction A B S.comp S.setp* φ*C* φ*D F G* ‹λ*y*. *G*› ‹λ*x*. *F*› η ε Φ Ψ
using *induces-adjunction* **by** *force*
show *natural-transformation A A FG.map A.map* ε
using *ε.natural-transformation-axioms* **by** *auto*
show *natural-transformation A A FG.map A.map A.map*
by (*simp add*: *A.as-nat-trans.natural-transformation-axioms inv*′)
show ⋀*a*. *A.ide a* ⟹ ε *a* = *A.map a*
using *ε-in-terms-of-*ψ *εo-def εo-in-hom* **by** *fastforce*
qed

end

16.12 Composition of Adjunctions

locale *composite-adjunction* =
A: *category A* +
B: *category B* +
C: *category C* +
F: *functor B A F* +
G: *functor A B G* +
F′: *functor C B F*′ +
G′: *functor B C G*′ +
FG: *meta-adjunction A B F G* φ ψ +
F′*G*′: *meta-adjunction B C F*′ *G*′ φ′ ψ′
for *A* :: ′*a comp* (**infixr** ‹·$_A$› *55*)
and *B* :: ′*b comp* (**infixr** ‹·$_B$› *55*)

and $C :: 'c\ comp$ (**infixr** $\langle\cdot_C\rangle$ 55)
and $F :: 'b \Rightarrow 'a$
and $G :: 'a \Rightarrow 'b$
and $F' :: 'c \Rightarrow 'b$
and $G' :: 'b \Rightarrow 'c$
and $\varphi :: 'b \Rightarrow 'a \Rightarrow 'b$
and $\psi :: 'a \Rightarrow 'b \Rightarrow 'a$
and $\varphi' :: 'c \Rightarrow 'b \Rightarrow 'c$
and $\psi' :: 'b \Rightarrow 'c \Rightarrow 'b$
begin

interpretation S: *replete-setcat* **.**
interpretation FG: *adjunction A B S.comp S.setp*
FG.φC FG.φD F G φ ψ FG.η FG.ε FG.Φ FG.Ψ
using *FG.induces-adjunction* **by** *simp*
interpretation $F'G'$: *adjunction B C S.comp S.setp F'G'.φC F'G'.φD F' G' φ' ψ'*
F'G'.η F'G'.ε F'G'.Φ F'G'.Ψ
using *F'G'.induces-adjunction* **by** *simp*

lemma *is-meta-adjunction*:
shows *meta-adjunction A C (F o F') (G' o G) (λz. φ' z o φ (F' z)) (λx. ψ x o ψ' (G x))*
proof −
interpret $G'oG$: *composite-functor A B C G G'* **..**
interpret FoF': *composite-functor C B A F' F* **..**
show *?thesis*
proof
fix *y f x*
assume *y*: *C.ide y* **and** *f*: «*f* : *FoF'.map y* →$_A$ *x*»
show «(*φ' y* ∘ *φ* (*F' y*)) *f* : *y* →$_C$ *G'oG.map x*»
using *y f FG.φ-in-hom F'G'.φ-in-hom* **by** *simp*
show (*ψ x* ∘ *ψ'* (*G x*)) ((*φ' y* ∘ *φ* (*F' y*)) *f*) = *f*
using *y f FG.φ-in-hom F'G'.φ-in-hom FG.ψ-φ F'G'.ψ-φ* **by** *simp*
next
fix *x g y*
assume *x*: *A.ide x* **and** *g*: «*g* : *y* →$_C$ *G'oG.map x*»
show «(*ψ x* ∘ *ψ'* (*G x*)) *g* : *FoF'.map y* →$_A$ *x*»
using *x g FG.ψ-in-hom F'G'.ψ-in-hom* **by** *auto*
show (*φ' y* ∘ *φ* (*F' y*)) ((*ψ x* ∘ *ψ'* (*G x*)) *g*) = *g*
using *x g FG.ψ-in-hom F'G'.ψ-in-hom FG.φ-ψ F'G'.φ-ψ* **by** *simp*
next
fix *f x x' g y' y h*
assume *f*: «*f* : *x* →$_A$ *x'*» **and** *g*: «*g* : *y'* →$_C$ *y*» **and** *h*: «*h* : *FoF'.map y* →$_A$ *x*»
show (*φ' y'* ∘ *φ* (*F' y'*)) (*f* ⋅$_A$ *h* ⋅$_A$ *FoF'.map g*) =
G'oG.map f ⋅$_C$ (*φ' y* ∘ *φ* (*F' y*)) *h* ⋅$_C$ *g*
using *f g h FG.φ-naturality* [*of f x x' F' g F' y' F' y h*]
F'G'.φ-naturality [*of G f G x G x' g y' y φ* (*F' y*) *h*]
FG.φ-in-hom

```
      by fastforce
  qed
qed

interpretation KηH: natural-transformation C C ‹G' o F'› ‹G' o G o F o F'›
                ‹G' o FG.η o F'›
proof −
  interpret ηF': natural-transformation C B F' ‹(G o F) o F'› ‹FG.η o F'›
    using FG.η-naturalitytransformation F'.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by fastforce
  interpret G'ηF': natural-transformation C C ‹G' o F'› ‹G' o (G o F o F')›
                ‹G' o (FG.η o F')›
    using ηF'.natural-transformation-axioms G'.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by blast
  show natural-transformation C C (G' o F') (G' o G o F o F') (G' o FG.η o F')
    using G'ηF'.natural-transformation-axioms o-assoc by metis
qed
interpretation G'ηF'oη': vertical-composite C C C.map ‹G' o F'› ‹G' o G o F o F'›
                    F'G'.η ‹G' o FG.η o F'› ..

interpretation FεG: natural-transformation A A ‹F o F' o G' o G› ‹F o G›
                ‹F o F'G'.ε o G›
proof −
  interpret Fε': natural-transformation B A ‹F o (F' o G')› F ‹F o F'G'.ε›
   using F'G'.ε.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by fastforce
   interpret Fε'G: natural-transformation A A ‹F o (F' o G') o G› ‹F o G› ‹F o F'G'.ε o
G›
    using Fε'.natural-transformation-axioms G.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by blast
  show natural-transformation A A (F o F' o G' o G) (F o G) (F o F'G'.ε o G)
    using Fε'G.natural-transformation-axioms o-assoc by metis
qed
interpretation εoFε'G: vertical-composite A A ‹F ∘ F' ∘ G' ∘ G› ‹F o G› A.map
                    ‹F o F'G'.ε o G› FG.ε ..

interpretation meta-adjunction A C ‹F o F'› ‹G' o G›
                            ‹λz. φ' z o φ (F' z)› ‹λx. ψ x o ψ' (G x)›
  using is-meta-adjunction by auto
interpretation S: replete-setcat .
interpretation adjunction A C S.comp S.setp φC φD ‹F ∘ F'› ‹G' ∘ G›
            ‹λz. φ' z ∘ φ (F' z)› ‹λx. ψ x ∘ ψ' (G x)› η ε Φ Ψ
  using induces-adjunction by simp

lemma η-char:
```

shows $\eta = G'\eta F'o\eta'.map$
proof (*intro natural-transformation-eqI*)
show *natural-transformation C C C.map* $(G' \circ G \circ F \circ F')$ $G'\eta F'o\eta'.map$ **..**
show *natural-transformation C C C.map* $(G' \circ G \circ F \circ F')$ η
by (*metis* (*no-types*, *lifting*) *η-naturalitytransformation o-assoc*)
fix *a*
assume *a*: *C.ide a*
show $\eta\ a = G'\eta F'o\eta'.map\ a$
unfolding *η-def*
using *a G'ηF'oη'.map-def FG.η.preserves-hom* [*of F' a F' a F' a*]
F'G'.φ-in-terms-of-η FG.η-map-simp η-map-simp [*of a*] *C.ide-in-hom*
F'G'.η-def FG.η-def
by *auto*
qed

lemma *ε-char*:
shows $\varepsilon = \varepsilon oF\varepsilon' G.map$
proof (*intro natural-transformation-eqI*)
show *natural-transformation A A* $(F \circ F' \circ G' \circ G)$ *A.map* ε
by (*metis* (*no-types*, *lifting*) *ε-naturalitytransformation o-assoc*)
show *natural-transformation A A* $(F \circ F' \circ G' \circ G)$ *A.map* $\varepsilon oF\varepsilon' G.map$ **..**
fix *a*
assume *a*: *A.ide a*
show $\varepsilon\ a = \varepsilon oF\varepsilon' G.map\ a$
proof −
have $\varepsilon\ a = \psi\ a\ (\psi'\ (G\ a)\ (G'\ (G\ a)))$
using *a ε-in-terms-of-ψ* **by** *simp*
also have $... = FG.\varepsilon\ a \cdot_A F\ (F'G'.\varepsilon\ (G\ a) \cdot_B F'\ (G'\ (G\ a)))$
by (*metis F'G'.ε-in-terms-of-ψ F'G'.εo-def F'G'.εo-in-hom F'G'.ηε.ε-in-terms-of-ψ*
F'G'.ηε.ψ-def FG.Gε.natural-transformation-axioms FG.ψ-in-terms-of-ε a
functor.preserves-ide natural-transformation-def)
also have $... = \varepsilon oF\varepsilon' G.map\ a$
using *a B.comp-arr-dom εoFε'G.map-def* **by** *simp*
finally show *?thesis* **by** *blast*
qed
qed

end

16.13 Right Adjoints are Unique up to Natural Isomorphism

As an example of the use of the of the foregoing development, we show that two right adjoints to the same functor are naturally isomorphic.

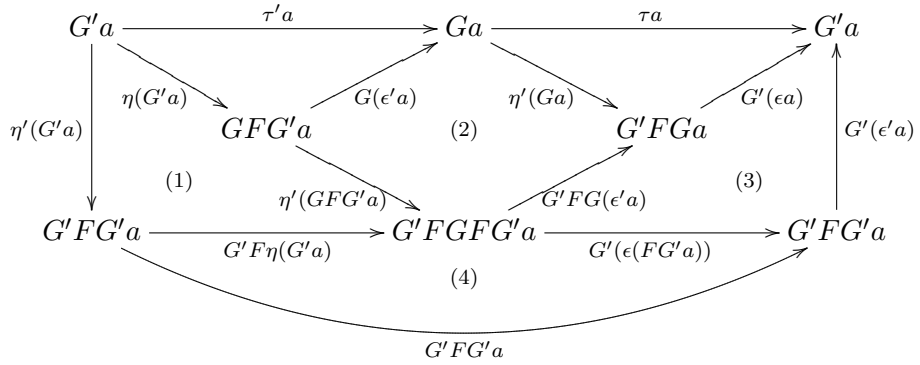
theorem *two-right-adjoints-naturally-isomorphic*:
assumes *adjoint-functors C D F G* **and** *adjoint-functors C D F G'*
shows *naturally-isomorphic C D G G'*
proof −

For any object x of C, we have that $\varepsilon\ x \in C.hom\ (F\ (G\ x))\ x$ is a terminal arrow from F to x, and similarly for $\varepsilon'\ x$. We may therefore obtain the unique coextension $\tau\ x \in D.hom\ (G\ x)\ (G'\ x)$ of $\varepsilon\ x$ along $\varepsilon'\ x$. An explicit formula for $\tau\ x$ is $D\ (G'\ (\varepsilon\ x))\ (\eta'\ (G\ x))$. Similarly, we obtain $\tau'\ x = D\ (G\ (\varepsilon'\ x))\ (\eta\ (G'\ x)) \in D.hom\ (G'\ x)\ (G\ x)$. We show these are the components of inverse natural transformations between G and G'.

```
obtain φ ψ where φψ: meta-adjunction C D F G φ ψ
  using assms adjoint-functors-def by blast
obtain φ' ψ' where φ'ψ': meta-adjunction C D F G' φ' ψ'
  using assms adjoint-functors-def by blast
interpret Adj: meta-adjunction C D F G φ ψ using φψ by auto
interpret S: replete-setcat .
interpret Adj: adjunction C D S.comp S.setp Adj.φC Adj.φD
                          F G φ ψ Adj.η Adj.ε Adj.Φ Adj.Ψ
  using Adj.induces-adjunction by auto
interpret Adj': meta-adjunction C D F G' φ' ψ' using φ'ψ' by auto
interpret Adj': adjunction C D S.comp S.setp Adj'.φC Adj'.φD
                          F G' φ' ψ' Adj'.η Adj'.ε Adj'.Φ Adj'.Ψ
  using Adj'.induces-adjunction by auto
write C (infixr ‹·C› 55)
write D (infixr ‹·D› 55)
write Adj.C.in-hom (‹«- : - →C -»›)
write Adj.D.in-hom (‹«- : - →D -»›)
let ?τo = λa. G' (Adj.ε a) ·D Adj'.η (G a)
interpret τ: transformation-by-components C D G G' ?τo
proof
  show ⋀a. Adj.C.ide a ⟹ «G' (Adj.ε a) ·D Adj'.η (G a) : G a →D G' a»
    by fastforce
  show ⋀f. Adj.C.arr f ⟹
            (G' (Adj.ε (Adj.C.cod f)) ·D Adj'.η (G (Adj.C.cod f))) ·D G f =
            G' f ·D G' (Adj.ε (Adj.C.dom f)) ·D Adj'.η (G (Adj.C.dom f))
  proof −
    fix f
    assume f: Adj.C.arr f
    let ?x = Adj.C.dom f
    let ?x' = Adj.C.cod f
    have (G' (Adj.ε (Adj.C.cod f)) ·D Adj'.η (G (Adj.C.cod f))) ·D G f =
         G' (Adj.ε (Adj.C.cod f) ·C F (G f)) ·D Adj'.η (G (Adj.C.dom f))
      using f Adj'.η.naturality [of G f] Adj.D.comp-assoc by simp
    also have ... = G' (f ·C Adj.ε (Adj.C.dom f)) ·D Adj'.η (G (Adj.C.dom f))
      using f Adj.ε.naturality by simp
    also have ... = G' f ·D G' (Adj.ε (Adj.C.dom f)) ·D Adj'.η (G (Adj.C.dom f))
      using f Adj.D.comp-assoc by simp
    finally show (G' (Adj.ε (Adj.C.cod f)) ·D Adj'.η (G (Adj.C.cod f))) ·D G f =
                 G' f ·D G' (Adj.ε (Adj.C.dom f)) ·D Adj'.η (G (Adj.C.dom f))
      by auto
  qed
qed
interpret natural-isomorphism C D G G' τ.map
```

```
proof
  fix a
  assume a: Adj.C.ide a
  show Adj.D.iso (τ.map a)
  proof
    show Adj.D.inverse-arrows (τ.map a) (φ (G' a) (Adj'.ε a))
    proof
```

The proof that the two composites are identities is a modest diagram chase. This is a good example of the inference rules for the *category*, *functor*, and *natural-transformation* locales in action. Isabelle is able to use the single hypothesis that a is an identity to implicitly fill in all the details that the various quantities are in fact arrows and that the indicated composites are all well-defined, as well as to apply associativity of composition. In most cases, this is done by auto or simp without even mentioning any of the rules that are used.




```
    show Adj.D.ide (τ.map a ·D φ (G' a) (Adj'.ε a))
    proof −
      have τ.map a ·D φ (G' a) (Adj'.ε a) = G' a
      proof −
        have τ.map a ·D φ (G' a) (Adj'.ε a) =
              G' (Adj.ε a) ·D (Adj'.η (G a) ·D G (Adj'.ε a)) ·D Adj.η (G' a)
          using a τ.map-simp-ide Adj.φ-in-terms-of-η Adj'.φ-in-terms-of-η
                Adj'.ε.preserves-hom [of a a a] Adj.C.ide-in-hom Adj.D.comp-assoc
                Adj.ε-def Adj.η-def
          by simp
        also have ... = G' (Adj.ε a) ·D (G' (F (G (Adj'.ε a))) ·D Adj'.η (G (F (G' a)))) ·D
                        Adj.η (G' a)
          using a Adj'.η.naturality [of G (Adj'.ε a)] by auto
        also have ... = (G' (Adj.ε a) ·D G' (F (G (Adj'.ε a)))) ·D G' (F (Adj.η (G' a))) ·D
                        Adj'.η (G' a)
          using a Adj'.η.naturality [of Adj.η (G' a)] Adj.D.comp-assoc by auto
        also have
            ... = G' (Adj'.ε a) ·D (G' (Adj.ε (F (G' a))) ·D G' (F (Adj.η (G' a)))) ·D
                  Adj'.η (G' a)
        proof −
          have
```

```
      G' (Adj.ε a) ·_D G' (F (G (Adj'.ε a))) = G' (Adj'.ε a) ·_D G' (Adj.ε (F (G' a)))
    proof −
      have G' (Adj.ε a ·_C F (G (Adj'.ε a))) = G' (Adj'.ε a ·_C Adj.ε (F (G' a)))
        using a Adj.ε.naturality [of Adj'.ε a] by auto
      thus ?thesis using a by force
    qed
    thus ?thesis using Adj.D.comp-assoc by auto
  qed
  also have ... = G' (Adj'.ε a) ·_D Adj'.η (G' a)
  proof −
    have G' (Adj.ε (F (G' a))) ·_D G' (F (Adj.η (G' a))) = G' (F (G' a))
    proof −
      have
        G' (Adj.ε (F (G' a))) ·_D G' (F (Adj.η (G' a))) = G' (Adj.εFoFη.map (G' a))
        using a Adj.εFoFη.map-simp-1 by auto
      moreover have Adj.εFoFη.map (G' a) = F (G' a)
        using a by (simp add: Adj.ηε.triangle-F)
      ultimately show ?thesis by auto
    qed
    thus ?thesis
      using a Adj.D.comp-cod-arr [of Adj'.η (G' a)] by auto
  qed
  also have ... = G' a
    using a Adj'.ηε.triangle-G Adj'.GεoηG.map-simp-1 [of a] by auto
  finally show ?thesis by auto
 qed
 thus ?thesis using a by simp
qed
show Adj.D.ide (φ (G' a) (Adj'.ε a) ·_D τ.map a)
proof −
 have φ (G' a) (Adj'.ε a) ·_D τ.map a = G a
 proof −
   have φ (G' a) (Adj'.ε a) ·_D τ.map a =
         G (Adj'.ε a) ·_D (Adj.η (G' a) ·_D G' (Adj.ε a)) ·_D Adj'.η (G a)
     using a τ.map-simp-ide Adj.φ-in-terms-of-η Adj'.ε.preserves-hom [of a a a]
           Adj.C.ide-in-hom Adj.D.comp-assoc Adj.η-def
     by auto
   also have
     ... = G (Adj'.ε a) ·_D (G (F (G' (Adj.ε a))) ·_D Adj.η (G' (F (G a)))) ·_D
           Adj'.η (G a)
     using a Adj.η.naturality [of G' (Adj.ε a)] by auto
   also have
     ... = (G (Adj'.ε a) ·_D G (F (G' (Adj.ε a)))) ·_D G (F (Adj'.η (G a))) ·_D
           Adj.η (G a)
     using a Adj.η.naturality [of Adj'.η (G a)] Adj.D.comp-assoc by auto
   also have
     ... = G (Adj.ε a) ·_D (G (Adj'.ε (F (G a))) ·_D G (F (Adj'.η (G a)))) ·_D
           Adj.η (G a)
   proof −
```

```
          have G (Adj'.ε a) ·_D G (F (G' (Adj.ε a))) = G (Adj.ε a) ·_D G (Adj'.ε (F (G a)))
           proof −
             have G (Adj'.ε a ·_C F (G' (Adj.ε a))) = G (Adj.ε a ·_C Adj'.ε (F (G a)))
               using a Adj'.ε.naturality [of Adj.ε a] by auto
             thus ?thesis using a by force
           qed
           thus ?thesis using Adj.D.comp-assoc by auto
         qed
         also have ... = G (Adj.ε a) ·_D Adj.η (G a)
         proof −
           have G (Adj'.ε (F (G a))) ·_D G (F (Adj'.η (G a))) = G (F (G a))
           proof −
             have
               G (Adj'.ε (F (G a))) ·_D G (F (Adj'.η (G a))) = G (Adj'.εFoFη.map (G a))
               using a Adj'.εFoFη.map-simp-1 [of G a] by auto
             moreover have Adj'.εFoFη.map (G a) = F (G a)
               using a by (simp add: Adj'.ηε.triangle-F)
             ultimately show ?thesis by auto
           qed
           thus ?thesis
             using a Adj.D.comp-cod-arr by auto
         qed
         also have ... = G a
           using a Adj.ηε.triangle-G Adj.GεoηG.map-simp-1 [of a] by auto
         finally show ?thesis by auto
       qed
       thus ?thesis using a by auto
     qed
   qed
  qed
  have natural-isomorphism C D G G' τ.map ..
  thus naturally-isomorphic C D G G'
    using naturally-isomorphic-def by blast
 qed

end
```

Chapter 17

Equivalence of Categories

In this chapter we define the notions of equivalence and adjoint equivalence of categories and establish some properties of functors that are part of an equivalence.

theory *EquivalenceOfCategories*
imports *Adjunction*
begin

locale *equivalence-of-categories* =
  C: *category C* +
  D: *category D* +
  F: *functor D C F* +
  G: *functor C D G* +
  η: *natural-isomorphism D D D.map G o F η* +
  ε: *natural-isomorphism C C F o G C.map ε*
for *C* :: *'c comp* (**infixr** ‹·$_C$› *55*)
and *D* :: *'d comp* (**infixr** ‹·$_D$› *55*)
and *F* :: *'d ⇒ 'c*
and *G* :: *'c ⇒ 'd*
and *η* :: *'d ⇒ 'd*
and *ε* :: *'c ⇒ 'c*
begin

  notation *C.in-hom* (‹«- : - →$_C$ -»›)
  notation *D.in-hom* (‹«- : - →$_D$ -»›)

  lemma *C-arr-expansion*:
  assumes *C.arr f*
  shows *ε (C.cod f) ·$_C$ F (G f) ·$_C$ C.inv (ε (C.dom f)) = f*
  and *C.inv (ε (C.cod f)) ·$_C$ f ·$_C$ ε (C.dom f) = F (G f)*
  proof −
    have *ε-dom*: *C.inverse-arrows (ε (C.dom f)) (C.inv (ε (C.dom f)))*
      using *assms C.inv-is-inverse* **by** *auto*
    have *ε-cod*: *C.inverse-arrows (ε (C.cod f)) (C.inv (ε (C.cod f)))*
      using *assms C.inv-is-inverse* **by** *auto*
    have *ε (C.cod f) ·$_C$ F (G f) ·$_C$ C.inv (ε (C.dom f)) =*

```
      (ε (C.cod f) ·C F (G f)) ·C C.inv (ε (C.dom f))
    using C.comp-assoc by force
  also have 1: ... = (f ·C ε (C.dom f)) ·C C.inv (ε (C.dom f))
    using assms ε.naturality by simp
  also have 2: ... = f
    using assms ε-dom C.comp-arr-inv C.comp-arr-dom C.comp-assoc by force
  finally show ε (C.cod f) ·C F (G f) ·C C.inv (ε (C.dom f)) = f by blast
  show C.inv (ε (C.cod f)) ·C f ·C ε (C.dom f) = F (G f)
    using assms 1 2 ε-dom ε-cod C.invert-side-of-triangle C.isoI C.iso-inv-iso
    by metis
qed

lemma G-is-faithful:
shows faithful-functor C D G
proof
  fix f f'
  assume par: C.par f f' and eq: G f = G f'
  show f = f'
  proof −
    have C.inv (ε (C.cod f)) ∈ C.hom (C.cod f) (F (G (C.cod f))) ∧
          C.iso (C.inv (ε (C.cod f)))
      using par by auto
    moreover have 1: ε (C.dom f) ∈ C.hom (F (G (C.dom f))) (C.dom f) ∧
                      C.iso (ε (C.dom f))
      using par by auto
    ultimately have 2: f ·C ε (C.dom f) = f' ·C ε (C.dom f)
      using par C-arr-expansion eq C.iso-is-section C.section-is-mono
      by (metis C-arr-expansion(1) eq)
    show ?thesis
    proof −
      have C.epi (ε (C.dom f))
        using 1 par C.iso-is-retraction C.retraction-is-epi by blast
      thus ?thesis using 2 par
        by (metis C-arr-expansion(1) eq)
    qed
  qed
qed

lemma G-is-essentially-surjective:
shows essentially-surjective-functor C D G
proof
  fix b
  assume b: D.ide b
  have C.ide (F b) ∧ D.isomorphic (G (F b)) b
  proof
    show C.ide (F b) using b by simp
    show D.isomorphic (G (F b)) b
    proof (unfold D.isomorphic-def)
      have «D.inv (η b) : G (F b) →D b» ∧ D.iso (D.inv (η b))
```

```
      using b by auto
    thus ∃f. «f : G (F b) →_D b» ∧ D.iso f by blast
  qed
 qed
 thus ∃a. C.ide a ∧ D.isomorphic (G a) b
  by blast
qed

interpretation ε-inv: inverse-transformation C C ‹F o G› C.map ε ..
interpretation η-inv: inverse-transformation D D D.map ‹G o F› η ..
interpretation GF: equivalence-of-categories D C G F ε-inv.map η-inv.map ..

lemma F-is-faithful:
shows faithful-functor D C F
 using GF.G-is-faithful by simp

lemma F-is-essentially-surjective:
shows essentially-surjective-functor D C F
 using GF.G-is-essentially-surjective by simp

lemma G-is-full:
shows full-functor C D G
proof
 fix a a' g
 assume a: C.ide a and a': C.ide a'
 assume g: «g : G a →_D G a'»
 show ∃f. «f : a →_C a'» ∧ G f = g
 proof
  have εa: C.inverse-arrows (ε a) (C.inv (ε a))
   using a C.inv-is-inverse by auto
  have εa': C.inverse-arrows (ε a') (C.inv (ε a'))
   using a' C.inv-is-inverse by auto
  let ?f = ε a' ·_C F g ·_C C.inv (ε a)
  have f: «?f : a →_C a'»
   using a a' g εa εa' ε.preserves-hom [of a' a' a'] ε-inv.preserves-hom [of a a a]
   by fastforce
  moreover have G ?f = g
  proof −
   interpret F: faithful-functor D C F
    using F-is-faithful by auto
   have F (G ?f) = F g
   proof −
    have F (G ?f) = C.inv (ε a') ·_C ?f ·_C ε a
     using f C-arr-expansion(2) [of ?f] by auto
    also have ... = (C.inv (ε a') ·_C ε a') ·_C F g ·_C C.inv (ε a) ·_C ε a
     using a a' f g C.comp-assoc by fastforce
    also have ... = F g
     using a a' g εa εa' C.comp-inv-arr C.comp-arr-dom C.comp-cod-arr by auto
    finally show ?thesis by blast
```

```
      qed
      moreover have D.par (G (ε a' ·_C F g ·_C C.inv (ε a))) g
        using f g by fastforce
      ultimately show ?thesis using f g F.is-faithful by blast
    qed
    ultimately show «?f : a →_C a'» ∧ G ?f = g by blast
  qed
qed

end



context equivalence-of-categories
begin

  interpretation ε-inv: inverse-transformation C C ‹F o G› C.map ε ..
  interpretation η-inv: inverse-transformation D D D.map ‹G o F› η ..
  interpretation GF: equivalence-of-categories D C G F ε-inv.map η-inv.map ..

  lemma F-is-full:
  shows full-functor D C F
    using GF.G-is-full by simp

end
```

Traditionally the term "equivalence of categories" is also used for a functor that is part of an equivalence of categories. However, it seems best to use that term for a situation in which all of the structure of an equivalence is explicitly given, and to have a different term for one of the functors involved.

```
locale equivalence-functor =
  C: category C +
  D: category D +
  functor C D G
for C :: 'c comp      (infixr ‹·_C› 55)
and D :: 'd comp      (infixr ‹·_D› 55)
and G :: 'c ⇒ 'd +
assumes induces-equivalence: ∃ F η ε. equivalence-of-categories C D F G η ε
begin

  notation C.in-hom    (‹«- : - →_C -»›)
  notation D.in-hom    (‹«- : - →_D -»›)

end

sublocale equivalence-of-categories ⊆ equivalence-functor C D G
  using equivalence-of-categories-axioms by (unfold-locales, blast)
```

An equivalence functor is fully faithful and essentially surjective.

sublocale *equivalence-functor* $\subseteq$ *fully-faithful-functor C D G*
proof −
  obtain *F* η ε **where** *1*: *equivalence-of-categories C D F G* η ε
    using *induces-equivalence* **by** *blast*
  interpret *equivalence-of-categories C D F G* η ε
    using *1* **by** *auto*
  show *fully-faithful-functor C D G*
    using *G-is-full G-is-faithful fully-faithful-functor.intro* **by** *auto*
qed

sublocale *equivalence-functor* $\subseteq$ *essentially-surjective-functor C D G*
proof −
  obtain *F* η ε **where** *1*: *equivalence-of-categories C D F G* η ε
    using *induces-equivalence* **by** *blast*
  interpret *equivalence-of-categories C D F G* η ε
    using *1* **by** *auto*
  show *essentially-surjective-functor C D G*
    using *G-is-essentially-surjective* **by** *auto*
qed

lemma (**in** *inverse-functors*) *induce-equivalence*:
shows *equivalence-of-categories A B F G B.map A.map*
  using *inv inv' A.extensionality B.extensionality B.comp-arr-dom B.comp-cod-arr*
      A.comp-arr-dom A.comp-cod-arr
  by *unfold-locales auto*

lemma (**in** *invertible-functor*) *is-equivalence*:
shows *equivalence-functor A B G*
  using *equivalence-functor-axioms.intro equivalence-functor-def equivalence-of-categories-def*
      induce-equivalence
  by *blast*

lemma (**in** *identity-functor*) *is-equivalence*:
shows *equivalence-functor C C map*
proof −
  interpret *inverse-functors C C map map*
    using *map-def* **by** *unfold-locales auto*
  interpret *invertible-functor C C map*
    using *inverse-functors-axioms*
    by *unfold-locales blast*
  show *?thesis*
    using *is-equivalence* **by** *blast*
qed

A special case of an equivalence functor is an endofunctor F equipped with a natural isomorphism from F to the identity functor.

context *endofunctor*
begin

```
    lemma isomorphic-to-identity-is-equivalence:
    assumes natural-isomorphism A A F A.map φ
    shows equivalence-functor A A F
    proof −
      interpret φ: natural-isomorphism A A F A.map φ
        using assms by auto
      interpret φ′: inverse-transformation A A F A.map φ ..
      interpret Fφ′: natural-isomorphism A A F ‹F o F› ‹F o φ′.map›
      proof −
        interpret Fφ′: natural-transformation A A F ‹F o F› ‹F o φ′.map›
          using φ′.natural-transformation-axioms functor-axioms
                horizontal-composite [of A A A.map F φ′.map A F F F]
          by simp
        show natural-isomorphism A A F (F o F) (F o φ′.map)
          apply unfold-locales
          using φ′.components-are-iso by fastforce
      qed
      interpret Fφ′oφ′: vertical-composite A A A.map F ‹F o F› φ′.map ‹F o φ′.map› ..
      interpret Fφ′oφ′: natural-isomorphism A A A.map ‹F o F› Fφ′oφ′.map
        using φ′.natural-isomorphism-axioms Fφ′.natural-isomorphism-axioms
              natural-isomorphisms-compose
        by fast
      interpret inv-Fφ′oφ′: inverse-transformation A A A.map ‹F o F› Fφ′oφ′.map ..
      interpret F: equivalence-of-categories A A F F Fφ′oφ′.map inv-Fφ′oφ′.map ..
      show ?thesis ..
    qed

  end

  locale dual-equivalence-of-categories =
    E: equivalence-of-categories
  begin

    interpretation Cop: dual-category C ..
    interpretation Dop: dual-category D ..
    interpretation Fop: dual-functor D C F ..
    interpretation Gop: dual-functor C D G ..
   interpretation Gop-o-Fop: composite-functor Dop.comp Cop.comp Dop.comp Fop.map Gop.map
..
   interpretation Fop-o-Gop: composite-functor Cop.comp Dop.comp Cop.comp Gop.map Fop.map
..
    sublocale η′: inverse-transformation D D E.D.map ‹G ∘ F› η ..
     interpretation ηop: natural-transformation Dop.comp Dop.comp Dop.map Gop-o-Fop.map
η′.map
      using η′.extensionality η′.naturality1 η′.naturality2
      by unfold-locales auto
     interpretation ηop: natural-isomorphism Dop.comp Dop.comp Dop.map Gop-o-Fop.map
η′.map
      by unfold-locales auto
```

sublocale ε': *inverse-transformation C C* ‹$F \circ G$› *E.C.map* ε **..**
interpretation εop: *natural-transformation Cop.comp Cop.comp Fop-o-Gop.map Cop.map* ε'*.map*
using ε'*.extensionality* ε'*.naturality1* ε'*.naturality2*
by *unfold-locales auto*
interpretation εop: *natural-isomorphism Cop.comp Cop.comp Fop-o-Gop.map Cop.map* ε'*.map*
by *unfold-locales auto*
sublocale *equivalence-of-categories Cop.comp Dop.comp Fop.map Gop.map* η'*.map* ε'*.map*
..

lemma *is-equivalence-of-categories*:
shows *equivalence-of-categories Cop.comp Dop.comp Fop.map Gop.map* η'*.map* ε'*.map*
..

end

locale *dual-equivalence-functor* =
G: *equivalence-functor*
begin

interpretation *Cop*: *dual-category C* **..**
interpretation *Dop*: *dual-category D* **..**
interpretation *Gop*: *dual-functor C D G* **..**

sublocale *equivalence-functor Cop.comp Dop.comp Gop.map*
proof −
obtain $F\ \eta\ \varepsilon$ **where** *F*: *equivalence-of-categories C D F G* $\eta\ \varepsilon$
using *G.equivalence-functor-axioms equivalence-functor-def*
equivalence-functor-axioms-def
by *blast*
interpret *E*: *equivalence-of-categories C D F G* $\eta\ \varepsilon$
using *F* **by** *blast*
interpret *dual-equivalence-of-categories C D F G* $\eta\ \varepsilon$ **..**
show *equivalence-functor Cop.comp Dop.comp Gop.map* **..**
qed

lemma *is-equivalence-functor*:
shows *equivalence-functor Cop.comp Dop.comp Gop.map*
..

end

An adjoint equivalence is an equivalence of categories that is also an adjunction.

locale *adjoint-equivalence* =
unit-counit-adjunction C D F G $\eta\ \varepsilon$ +
η: *natural-isomorphism D D D.map G o F* η +
ε: *natural-isomorphism C C F o G C.map* ε
for *C* :: *'c comp* (**infixr** ‹$\cdot_C$› *55*)

and $D :: 'd\ comp$ (**infixr** $\langle\cdot_D\rangle$ *55*)
and $F :: 'd \Rightarrow 'c$
and $G :: 'c \Rightarrow 'd$
and $\eta :: 'd \Rightarrow 'd$
and $\varepsilon :: 'c \Rightarrow 'c$

An adjoint equivalence is clearly an equivalence of categories.

sublocale *adjoint-equivalence* $\subseteq$ *equivalence-of-categories* **..**

context *adjoint-equivalence*
begin

The triangle identities for an adjunction reduce to inverse relations when η and ε are natural isomorphisms.

lemma *triangle-G'*:
assumes *C.ide a*
shows *D.inverse-arrows* $(\eta\ (G\ a))\ (G\ (\varepsilon\ a))$
proof
show *D.ide* $(G\ (\varepsilon\ a) \cdot_D \eta\ (G\ a))$
using *assms triangle-G GεoηG.map-simp-ide* **by** *fastforce*
thus *D.ide* $(\eta\ (G\ a) \cdot_D G\ (\varepsilon\ a))$
using *assms D.section-retraction-of-iso* [*of G* $(\varepsilon\ a)\ \eta\ (G\ a)$] **by** *auto*
qed

lemma *triangle-F'*:
assumes *D.ide b*
shows *C.inverse-arrows* $(F\ (\eta\ b))\ (\varepsilon\ (F\ b))$
proof
show *C.ide* $(\varepsilon\ (F\ b) \cdot_C F\ (\eta\ b))$
using *assms triangle-F εFoFη.map-simp-ide* **by** *auto*
thus *C.ide* $(F\ (\eta\ b) \cdot_C \varepsilon\ (F\ b))$
using *assms C.section-retraction-of-iso* [*of* $\varepsilon\ (F\ b)\ F\ (\eta\ b)$] **by** *auto*
qed

An adjoint equivalence can be dualized by interchanging the two functors and inverting the natural isomorphisms. This is somewhat awkward to prove, but probably useful to have done it once and for all.

lemma *dual-adjoint-equivalence*:
assumes *adjoint-equivalence C D F G η ε*
shows *adjoint-equivalence D C G F* (*inverse-transformation.map C C* (*C.map*) *ε*)
(*inverse-transformation.map D D* (*G o F*) *η*)
proof −
interpret *adjoint-equivalence C D F G η ε* **using** *assms* **by** *auto*
interpret *ε'*: *inverse-transformation C C* ‹*F o G*› *C.map ε* **..**
interpret *η'*: *inverse-transformation D D D.map* ‹*G o F*› *η* **..**
interpret *Gε'*: *natural-transformation C D G* ‹*G o F o G*› ‹*G o ε'.map*›
proof −
have *natural-transformation C D G* (*G o* (*F o G*)) (*G o ε'.map*)
using *G.as-nat-trans.natural-transformation-axioms ε'.natural-transformation-axioms*

```
        horizontal-composite
    by fastforce
  thus natural-transformation C D G (G o F o G) (G o ε'.map)
    using o-assoc by metis
qed
interpret η'G: natural-transformation C D ‹G o F o G› G ‹η'.map o G›
  using η'.natural-transformation-axioms G.as-nat-trans.natural-transformation-axioms
      horizontal-composite
  by fastforce
interpret ε'F: natural-transformation D C F ‹F o G o F› ‹ε'.map o F›
  using ε'.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
      horizontal-composite
  by fastforce
interpret Fη': natural-transformation D C ‹F o G o F› F ‹F o η'.map›
proof −
  have natural-transformation D C (F o (G o F)) F (F o η'.map)
    using η'.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
        horizontal-composite
    by fastforce
  thus natural-transformation D C (F o G o F) F (F o η'.map)
    using o-assoc by metis
qed
interpret Fη'oε'F: vertical-composite D C F ‹(F o G) o F› F ‹ε'.map o F› ‹F o η'.map›
..
interpret η'GoGε': vertical-composite C D G ‹G o F o G› G ‹G o ε'.map› ‹η'.map o G› ..
show ?thesis
proof
  show η'GoGε'.map = G
  proof (intro natural-transformation-eqI)
    show natural-transformation C D G G G
      using G.as-nat-trans.natural-transformation-axioms by auto
    show natural-transformation C D G G η'GoGε'.map
      using η'GoGε'.natural-transformation-axioms by auto
    show ⋀a. C.ide a ⟹ η'GoGε'.map a = G a
    proof −
      fix a
      assume a: C.ide a
      show η'GoGε'.map a = G a
        using a η'GoGε'.map-simp-ide triangle-G' G.preserves-ide
            η'.inverts-components ε'.inverts-components
            D.inverse-unique G.preserves-inverse-arrows GεoηG.map-simp-ide
            D.inverse-arrows-sym triangle-G
        by (metis o-apply)
    qed
  qed
  show Fη'oε'F.map = F
  proof (intro natural-transformation-eqI)
    show natural-transformation D C F F F
      using F.as-nat-trans.natural-transformation-axioms by auto
```

```
      show natural-transformation D C F F Fη'oε'F.map
        using Fη'oε'F.natural-transformation-axioms by auto
      show ⋀b. D.ide b ⟹ Fη'oε'F.map b = F b
      proof −
        fix b
        assume b: D.ide b
        show Fη'oε'F.map b = F b
          using b Fη'oε'F.map-simp-ide εFoFη.map-simp-ide triangle-F triangle-F'
                η'.inverts-components ε'.inverts-components F.preserves-ide
                C.inverse-unique F.preserves-inverse-arrows C.inverse-arrows-sym
          by (metis o-apply)
      qed
    qed
  qed
qed

end
```

Every fully faithful and essentially surjective functor underlies an adjoint equivalence. To prove this without repeating things that were already proved in *Category3.Adjunction*, we first show that a fully faithful and essentially surjective functor is a left adjoint functor, and then we show that if the left adjoint in a unit-counit adjunction is fully faithful and essentially surjective, then the unit and counit are natural isomorphisms; hence the adjunction is in fact an adjoint equivalence.

```
locale fully-faithful-and-essentially-surjective-functor =
  C: category C +
  D: category D +
  fully-faithful-functor C D F +
  essentially-surjective-functor C D F
  for C :: 'c comp      (infixr ‹·C› 55)
  and D :: 'd comp      (infixr ‹·D› 55)
  and F :: 'c ⇒ 'd
begin

  notation C.in-hom      (‹«- : - →C -»›)
  notation D.in-hom      (‹«- : - →D -»›)

  lemma is-left-adjoint-functor:
  shows left-adjoint-functor C D F
  proof
    fix y
    assume y: D.ide y
    let ?x = SOME x. C.ide x ∧ (∃e. D.iso e ∧ «e : F x →D y»)
    let ?e = SOME e. D.iso e ∧ «e : F ?x →D y»
    have ∃x e. D.iso e ∧ terminal-arrow-from-functor C D F x y e
    proof −
      have ∃x. D.iso ?e ∧ terminal-arrow-from-functor C D F x y ?e
      proof −
        have x: C.ide ?x ∧ (∃e. D.iso e ∧ «e : F ?x →D y»)
```

```
  using y essentially-surjective
      someI-ex [of λx. C.ide x ∧ (∃e. D.iso e ∧ «e : F x →_D y»)]
  by blast
hence e: D.iso ?e ∧ «?e : F ?x →_D y»
  using someI-ex [of λe. D.iso e ∧ «e : F ?x →_D y»] by blast
interpret arrow-from-functor C D F ?x y ?e
  using x e by (unfold-locales, simp)
interpret terminal-arrow-from-functor C D F ?x y ?e
proof
  fix x' f
  assume 1: arrow-from-functor C D F x' y f
  interpret f: arrow-from-functor C D F x' y f
    using 1 by simp
  have f: «f: F x' →_D y»
    by (meson f.arrow)
  show ∃!g. is-coext x' f g
  proof
    let ?g = SOME g. «g : x' →_C ?x» ∧ F g = D.inv ?e ·_D f
    have g: «?g : x' →_C ?x» ∧ F ?g = D.inv ?e ·_D f
      using f e x f.arrow is-full D.comp-in-homI D.inv-in-hom
          someI-ex [of λg. «g : x' →_C ?x» ∧ F g = D.inv ?e ·_D f]
      by auto
    show 1: is-coext x' f ?g
    proof −
      have «?g : x' →_C ?x»
        using g by simp
      moreover have ?e ·_D F ?g = f
      proof −
        have ?e ·_D F ?g = ?e ·_D D.inv ?e ·_D f
          using g by simp
        also have ... = (?e ·_D D.inv ?e) ·_D f
          using e f D.inv-in-hom by (metis D.comp-assoc)
        also have ... = f
        proof −
          have ?e ·_D D.inv ?e = y
            using e D.comp-arr-inv D.inv-is-inverse by auto
          thus ?thesis
            using f D.comp-cod-arr by auto
        qed
        finally show ?thesis by blast
      qed
      ultimately show ?thesis
        unfolding is-coext-def by simp
    qed
    show ⋀g'. is-coext x' f g' ⟹ g' = ?g
    proof −
      fix g'
      assume g': is-coext x' f g'
      have 2: «g' : x' →_C ?x» ∧ ?e ·_D F g' = f using g' is-coext-def by simp
```

```
        have 3: «?g : x' →C ?x» ∧ ?e ·D F ?g = f using 1 is-coext-def by simp
        have F g' = F ?g
          using e f 2 3 D.iso-is-section D.section-is-mono D.mono-cancel D.arrI
          by (metis (no-types, lifting) D.arrI)
        moreover have C.par g' ?g
          using 2 3 by fastforce
        ultimately show g' = ?g
          using is-faithful [of g' ?g] by simp
      qed
    qed
  qed
  show ?thesis
    using e terminal-arrow-from-functor-axioms by auto
 qed
 thus ?thesis by auto
qed
thus ∃x e. terminal-arrow-from-functor C D F x y e by blast
qed

lemma extends-to-adjoint-equivalence:
shows ∃G η ε. adjoint-equivalence C D G F η ε
proof −
 interpret left-adjoint-functor C D F
   using is-left-adjoint-functor by blast
 interpret Adj: meta-adjunction D C F G φ ψ
   using induces-meta-adjunction by simp
 interpret S: replete-setcat .
 interpret Adj: adjunction D C S.comp S.setp
                Adj.φC Adj.φD F G φ ψ Adj.η Adj.ε Adj.Φ Adj.Ψ
   using induces-adjunction by simp
 interpret equivalence-of-categories D C F G Adj.η Adj.ε
 proof
   show 1: ⋀a. D.ide a ⟹ D.iso (Adj.ε a)
   proof −
     fix a
     assume a: D.ide a
     interpret εa: terminal-arrow-from-functor C D F ‹G a› a ‹Adj.ε a›
       using a Adj.has-terminal-arrows-from-functor [of a] by blast
     have D.retraction (Adj.ε a)
     proof −
       obtain b φ where φ: C.ide b ∧ D.iso φ ∧ «φ: F b →D a»
         using a essentially-surjective by blast
       interpret φ: arrow-from-functor C D F b a φ
         using φ by (unfold-locales, simp)
       let ?g = εa.the-coext b φ
       have 1: «?g : b →C G a» ∧ Adj.ε a ·D F ?g = φ
         using φ.arrow-from-functor-axioms εa.the-coext-prop [of b φ] by simp
       have a = (Adj.ε a ·D F ?g) ·D D.inv φ
         using a 1 φ D.comp-cod-arr Adj.ε.preserves-hom D.invert-side-of-triangle(2)
```

```
    by auto
  also have ... = Adj.ε a ·_D F ?g ·_D D.inv φ
    using a 1 φ D.inv-in-hom Adj.ε.preserves-hom [of a a a] D.comp-assoc
    by blast
  finally have ∃f. D.ide (Adj.ε a ·_D f)
    using a by metis
  thus ?thesis
    unfolding D.retraction-def by blast
qed
moreover have D.mono (Adj.ε a)
proof
  show D.arr (Adj.ε a)
    using a by simp
  show ⋀f f'. ⟦D.seq (Adj.ε a) f; Adj.ε a ·_D f = Adj.ε a ·_D f'⟧
                ⟹ f = f'
  proof −
    fix f f'
    assume seq: D.seq (Adj.ε a) f
    and eq: Adj.ε a ·_D f = Adj.ε a ·_D f'
    have f: «f : D.dom f →_D F (G a)»
      using a seq Adj.ε.preserves-hom [of a a a] by fastforce
    have f': «f' : D.dom f' →_D F (G a)»
      using a seq Adj.ε.preserves-hom [of a a a] D.in-homI eq by auto
    have par: D.par f f'
      using f f' seq eq D.dom-comp [of Adj.ε a f] by force
    obtain b' φ where φ: C.ide b' ∧ D.iso φ ∧ «φ: F b' →_D D.dom f»
      using par essentially-surjective D.ide-dom [of f] by blast
    have 1: Adj.ε a ·_D f ·_D φ = Adj.ε a ·_D f' ·_D φ
      using eq φ par D.comp-assoc by metis
    obtain g where g: «g : b' →_C G a» ∧ F g = f ·_D φ
      using a f φ is-full [of G a b' f ·_D φ] by auto
    obtain g' where g': «g' : b' →_C G a» ∧ F g' = f' ·_D φ
      using a f' par φ is-full [of G a b' f' ·_D φ] by auto
    interpret fφ: arrow-from-functor C D F b' a ‹Adj.ε a ·_D f ·_D φ›
      using a φ f Adj.ε.preserves-hom
      by (unfold-locales, fastforce)
    interpret f'φ: arrow-from-functor C D F b' a ‹Adj.ε a ·_D f' ·_D φ›
      using a φ f' par Adj.ε.preserves-hom
      by (unfold-locales, fastforce)
    have εa.is-coext b' (Adj.ε a ·_D f ·_D φ) g
      unfolding εa.is-coext-def using g 1 by auto
    moreover have εa.is-coext b' (Adj.ε a ·_D f' ·_D φ) g'
      unfolding εa.is-coext-def using g' 1 by auto
    ultimately have g = g'
      using 1 fφ.arrow-from-functor-axioms f'φ.arrow-from-functor-axioms
            εa.the-coext-unique εa.the-coext-unique [of b' Adj.ε a ·_D f' ·_D φ g']
      by auto
    hence f ·_D φ = f' ·_D φ
      using g g' is-faithful by argo
```

```
        thus f = f'
          using φ f f' par D.iso-is-retraction D.retraction-is-epi
              D.epi-cancel [of φ f f']
          by auto
      qed
    qed
    ultimately show D.iso (Adj.ε a)
      using D.iso-iff-mono-and-retraction by simp
  qed
  interpret ε: natural-isomorphism D D ‹F o G› D.map Adj.ε
    using 1 by (unfold-locales, auto)
  interpret εF: natural-isomorphism C D ‹F o G o F› F ‹Adj.ε o F›
    using ε.components-are-iso by (unfold-locales, simp)
  show ⋀a. C.ide a ⟹ C.iso (Adj.η a)
  proof −
    fix a
    assume a: C.ide a
    have D.inverse-arrows ((Adj.ε o F) a) ((F o Adj.η) a)
      using a ε.components-are-iso Adj.ηε.triangle-F Adj.εFoFη.map-simp-ide
          D.section-retraction-of-iso
      by simp
    hence D.iso ((F o Adj.η) a)
      by blast
    thus C.iso (Adj.η a)
      using a reflects-iso [of Adj.η a] by fastforce
  qed
qed

interpret adjoint-equivalence D C F G Adj.η Adj.ε ..
interpret ε': inverse-transformation D D ‹F o G› D.map Adj.ε ..
interpret η': inverse-transformation C C C.map ‹G o F› Adj.η ..
interpret E: adjoint-equivalence C D G F ε'.map η'.map
  using adjoint-equivalence-axioms dual-adjoint-equivalence by blast
show ?thesis
  using E.adjoint-equivalence-axioms by auto
qed

lemma is-right-adjoint-functor:
shows right-adjoint-functor C D F
proof −
  obtain G η ε where E: adjoint-equivalence C D G F η ε
    using extends-to-adjoint-equivalence by auto
  interpret E: adjoint-equivalence C D G F η ε
    using E by simp
  interpret Adj: meta-adjunction C D G F E.φ E.ψ
    using E.induces-meta-adjunction by simp
  show ?thesis
    using Adj.has-right-adjoint-functor by simp
qed
```

```
  lemma is-equivalence-functor:
  shows equivalence-functor C D F
  proof
    obtain G η ε where E: adjoint-equivalence C D G F η ε
      using extends-to-adjoint-equivalence by auto
    interpret E: adjoint-equivalence C D G F η ε
      using E by simp
    have equivalence-of-categories C D G F η ε ..
    thus ∃ G η ε. equivalence-of-categories C D G F η ε by blast
  qed

  sublocale equivalence-functor C D F
    using is-equivalence-functor by blast

end

context equivalence-of-categories
begin
```

The following development shows that an equivalence of categories can be refined to an adjoint equivalence by replacing just the counit.

```
  abbreviation ε'
  where ε' a ≡ ε a ·C F (D.inv (η (G a))) ·C C.inv (ε (F (G a)))

  interpretation ε': transformation-by-components C C ‹F ∘ G› C.map ε'
  proof
    show ⋀a. C.ide a ⟹ «ε' a : (F ∘ G) a →C C.map a»
      using η.components-are-iso ε.components-are-iso by simp
    fix f
    assume f: C.arr f
    show ε' (C.cod f) ·C (F ∘ G) f = C.map f ·C ε' (C.dom f)
    proof −
      have ε' (C.cod f) ·C (F ∘ G) f =
           ε (C.cod f) ·C F (D.inv (η (G (C.cod f)))) ·C C.inv (ε (F (G (C.cod f)))) ·C F (G
f)
        using f C.comp-assoc by simp
      also have ... = ε (C.cod f) ·C (F (D.inv (η (G (C.cod f)))) ·C
                        F (G (F (G f)))) ·C C.inv (ε (F (G (C.dom f))))
        using f ε.inv-naturality [of F (G f)] C.comp-assoc by simp
      also have ... = (ε (C.cod f) ·C F (G f)) ·C F (D.inv (η (G (C.dom f)))) ·C
                        C.inv (ε (F (G (C.dom f))))
      proof −
        have F (D.inv (η (G (C.cod f)))) ·C F (G (F (G f))) =
             F (G f) ·C F (D.inv (η (G (C.dom f))))
        proof −
          have F (D.inv (η (G (C.cod f)))) ·C F (G (F (G f))) =
               F (D.inv (η (G (C.cod f)))) ·D G (F (G f)))
            using f by simp
```

```
      also have ... = F (G f ·_D D.inv (η (G (C.dom f))))
        using f η.inv-naturality [of G f] by simp
      also have ... = F (G f) ·_C F (D.inv (η (G (C.dom f))))
        using f by simp
      finally show ?thesis by blast
    qed
    thus ?thesis
      using C.comp-assoc by simp
  qed
  also have ... = C.map f ·_C ε (C.dom f) ·_C F (D.inv (η (G (C.dom f)))) ·_C
                  C.inv (ε (F (G (C.dom f))))
    using f ε.naturality C.comp-assoc by simp
  finally show ?thesis by blast
 qed
qed

interpretation ε': natural-isomorphism C C ‹F ∘ G› C.map ε'.map
proof
 show ⋀a. C.ide a ⟹ C.iso (ε'.map a)
  unfolding ε'.map-def
  using η.components-are-iso ε.components-are-iso
  apply simp
  by (intro C.isos-compose) auto
qed

lemma Fη-inverse:
assumes D.ide b
shows F (η (G (F b))) = F (G (F (η b)))
and F (η b) ·_C ε (F b) = ε (F (G (F b))) ·_C F (η (G (F b)))
and C.inverse-arrows (F (η b)) (ε' (F b))
and F (η b) = C.inv (ε' (F b))
and C.inv (F (η b)) = ε' (F b)
proof −
 let ?ε' = λa. ε a ·_C F (D.inv (η (G a))) ·_C C.inv (ε (F (G a)))
 show 1: F (η (G (F b))) = F (G (F (η b)))
 proof −
  have F (η (G (F b))) ·_C F (η b) = F (G (F (η b))) ·_C F (η b)
  proof −
   have F (η (G (F b))) ·_C F (η b) = F (η (G (F b)) ·_D η b)
    using assms by simp
   also have ... = F (G (F (η b)) ·_D η b)
    using assms η.naturality [of η b] by simp
   also have ... = F (G (F (η b))) ·_C F (η b)
    using assms by simp
   finally show ?thesis by blast
  qed
  thus ?thesis
   using assms η.components-are-iso C.iso-cancel-right by simp
 qed
```

```
    show F (η b) ⋅C ε (F b) = ε (F (G (F b))) ⋅C F (η (G (F b)))
      using assms 1 ε.naturality [of F (η b)] by simp
    show 2: C.inverse-arrows (F (η b)) (?ε' (F b))
    proof
      show 3: C.ide (?ε' (F b) ⋅C F (η b))
      proof −
        have ?ε' (F b) ⋅C F (η b) =
              ε (F b) ⋅C (F (D.inv (η (G (F b)))) ⋅C C.inv (ε (F (G (F b))))) ⋅C F (η b)
          using C.comp-assoc by simp
        also have ... = ε (F b) ⋅C (F (D.inv (η (G (F b)))) ⋅C F (G (F (η b)))) ⋅C C.inv (ε (F
b))
          using assms ε.naturality [of F (η b)] ε.components-are-iso C.comp-assoc
                C.invert-opposite-sides-of-square
                  [of ε (F (G (F b))) F (G (F (η b))) F (η b) ε (F b)]
          by simp
        also have ... = ε (F b) ⋅C C.inv (ε (F b))
        proof −
          have F (D.inv (η (G (F b)))) ⋅C F (G (F (η b))) = F (G (F b))
            using assms 1 D.comp-inv-arr' η.components-are-iso
            by (metis D.ideD(1) D.ideD(2) F.preserves-comp
                F.preserves-ide G.preserves-ide η.preserves-dom D.map-simp)
          moreover have F (G (F b)) ⋅C C.inv (ε (F b)) = C.inv (ε (F b))
            using assms D.comp-cod-arr ε.components-are-iso C.inv-in-hom [of ε (F b)]
            by (metis C.comp-ide-arr C-arr-expansion(1) D.ide-char F.preserves-arr
                F.preserves-dom F.preserves-ide G.preserves-ide C.seqE)
          ultimately show ?thesis by simp
        qed
        also have ... = F b
          using assms ε.components-are-iso C.comp-arr-inv' by simp
        finally have (ε (F b) ⋅C F (D.inv (η (G (F b)))) ⋅C C.inv (ε (F (G (F b))))) ⋅C F (η
b) = F b
          by blast
        thus ?thesis
          using assms by simp
      qed
      show C.ide (F (η b) ⋅C ?ε' (F b))
      proof −
        have (F (η b) ⋅C ?ε' (F b)) ⋅C F (η b) = F (G (F b)) ⋅C F (η b)
        proof −
          have (F (η b) ⋅C ?ε' (F b)) ⋅C F (η b) =
                F (η b) ⋅C (ε (F b) ⋅C F (D.inv (η (G (F b)))) ⋅C C.inv (ε (F (G (F b))))) ⋅C F
(η b)
            using C.comp-assoc by simp
          also have ... = F (η b)
            using assms 3
                  C.comp-arr-dom
                    [of F (η b) (ε (F b) ⋅C F (D.inv (η (G (F b)))) ⋅C
                                 C.inv (ε (F (G (F b))))) ⋅C F (η b)]
            by auto
```

```
      also have ... = F (G (F b)) ·_C F (η b)
        using assms C.comp-cod-arr by simp
      finally show ?thesis by blast
    qed
    hence F (η b) ·_C ?ε' (F b) = F (G (F b))
      using assms C.iso-cancel-right by simp
    thus ?thesis
      using assms by simp
  qed
 qed
 show C.inv (F (η b)) = ?ε' (F b)
  using assms 2 C.inverse-unique by simp
 show F (η b) = C.inv (?ε' (F b))
 proof −
  have C.inverse-arrows (?ε' (F b)) (F (η b))
    using assms 2 by auto
  thus ?thesis
    using assms C.inverse-unique by simp
 qed
qed

interpretation FoGoF: composite-functor D C C F ‹F o G› ..
interpretation GoFoG: composite-functor C D D G ‹G o F› ..

interpretation natural-transformation D C F FoGoF.map ‹F ∘ η›
proof −
 have F ∘ D.map = F
  using hcomp-ide-dom F.as-nat-trans.natural-transformation-axioms by blast
 moreover have F o (G o F) = FoGoF.map
  by auto
 ultimately show natural-transformation D C F FoGoF.map (F ∘ η)
  using η.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
        horizontal-composite [of D D D.map G o F η C F F F]
  by simp
qed

interpretation natural-transformation C D G GoFoG.map ‹η ∘ G›
 using η.natural-transformation-axioms G.as-nat-trans.natural-transformation-axioms
       horizontal-composite [of C D G G G ]
 by fastforce

interpretation natural-transformation D C FoGoF.map F ‹ε'.map ∘ F›
 using ε'.natural-transformation-axioms F.as-nat-trans.natural-transformation-axioms
       horizontal-composite [of D C F F F]
 by fastforce

interpretation natural-transformation C D GoFoG.map G ‹G ∘ ε'.map›
proof −
 have G o C.map = G
```

```
    using hcomp-ide-dom G.as-nat-trans.natural-transformation-axioms by blast
  moreover have G o (F o G) = GoFoG.map
    by auto
  ultimately show natural-transformation C D GoFoG.map G (G ∘ ε'.map)
    using G.as-nat-trans.natural-transformation-axioms ε'.natural-transformation-axioms
          horizontal-composite [of C C F o G C.map ε'.map D G G G]
    by simp
qed

interpretation ε'F-Fη: vertical-composite D C F FoGoF.map F ‹F ∘ η› ‹ε'.map ∘ F› ..
interpretation Gε'-ηG: vertical-composite C D G GoFoG.map G ‹η o G› ‹G o ε'.map› ..

interpretation ηε': unit-counit-adjunction C D F G η ε'.map
proof
  show 1: ε'F-Fη.map = F
  proof
    fix g
    show ε'F-Fη.map g = F g
    proof (cases D.arr g)
      show ¬ D.arr g ⟹ ε'F-Fη.map g = F g
        using ε'F-Fη.extensionality F.extensionality by simp
      assume g: D.arr g
      have ε'F-Fη.map g = ε' (F (D.cod g)) ·C F (η g)
        using g ε'F-Fη.map-def by simp
      also have ... = ε' (F (D.cod g)) ·C F (η (D.cod g) ·D g)
        using g η.naturality2 by simp
      also have ... = (ε' (F (D.cod g)) ·C F (η (D.cod g))) ·C F g
        using g C.comp-assoc by simp
      also have ... = F (D.cod g) ·C F g
        using g Fη-inverse(3) [of D.cod g] by fastforce
      also have ... = F g
        using g C.comp-cod-arr by simp
      finally show ε'F-Fη.map g = F g by blast
    qed
  qed
  show Gε'-ηG.map = G
  proof
    fix f
    show Gε'-ηG.map f = G f
    proof (cases C.arr f)
      show ¬ C.arr f ⟹ Gε'-ηG.map f = G f
        using Gε'-ηG.extensionality G.extensionality by simp
      assume f: C.arr f
      have F (Gε'-ηG.map f) = F (G (ε' (C.cod f)) ·D η (G f))
        using f Gε'-ηG.map-def D.comp-assoc by simp
      also have ... = F (G (ε' (C.cod f)) ·D η (G (C.cod f)) ·D G f)
        using f η.naturality2 [of G f] by simp
      also have ... = F (G (ε' (C.cod f))) ·C F (η (G (C.cod f))) ·C F (G f)
        using f by simp
```

```
        also have ... = (F (G (ε' (C.cod f))) ·_C C.inv (ε' (F (G (C.cod f))))) ·_C F (G f)
          using f Fη-inverse(4) C.comp-assoc by simp
        also have ... = (C.inv (ε' (C.cod f)) ·_C ε' (C.cod f)) ·_C F (G f)
          using f ε'.inv-naturality [of ε' (C.cod f)] by simp
        also have ... = F (G (C.cod f)) ·_C F (G f)
          using f C.comp-inv-arr' [of ε' (C.cod f)] ε'.components-are-iso by simp
        also have ... = F (G f)
          using f C.comp-cod-arr by simp
        finally have F (Gε'-ηG.map f) = F (G f) by blast
        moreover have D.par (Gε'-ηG.map f) (G f)
          using f by simp
        ultimately show Gε'-ηG.map f = G f
          using f F-is-faithful
          by (simp add: faithful-functor-axioms-def faithful-functor-def)
      qed
    qed
  qed

  interpretation ηε': adjoint-equivalence C D F G η ε'.map ..

  lemma refines-to-adjoint-equivalence:
  shows adjoint-equivalence C D F G η ε'.map
    ..

 end

end
```

Chapter 18

FreeCategory

theory *FreeCategory*
imports *Category ConcreteCategory*
begin

This theory defines locales for constructing the free category generated by a graph, as well as some special cases, including the discrete category generated by a set of objects, the "quiver" generated by a set of arrows, and a "parallel pair" of arrows, which is the diagram shape required for equalizers. Other diagram shapes can be constructed in a similar fashion.

18.1 Graphs

The following locale gives a definition of graphs in a traditional style.

locale *graph* =
fixes *Obj* :: *'obj set*
and *Arr* :: *'arr set*
and *Dom* :: *'arr* ⇒ *'obj*
and *Cod* :: *'arr* ⇒ *'obj*
assumes *dom-is-obj*: $x \in Arr \Longrightarrow Dom\ x \in Obj$
and *cod-is-obj*: $x \in Arr \Longrightarrow Cod\ x \in Obj$
begin

The list of arrows p forms a path from object x to object y if the domains and codomains of the arrows match up in the expected way.

definition *path*
where $path\ x\ y\ p \equiv (p = [] \wedge x = y \wedge x \in Obj) \vee$
$(p \neq [] \wedge x = Dom\ (hd\ p) \wedge y = Cod\ (last\ p) \wedge$
$(\forall n.\ n \geq 0 \wedge n < length\ p \longrightarrow nth\ p\ n \in Arr) \wedge$
$(\forall n.\ n \geq 0 \wedge n < (length\ p) - 1 \longrightarrow Cod\ (nth\ p\ n) = Dom\ (nth\ p\ (n+1))))$

lemma *path-Obj*:
assumes $x \in Obj$
shows $path\ x\ x\ []$

```
  using assms path-def by simp

lemma path-single-Arr:
assumes x ∈ Arr
shows path (Dom x) (Cod x) [x]
  using assms path-def by simp

lemma path-concat:
assumes path x y p and path y z q
shows path x z (p @ q)
proof −
  have p = [] ∨ q = [] ⟹ ?thesis
    using assms path-def by auto
  moreover have p ≠ [] ∧ q ≠ [] ⟹ ?thesis
  proof −
    assume pq: p ≠ [] ∧ q ≠ []
    have Cod-last: Cod (last p) = Cod (nth (p @ q) ((length p)−1))
      using assms pq by (simp add: last-conv-nth nth-append)
    moreover have Dom-hd: Dom (hd q) = Dom (nth (p @ q) (length p))
      using assms pq by (simp add: hd-conv-nth less-not-refl2 nth-append)
    show ?thesis
    proof −
      have 1: ⋀n. n ≥ 0 ∧ n < length (p @ q) ⟹ nth (p @ q) n ∈ Arr
      proof −
        fix n
        assume n: n ≥ 0 ∧ n < length (p @ q)
        have (n ≥ 0 ∧ n < length p) ∨ (n ≥ length p ∧ n < length (p @ q))
          using n by auto
        thus nth (p @ q) n ∈ Arr
          using assms pq nth-append path-def le-add-diff-inverse length-append
                less-eq-nat.simps(1) nat-add-left-cancel-less
          by metis
      qed
      have 2: ⋀n. n ≥ 0 ∧ n < length (p @ q) − 1 ⟹
                          Cod (nth (p @ q) n) = Dom (nth (p @ q) (n+1))
      proof −
        fix n
        assume n: n ≥ 0 ∧ n < length (p @ q) − 1
        have 1: (n ≥ 0 ∧ n < (length p) − 1) ∨ (n ≥ length p ∧ n < length (p @ q) − 1)
                  ∨ n = (length p) − 1
          using n by auto
        thus Cod (nth (p @ q) n) = Dom (nth (p @ q) (n+1))
        proof −
          have n ≥ 0 ∧ n < (length p) − 1 ⟹ ?thesis
            using assms pq nth-append path-def by (metis add-lessD1 less-diff-conv)
          moreover have n = (length p) − 1 ⟹ ?thesis
            using assms pq nth-append path-def Dom-hd Cod-last by simp
          moreover have n ≥ length p ∧ n < length (p @ q) − 1 ⟹ ?thesis
          proof −
```

```
        assume 1: n ≥ length p ∧ n < length (p @ q) − 1
        have Cod (nth (p @ q) n) = Cod (nth q (n − length p))
          using 1 nth-append leD by metis
        also have ... = Dom (nth q (n − length p + 1))
          using 1 assms(2) path-def by auto
        also have ... = Dom (nth (p @ q) (n + 1))
          using 1 nth-append
          by (metis Nat.add-diff-assoc2 ex-least-nat-le le-0-eq le-add1 le-neq-implies-less
                le-refl le-trans length-0-conv pq)
        finally show Cod (nth (p @ q) n) = Dom (nth (p @ q) (n + 1)) by auto
      qed
      ultimately show ?thesis using 1 by auto
    qed
  qed
  show ?thesis
    unfolding path-def using assms pq path-def hd-append2 Cod-last Dom-hd 1 2
    by simp
 qed
qed
ultimately show ?thesis by auto
qed

end
```

18.2 Free Categories

The free category generated by a graph has as its arrows all triples *MkArr x y p*, where x and y are objects and p is a path from x to y. We construct it here an instance of the general construction given by the *concrete-category* locale.

```
locale free-category =
  G: graph Obj Arr D C
for Obj :: 'obj set
and Arr :: 'arr set
and D :: 'arr ⇒ 'obj
and C :: 'arr ⇒ 'obj
begin

  type-synonym ('o, 'a) arr = ('o, 'a list) concrete-category.arr

  sublocale concrete-category ‹Obj :: 'obj set› ‹λx y. Collect (G.path x y)›
    ‹λ-. []› ‹λ- - - g f. f @ g›
    using G.path-Obj G.path-concat
    by (unfold-locales, simp-all)

  abbreviation comp      (infixr ‹·› 55)
  where comp ≡ COMP
  notation in-hom      (‹«- : - → -»›)
```

abbreviation *Path*
where *Path* ≡ *Map*

lemma *arr-single* [*simp*]:
assumes *x* ∈ *Arr*
shows *arr* (*MkArr* (*D x*) (*C x*) [*x*])
using *assms*
by (*simp add*: *G.cod-is-obj G.dom-is-obj G.path-single-Arr*)

end

18.3 Discrete Categories

A discrete category is a category in which every arrow is an identity. We could construct it as the free category generated by a graph with no arrows, but it is simpler just to apply the *concrete-category* construction directly.

locale *discrete-category* =
fixes *Obj* :: *'obj set*
begin

type-synonym *'o arr* = (*'o*, *unit*) *concrete-category.arr*

sublocale *concrete-category* ‹*Obj* :: *'obj set*› ‹λ*x y*. *if x* = *y then* {*x*} *else* {}›
‹λ*x*. *x*› ‹λ- - *x* - -. *x*›
apply *unfold-locales*
apply *simp-all*
apply (*metis empty-iff*)
by (*metis empty-iff singletonD*)

abbreviation *comp* (**infixr** ‹·› *55*)
where *comp* ≡ *COMP*
notation *in-hom* (‹«- : - → -»›)

lemma *is-discrete*:
shows *arr f* ⟷ *ide f*
using *ide-char*$_{CC}$ *arr-char* **by** *simp*

lemma *arr-char*:
shows *arr f* ⟷ *Dom f* ∈ *Obj* ∧ *f* = *MkIde* (*Dom f*)
using *is-discrete*
by (*metis* (*no-types*, *lifting*) *cod-char dom-char ide-MkIde ide-char*$_{CC}$ *ide-char'*)

lemma *arr-char'*:
shows *arr f* ⟷ *f* ∈ *MkIde* ‘ *Obj*
using *arr-char image-iff* **by** *auto*

lemma *dom-char*:
shows *dom f* = (*if arr f then f else null*)

using *dom-char is-discrete* **by** *simp*

lemma *cod-char*:
shows *cod f* = (*if arr f then f else null*)
using *cod-char is-discrete* **by** *simp*

lemma *in-hom-char*:
shows «*f* : *a* → *b*» ⟷ *arr f* ∧ *f* = *a* ∧ *f* = *b*
using *is-discrete* **by** *auto*

lemma *seq-char*:
shows *seq g f* ⟷ *arr f* ∧ *f* = *g*
using *is-discrete*
by (*metis* (*no-types*, *lifting*) *comp-arr-dom seqE dom-char*)

lemma *comp-char*:
shows *g* · *f* = (*if seq g f then f else null*)
proof −
have ¬ *seq g f* ⟹ *?thesis*
using *comp-char* **by** *presburger*
moreover have *seq g f* ⟹ *?thesis*
using *seq-char comp-char comp-arr-ide is-discrete*
by (*metis* (*no-types*, *lifting*))
ultimately show *?thesis* **by** *blast*
qed

end

The empty category is the discrete category generated by an empty set of objects.

locale *empty-category* =
discrete-category {} :: *unit set*
begin

lemma *is-empty*:
shows ¬*arr f*
using *arr-char* **by** *simp*

end

18.4 Quivers

A quiver is a two-object category whose non-identity arrows all point in the same direction. A quiver is specified by giving the set of these non-identity arrows.

locale *quiver* =
fixes *Arr* :: *'arr set*
begin

type-synonym *'a arr* = (*unit*, *'a*) *concrete-category.arr*

sublocale *free-category* {*False*, *True*} *Arr* λ-. *False* λ-. *True*
  by (*unfold-locales*, *simp-all*)

notation *comp* (**infixr** ‹·› *55*)
notation *in-hom* (‹«- : - → -»›)

definition *Zero*
where *Zero* ≡ *MkIde False*

definition *One*
where *One* ≡ *MkIde True*

definition *fromArr*
where *fromArr x* ≡ *if x* ∈ *Arr then MkArr False True* [*x*] *else null*

definition *toArr*
where *toArr f* ≡ *hd* (*Path f*)

lemma *ide-char*:
shows *ide f* ⟷ *f* = *Zero* ∨ *f* = *One*
proof −
  have *ide f* ⟷ *f* = *MkIde False* ∨ *f* = *MkIde True*
    using *ide-char*$_{CC}$ *concrete-category.MkIde-Dom′ concrete-category-axioms* **by** *fastforce*
  thus *?thesis*
    using *comp-def Zero-def One-def* **by** *simp*
qed

lemma *arr-char′*:
shows *arr f* ⟷ *f* =
      MkIde False ∨ *f* = *MkIde True* ∨ *f* ∈ (λ*x*. *MkArr False True* [*x*]) ‘ *Arr*
proof
  assume *f*: *f* = *MkIde False* ∨ *f* = *MkIde True* ∨ *f* ∈ (λ*x*. *MkArr False True* [*x*]) ‘ *Arr*
  show *arr f* **using** *f* **by** *auto*
  next
  assume *f*: *arr f*
  have ¬(*f* = *MkIde False* ∨ *f* = *MkIde True*) ⟹ *f* ∈ (λ*x*. *MkArr False True* [*x*]) ‘ *Arr*
  proof −
    assume *f′*: ¬(*f* = *MkIde False* ∨ *f* = *MkIde True*)
    have *0*: *Dom f* = *False* ∧ *Cod f* = *True*
      using *f f′ arr-char G.path-def MkArr-Map* **by** *fastforce*
    have *1*: *f* = *MkArr False True* (*Path f*)
      using *f 0 arr-char MkArr-Map* **by** *force*
    moreover have *length* (*Path f*) = *1*
    proof −
      have *length* (*Path f*) ≠ *0*
        using *f f′ 0 arr-char G.path-def* **by** *simp*
      moreover have ⋀*x y p*. *length p* > *1* ⟹ ¬ *G.path x y p*
        using *G.path-def less-diff-conv* **by** *fastforce*

```
    ultimately show ?thesis
      using f arr-char
      by (metis less-one linorder-neqE-nat mem-Collect-eq)
  qed
  moreover have ⋀p. length p = 1 ⟷ (∃x. p = [x])
    by (auto simp: length-Suc-conv)
  ultimately have ∃x. x ∈ Arr ∧ Path f = [x]
    using f G.path-def arr-char
    by (metis (no-types, lifting) Cod.simps(1) Dom.simps(1) le-eq-less-or-eq
        less-numeral-extra(1) mem-Collect-eq nth-Cons-0)
  thus f ∈ (λx. MkArr False True [x]) ‘ Arr
    using 1 by auto
 qed
 thus f = MkIde False ∨ f = MkIde True ∨ f ∈ (λx. MkArr False True [x]) ‘ Arr
  by auto
qed

lemma arr-char:
shows arr f ⟷ f = Zero ∨ f = One ∨ f ∈ fromArr ‘ Arr
 using arr-char′ Zero-def One-def fromArr-def by simp

lemma dom-char:
shows dom f = (if arr f then
                 if f = One then One else Zero
               else null)
proof −
 have ¬ arr f ⟹ ?thesis
  using dom-char by simp
 moreover have arr f ⟹ ?thesis
 proof −
  assume f: arr f
  have 1: dom f = MkIde (Dom f)
    using f dom-char by simp
  have f = One ⟹ ?thesis
    using f 1 One-def by (metis (full-types) Dom.simps(1))
  moreover have f = Zero ⟹ ?thesis
    using f 1 Zero-def by (metis (full-types) Dom.simps(1))
  moreover have f ∈ fromArr ‘ Arr ⟹ ?thesis
    using f fromArr-def G.path-def Zero-def calculation(1) by auto
  ultimately show ?thesis
    using f arr-char by blast
 qed
 ultimately show ?thesis by blast
qed

lemma cod-char:
shows cod f = (if arr f then
                 if f = Zero then Zero else One
               else null)
```

proof −
  have ¬ *arr f* ⟹ *?thesis*
    using *cod-char* **by** *simp*
  moreover have *arr f* ⟹ *?thesis*
  proof −
    assume *f*: *arr f*
    have *1*: *cod f* = *MkIde* (*Cod f*)
      using *f cod-char* **by** *simp*
    have *f* = *One* ⟹ *?thesis*
      using *f 1 One-def* **by** (*metis* (*full-types*) *Cod.simps*(*1*) *f*)
    moreover have *f* = *Zero* ⟹ *?thesis*
      using *f 1 Zero-def* **by** (*metis* (*full-types*) *Cod.simps*(*1*) *f*)
    moreover have *f* ∈ *fromArr* ‘ *Arr* ⟹ *?thesis*
      using *f fromArr-def G.path-def One-def calculation*(*2*) **by** *auto*
    ultimately show *?thesis*
      using *f arr-char* **by** *blast*
  qed
  ultimately show *?thesis* **by** *blast*
qed

lemma *seq-char*:
shows *seq g f* ⟷ *arr g* ∧ *arr f* ∧ ((*f* = *Zero* ∧ *g* ≠ *One*) ∨ (*f* ≠ *Zero* ∧ *g* = *One*))
proof
  assume *gf*: *arr g* ∧ *arr f* ∧ ((*f* = *Zero* ∧ *g* ≠ *One*) ∨ (*f* ≠ *Zero* ∧ *g* = *One*))
  show *seq g f*
    using *gf dom-char cod-char* **by** *auto*
  next
  assume *gf*: *seq g f*
  hence *1*: *arr f* ∧ *arr g* ∧ *dom g* = *cod f* **by** *auto*
  have *Cod f* = *False* ⟹ *f* = *Zero*
    using *gf 1 arr-char* [*of f*] *G.path-def Zero-def One-def cod-char Dom-cod*
    by (*metis* (*no-types*, *lifting*) *Dom.simps*(*1*))
  moreover have *Cod f* = *True* ⟹ *g* = *One*
    using *gf 1 arr-char* [*of f*] *G.path-def Zero-def One-def dom-char Dom-cod*
    by (*metis* (*no-types*, *lifting*) *Dom.simps*(*1*))
  moreover have ¬(*f* = *MkIde False* ∧ *g* = *MkIde True*)
    using *1* **by** *auto*
  ultimately show *arr g* ∧ *arr f* ∧ ((*f* = *Zero* ∧ *g* ≠ *One*) ∨ (*f* ≠ *Zero* ∧ *g* = *One*))
    using *gf arr-char One-def Zero-def* **by** *blast*
qed

lemma *not-ide-fromArr*:
shows ¬ *ide* (*fromArr x*)
  using *fromArr-def ide-char ide-def Zero-def One-def*
  by (*metis Cod.simps*(*1*) *Dom.simps*(*1*))

lemma *in-hom-char*:
shows «*f* : *a* → *b*» ⟷ (*a* = *Zero* ∧ *b* = *Zero* ∧ *f* = *Zero*) ∨
                    (*a* = *One* ∧ *b* = *One* ∧ *f* = *One*) ∨

$$(a = Zero \wedge b = One \wedge f \in fromArr\ `\ Arr)$$

proof −
  have $f = Zero \Longrightarrow ?thesis$
    using *arr-char'* [*of f*] *ide-char'*
    by (*metis* (*no-types*, *lifting*) *Zero-def category.in-homE category.in-homI*
        cod-MkArr dom-MkArr imageE is-category not-ide-fromArr)
  moreover have $f = One \Longrightarrow ?thesis$
    using *arr-char'* [*of f*] *ide-char'*
    by (*metis* (*no-types*, *lifting*) *One-def category.in-homE category.in-homI*
        cod-MkArr dom-MkArr image-iff is-category not-ide-fromArr)
  moreover have $f \in fromArr\ `\ Arr \Longrightarrow ?thesis$
  proof −
    assume *f*: $f \in fromArr\ `\ Arr$
    have *1*: *arr f* **using** *f arr-char* **by** *simp*
    moreover have $dom\ f = Zero \wedge cod\ f = One$
      using *f 1 arr-char dom-char cod-char fromArr-def*
      by (*metis* (*no-types*, *lifting*) *ide-char imageE not-ide-fromArr*)
    ultimately have *in-hom f Zero One* **by** *auto*
    thus $in\text{-}hom\ f\ a\ b \longleftrightarrow (a = Zero \wedge b = Zero \wedge f = Zero \vee$
                                $a = One \wedge b = One \wedge f = One \vee$
                                $a = Zero \wedge b = One \wedge f \in fromArr\ `\ Arr)$
      using *f ide-char* **by** *auto*
  qed
  ultimately show *?thesis*
    using *arr-char* [*of f*] **by** *fast*
qed

lemma *Zero-not-eq-One* [*simp*]:
shows $Zero \neq One$
  by (*simp add*: *One-def Zero-def*)

lemma *Zero-not-eq-fromArr* [*simp*]:
shows $Zero \notin fromArr\ `\ Arr$
  using *ide-char not-ide-fromArr*
  by (*metis* (*no-types*, *lifting*) *image-iff*)

lemma *One-not-eq-fromArr* [*simp*]:
shows $One \notin fromArr\ `\ Arr$
  using *ide-char not-ide-fromArr*
  by (*metis* (*no-types*, *lifting*) *image-iff*)

lemma *comp-char*:
shows $g \cdot f = (if\ seq\ g\ f\ then$
                 $if\ f = Zero\ then\ g\ else\ if\ g = One\ then\ f\ else\ null$
               $else\ null)$
proof −
  have $seq\ g\ f \Longrightarrow f = Zero \Longrightarrow g \cdot f = g$
    using *seq-char comp-char* [*of g f*] *Zero-def dom-char cod-char comp-arr-dom*
    by *auto*

```
  moreover have seq g f ⟹ g = One ⟹ g ⋅ f = f
    using seq-char comp-char [of g f] One-def dom-char cod-char comp-cod-arr
    by simp
  moreover have seq g f ⟹ f ≠ Zero ⟹ g ≠ One ⟹ g ⋅ f = null
    using seq-char Zero-def One-def by simp
  moreover have ¬seq g f ⟹ g ⋅ f = null
    using comp-char ext by fastforce
  ultimately show ?thesis by argo
qed

lemma comp-simp [simp]:
assumes seq g f
shows f = Zero ⟹ g ⋅ f = g
and g = One ⟹ g ⋅ f = f
  using assms seq-char comp-char by metis+

lemma arr-fromArr:
assumes x ∈ Arr
shows arr (fromArr x)
  using assms fromArr-def arr-char image-eqI by simp

lemma toArr-in-Arr:
assumes arr f and ¬ide f
shows toArr f ∈ Arr
proof −
  have ⋀a. a ∈ Arr ⟹ Path (fromArr a) = [a]
    using fromArr-def arr-char by simp
  hence hd (Path f) ∈ Arr
    using assms arr-char ide-char by auto
  thus ?thesis
    by (simp add: toArr-def)
qed

lemma toArr-fromArr [simp]:
assumes x ∈ Arr
shows toArr (fromArr x) = x
  using assms fromArr-def toArr-def
  by (simp add: toArr-def)

lemma fromArr-toArr [simp]:
assumes arr f and ¬ide f
shows fromArr (toArr f) = f
  using assms fromArr-def toArr-def arr-char ide-char toArr-fromArr by auto

end
```

18.5 Parallel Pairs

A parallel pair is a quiver with two non-identity arrows. It is important in the definition of equalizers.

locale *parallel-pair* =
  quiver {*False*, *True*} :: *bool set*
begin

  typedef *arr* = *UNIV* :: *bool quiver.arr set* **..**

  definition *j0*
  where *j0* ≡ *fromArr False*

  definition *j1*
  where *j1* ≡ *fromArr True*

  lemma *arr-char*:
  shows *arr f* ⟷ *f* = *Zero* ∨ *f* = *One* ∨ *f* = *j0* ∨ *f* = *j1*
    using *arr-char j0-def j1-def* **by** *simp*

  lemma *dom-char*:
  shows *dom f* = (*if f* = *j0* ∨ *f* = *j1 then Zero else if arr f then f else null*)
    using *arr-char dom-char j0-def j1-def*
    by (*metis ide-char not-ide-fromArr*)

  lemma *cod-char*:
  shows *cod f* = (*if f* = *j0* ∨ *f* = *j1 then One else if arr f then f else null*)
    using *arr-char cod-char j0-def j1-def*
    by (*metis ide-char not-ide-fromArr*)

  lemma *j0-not-eq-j1* [*simp*]:
  shows *j0* ≠ *j1*
    using *j0-def j1-def*
    by (*metis insert-iff toArr-fromArr*)

  lemma *Zero-not-eq-j0* [*simp*]:
  shows *Zero* ≠ *j0*
    using *Zero-def j0-def Zero-not-eq-fromArr* **by** *auto*

  lemma *Zero-not-eq-j1* [*simp*]:
  shows *Zero* ≠ *j1*
    using *Zero-def j1-def Zero-not-eq-fromArr* **by** *auto*

  lemma *One-not-eq-j0* [*simp*]:
  shows *One* ≠ *j0*
    using *One-def j0-def One-not-eq-fromArr* **by** *auto*

  lemma *One-not-eq-j1* [*simp*]:
  shows *One* ≠ *j1*

using *One-def j1-def One-not-eq-fromArr* **by** *auto*

lemma *dom-simp* [*simp*]:
shows *dom Zero* = *Zero*
and *dom One* = *One*
and *dom j0* = *Zero*
and *dom j1* = *Zero*
using *dom-char arr-char* **by** *auto*

lemma *cod-simp* [*simp*]:
shows *cod Zero* = *Zero*
and *cod One* = *One*
and *cod j0* = *One*
and *cod j1* = *One*
using *cod-char arr-char* **by** *auto*

end

end

Chapter 19

DiscreteCategory

theory *DiscreteCategory*
imports *Category*
begin

The locale defined here permits us to construct a discrete category having a specified set of objects, assuming that the set does not exhaust the elements of its type. In that case, we have the convenient situation that the arrows of the category can be directly identified with the elements of the given set, rather than having to pass between the two via tedious coercion maps. If it cannot be guaranteed that the given set is not the universal set at its type, then the more general discrete category construction defined (using coercions) in *FreeCategory* can be used.

locale *discrete-category* =
  fixes *Obj* :: *'a set*
  and *Null* :: *'a*
  assumes *Null-not-in-Obj*: *Null* $\notin$ *Obj*
begin

  definition *comp* :: *'a comp* (**infixr** ‹·› *55*)
  where $y \cdot x \equiv$ (*if* $x \in$ *Obj* $\wedge$ $x = y$ *then* x *else Null*)

  interpretation *partial-composition comp*
    apply *unfold-locales*
    using *comp-def* **by** *metis*

  lemma *null-char*:
  shows *null* = *Null*
    using *comp-def null-def* **by** *auto*

  lemma *ide-char* [*iff*]:
  shows *ide* $f \longleftrightarrow f \in$ *Obj*
    using *comp-def null-char ide-def Null-not-in-Obj* **by** *auto*

  lemma *domains-char*:
  shows *domains* $f = \{x.\ x \in$ *Obj* $\wedge\ x = f\}$

```
    unfolding domains-def
    using ide-char ide-def comp-def null-char by metis

  theorem is-category:
  shows category comp
    using comp-def
    apply unfold-locales
    using arr-def null-char self-domain-iff-ide ide-char
        apply fastforce
    using null-char self-codomain-iff-ide domains-char codomains-def ide-char
       apply fastforce
      apply (metis not-arr-null null-char)
     apply (metis not-arr-null null-char)
    by auto

end

sublocale discrete-category ⊆ category comp
  using is-category by auto

context discrete-category
begin

  lemma arr-char [iff]:
  shows arr f ⟷ f ∈ Obj
    using comp-def comp-cod-arr
   by (metis empty-iff has-codomain-iff-arr not-arr-null null-char self-codomain-iff-ide ide-char)

  lemma dom-char [simp]:
  shows dom f = (if f ∈ Obj then f else null)
    using arr-def dom-def arr-char ideD(2) by auto

  lemma cod-char [simp]:
  shows cod f = (if f ∈ Obj then f else null)
    using arr-def in-homE cod-def ideD(3) by auto

  lemma comp-char [simp]:
  shows comp g f = (if f ∈ Obj ∧ f = g then f else null)
    using comp-def null-char by auto

  lemma is-discrete:
  shows ide = arr
    using arr-char ide-char by auto

  lemma seq-char [iff]:
  shows seq f g ⟷ ide f ∧ f = g
    using is-discrete by (metis (full-types) ide-def seqE)

end
```

end

Chapter 20

Limit

theory *Limit*
imports *FreeCategory DiscreteCategory Adjunction*
begin

This theory defines the notion of limit in terms of diagrams and cones and relates it to the concept of a representation of a functor. The diagonal functor associated with a diagram shape J is defined and it is shown that a right adjoint to the diagonal functor gives limits of shape J and that a category has limits of shape J if and only if the diagonal functor is a left adjoint functor. Products and equalizers are defined as special cases of limits, and it is shown that a category with equalizers has limits of shape J if it has products indexed by the sets of objects and arrows of J. The existence of limits in a set category is investigated, and it is shown that every set category has equalizers and that a set category S has I-indexed products if and only if the universe of S "admits I-indexed tupling." The existence of limits in functor categories is also developed, showing that limits in functor categories are "determined pointwise" and that a functor category $[A, B]$ has limits of shape J if B does. Finally, it is shown that the Yoneda functor preserves limits.

This theory concerns itself only with limits; I have made no attempt to consider colimits. Although it would be possible to rework the entire development in dual form, it is possible that there is a more efficient way to dualize at least parts of it without repeating all the work. This is something that deserves further thought.

20.1 Representations of Functors

A representation of a contravariant functor F: $Cop \to S$, where S is a set category that is the target of a hom-functor for C, consists of an object a of C and a natural isomorphism $\Phi \in Y\ a \to F$, where Y: $C \to [Cop, S]$ is the Yoneda functor.

locale *representation-of-functor* =
  C: *category* C +
  Cop: *dual-category* C +
  S: *set-category* S *setp* +

F: *functor Cop.comp S F* +
Hom: *hom-functor C S setp φ* +
Ya: *yoneda-functor-fixed-object C S setp φ a* +
natural-isomorphism Cop.comp S ‹*Ya.Y a*› *F* Φ
for *C* :: *'c comp* (**infixr** ‹·› *55*)
and *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *setp* :: *'s set* ⇒ *bool*
and *φ* :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
and *F* :: *'c* ⇒ *'s*
and *a* :: *'c*
and Φ :: *'c* ⇒ *'s*
begin

abbreviation *Y* **where** *Y* ≡ *Ya.Y*
abbreviation *ψ* **where** *ψ* ≡ *Hom.ψ*

end

Two representations of the same functor are uniquely isomorphic.

locale *two-representations-one-functor* =
C: *category C* +
Cop: *dual-category C* +
S: *set-category S setp* +
F: *set-valued-functor Cop.comp S setp F* +
yoneda-functor C S setp φ +
Ya: *yoneda-functor-fixed-object C S setp φ a* +
Ya': *yoneda-functor-fixed-object C S setp φ a'* +
Φ: *representation-of-functor C S setp φ F a* Φ +
Φ': *representation-of-functor C S setp φ F a'* Φ'
for *C* :: *'c comp* (**infixr** ‹·› *55*)
and *S* :: *'s comp* (**infixr** ‹·$_S$› *55*)
and *setp* :: *'s set* ⇒ *bool*
and *F* :: *'c* ⇒ *'s*
and *φ* :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
and *a* :: *'c*
and Φ :: *'c* ⇒ *'s*
and *a'* :: *'c*
and Φ' :: *'c* ⇒ *'s*
begin

interpretation Ψ: *inverse-transformation Cop.comp S* ‹*Y a*› *F* Φ **..**
interpretation Ψ': *inverse-transformation Cop.comp S* ‹*Y a'*› *F* Φ' **..**
interpretation ΦΨ': *vertical-composite Cop.comp S* ‹*Y a*› *F* ‹*Y a'*› Φ Ψ'*.map* **..**
interpretation Φ'Ψ: *vertical-composite Cop.comp S* ‹*Y a'*› *F* ‹*Y a*› Φ' Ψ*.map* **..**

lemma *are-uniquely-isomorphic*:
shows ∃!*φ*. «*φ* : *a* → *a'*» ∧ *C.iso φ* ∧ *map φ* = *Cop-S.MkArr* (*Y a*) (*Y a'*) ΦΨ'*.map*
proof −
interpret ΦΨ': *natural-isomorphism Cop.comp S* ‹*Y a*› ‹*Y a'*› ΦΨ'*.map*

```
    using Φ.natural-isomorphism-axioms Ψ'.natural-isomorphism-axioms
        natural-isomorphisms-compose
    by blast
  interpret Φ'Ψ: natural-isomorphism Cop.comp S ‹Y a'› ‹Y a› Φ'Ψ.map
    using Φ'.natural-isomorphism-axioms Ψ.natural-isomorphism-axioms
        natural-isomorphisms-compose
    by blast
  interpret ΦΨ'-Φ'Ψ: inverse-transformations Cop.comp S ‹Y a› ‹Y a'› ΦΨ'.map Φ'Ψ.map
  proof
    fix x
    assume X: Cop.ide x
    show S.inverse-arrows (ΦΨ'.map x) (Φ'Ψ.map x)
      using S.inverse-arrows-compose S.inverse-arrows-sym X Φ'Ψ.map-simp-ide
          ΦΨ'.map-simp-ide Ψ'.inverts-components Ψ.inverts-components
      by force
  qed
  have Cop-S.inverse-arrows (Cop-S.MkArr (Y a) (Y a') ΦΨ'.map)
                            (Cop-S.MkArr (Y a') (Y a) Φ'Ψ.map)
  proof −
    have Ya: functor Cop.comp S (Y a) ..
    have Ya': functor Cop.comp S (Y a') ..
    have ΦΨ': natural-transformation Cop.comp S (Y a) (Y a') ΦΨ'.map ..
    have Φ'Ψ: natural-transformation Cop.comp S (Y a') (Y a) Φ'Ψ.map ..
    show ?thesis
      by (metis (no-types, lifting) Cop-S.arr-MkArr Cop-S.comp-MkArr Cop-S.ide-MkIde
          Cop-S.inverse-arrows-def Ya'.functor-axioms Ya.functor-axioms
          Φ'Ψ.natural-transformation-axioms ΦΨ'.natural-transformation-axioms
          ΦΨ'-Φ'Ψ.inverse-transformations-axioms inverse-transformations-inverse(1−2)
          mem-Collect-eq)
  qed
  hence 3: Cop-S.iso (Cop-S.MkArr (Y a) (Y a') ΦΨ'.map) using Cop-S.isoI by blast
  hence ∃f. «f : a → a'» ∧ map f = Cop-S.MkArr (Y a) (Y a') ΦΨ'.map
    using Ya.ide-a Ya'.ide-a is-full Y-def Cop-S.iso-is-arr full-functor.is-full
        Cop-S.MkArr-in-hom ΦΨ'.natural-transformation-axioms preserves-ide
    by force
  from this obtain φ
    where φ: «φ : a → a'» ∧ map φ = Cop-S.MkArr (Y a) (Y a') ΦΨ'.map
    by blast
  show ?thesis
    by (metis 3 C.in-homE φ is-faithful reflects-iso)
qed

end
```

20.2 Diagrams and Cones

A *diagram* in a category C is a functor $D\colon J \to C$. We refer to the category J as the diagram *shape*. Note that in the usual expositions of category theory that use set theory

as their foundations, the shape J of a diagram is required to be a "small" category, where smallness means that the collection of objects of J, as well as each of the "homs," is a set. However, in HOL there is no class of all sets, so it is not meaningful to speak of J as "small" in any kind of absolute sense. There is likely a meaningful notion of smallness of J *relative to* C (the result below that states that a set category has I-indexed products if and only if its universe "admits I-indexed tuples" is suggestive of how this might be defined), but I haven't fully explored this idea at present.

```
locale diagram =
  C: category C +
  J: category J +
  functor J C D
for J :: 'j comp      (infixr ‹·_J› 55)
and C :: 'c comp      (infixr ‹·› 55)
and D :: 'j ⇒ 'c
begin

  notation J.in-hom (‹«- : - →_J -»›)

end

lemma comp-diagram-functor:
assumes diagram J C D and functor J' J F
shows diagram J' C (D o F)
  by (meson assms(1) assms(2) diagram-def functor.axioms(1) functor-comp)
```

A *cone* over a diagram $D\colon J \to C$ is a natural transformation from a constant functor to D. The value of the constant functor is the *apex* of the cone.

```
locale cone =
  C: category C +
  J: category J +
  D: diagram J C D +
  A: constant-functor J C a +
  natural-transformation J C A.map D χ
for J :: 'j comp      (infixr ‹·_J› 55)
and C :: 'c comp      (infixr ‹·› 55)
and D :: 'j ⇒ 'c
and a :: 'c
and χ :: 'j ⇒ 'c
begin

  lemma ide-apex:
  shows C.ide a
    using A.value-is-ide by auto

  lemma component-in-hom:
  assumes J.arr j
  shows «χ j : a → D (J.cod j)»
    using assms by auto
```

```
  lemma cod-determines-component:
  assumes J.arr j
  shows χ j = χ (J.cod j)
    using assms naturality2 A.map-simp C.comp-arr-ide ide-apex preserves-reflects-arr
    by metis

end
```

A cone over diagram D is transformed into a cone over diagram $D \circ F$ by pre-composing with F.

```
lemma comp-cone-functor:
assumes cone J C D a χ and functor J' J F
shows cone J' C (D o F) a (χ o F)
proof −
  interpret χ: cone J C D a χ using assms(1) by auto
  interpret F: functor J' J F using assms(2) by auto
  interpret A': constant-functor J' C a
    using χ.A.value-is-ide by unfold-locales auto
  have 1: χ.A.map o F = A'.map
    using χ.A.map-def A'.map-def χ.J.not-arr-null by auto
  interpret χ': natural-transformation J' C A'.map ‹D o F› ‹χ o F›
    using 1 horizontal-composite F.as-nat-trans.natural-transformation-axioms
          χ.natural-transformation-axioms
    by fastforce
  show cone J' C (D o F) a (χ o F) ..
qed
```

A cone over diagram D can be transformed into a cone over a diagram D' by post-composing with a natural transformation from D to D'.

```
lemma vcomp-transformation-cone:
assumes cone J C D a χ
and natural-transformation J C D D' τ
shows cone J C D' a (vertical-composite.map J C χ τ)
 by (meson assms cone.axioms(4−5) cone.intro diagram.intro natural-transformation.axioms(1−4)
          vertical-composite.intro vertical-composite.is-natural-transformation)

context functor
begin

  lemma preserves-diagrams:
  fixes J :: 'j comp
  assumes diagram J A D
  shows diagram J B (F o D)
    by (meson assms diagram-def functor-axioms functor-comp functor-def)

  lemma preserves-cones:
  fixes J :: 'j comp
```

```
  assumes cone J A D a χ
  shows cone J B (F o D) (F a) (F o χ)
  proof −
    interpret χ: cone J A D a χ using assms by auto
    interpret Fa: constant-functor J B ‹F a›
      using χ.ide-apex by unfold-locales auto
    have 1: F o χ.A.map = Fa.map
    proof
      fix f
      show (F ∘ χ.A.map) f = Fa.map f
        using extensionality Fa.extensionality χ.A.extensionality
        by (cases χ.J.arr f, simp-all)
    qed
    interpret χ′: natural-transformation J B Fa.map ‹F o D› ‹F o χ›
      using 1 horizontal-composite χ.natural-transformation-axioms
            as-nat-trans.natural-transformation-axioms
      by fastforce
    show cone J B (F o D) (F a) (F o χ) ..
  qed

end

context diagram
begin

  abbreviation cone
  where cone a χ ≡ Limit.cone J C D a χ

  abbreviation cones :: 'c ⇒ ('j ⇒ 'c) set
  where cones a ≡ { χ. cone a χ }
```

An arrow $f \in C.hom\ a'\ a$ induces by composition a transformation from cones with apex a to cones with apex a'. This transformation is functorial in f.

```
  abbreviation cones-map :: 'c ⇒ ('j ⇒ 'c) ⇒ ('j ⇒ 'c)
  where cones-map f ≡ (λχ ∈ cones (C.cod f). λj. if J.arr j then χ j · f else C.null)

  lemma cones-map-mapsto:
  assumes C.arr f
  shows cones-map f ∈
          extensional (cones (C.cod f)) ∩ (cones (C.cod f) → cones (C.dom f))
  proof
    show cones-map f ∈ extensional (cones (C.cod f)) by blast
    show cones-map f ∈ cones (C.cod f) → cones (C.dom f)
    proof
      fix χ
      assume χ ∈ cones (C.cod f)
      hence χ: cone (C.cod f) χ by auto
      interpret χ: cone J C D ‹C.cod f› χ using χ by auto
      interpret B: constant-functor J C ‹C.dom f›
```

```
      using assms by unfold-locales auto
    have cone (C.dom f) (λj. if J.arr j then χ j · f else C.null)
      using assms B.value-is-ide χ.naturality1 χ.naturality2
      apply (unfold-locales, auto)
      using χ.naturality1
       apply (metis C.comp-assoc)
      using χ.naturality2 C.comp-arr-dom
      by (metis J.arr-cod-iff-arr J.cod-cod C.comp-assoc)
    thus (λj. if J.arr j then χ j · f else C.null) ∈ cones (C.dom f) by auto
  qed
qed

lemma cones-map-ide:
assumes χ ∈ cones a
shows cones-map a χ = χ
proof −
  interpret χ: cone J C D a χ using assms by auto
  show ?thesis
  proof
    fix j
    show cones-map a χ j = χ j
      using assms χ.A.value-is-ide χ.preserves-hom C.comp-arr-dom χ.extensionality
      by (cases J.arr j, auto)
  qed
qed

lemma cones-map-comp:
assumes C.seq f g
shows cones-map (f · g) = restrict (cones-map g o cones-map f) (cones (C.cod f))
proof (intro restr-eqI)
  show cones (C.cod (f · g)) = cones (C.cod f) using assms by simp
  show ⋀χ. χ ∈ cones (C.cod (f · g)) ⟹
            (λj. if J.arr j then χ j · f · g else C.null) = (cones-map g o cones-map f) χ
  proof −
    fix χ
    assume χ: χ ∈ cones (C.cod (f · g))
    show (λj. if J.arr j then χ j · f · g else C.null) = (cones-map g o cones-map f) χ
    proof −
      have ((cones-map g) o (cones-map f)) χ = cones-map g (cones-map f χ)
        by force
      also have ... = (λj. if J.arr j then
                           (λj. if J.arr j then χ j · f else C.null) j · g else C.null)
      proof
        fix j
        have cone (C.dom f) (cones-map f χ)
          using assms χ cones-map-mapsto by (elim C.seqE, force)
        thus cones-map g (cones-map f χ) j =
             (if J.arr j then C (if J.arr j then χ j · f else C.null) g else C.null)
          using χ assms by auto
```

```
      qed
      also have ... = (λj. if J.arr j then χ j · f · g else C.null)
        using C.comp-assoc by fastforce
      finally show ?thesis by auto
    qed
  qed
qed

end
```

Changing the apex of a cone by pre-composing with an arrow f commutes with changing the diagram of a cone by post-composing with a natural transformation.

```
lemma cones-map-vcomp:
assumes diagram J C D and diagram J C D'
and natural-transformation J C D D' τ
and cone J C D a χ
and f: partial-composition.in-hom C f a' a
shows diagram.cones-map J C D' f (vertical-composite.map J C χ τ)
        = vertical-composite.map J C (diagram.cones-map J C D f χ) τ
proof −
  interpret D: diagram J C D using assms(1) by auto
  interpret D': diagram J C D' using assms(2) by auto
  interpret τ: natural-transformation J C D D' τ using assms(3) by auto
  interpret χ: cone J C D a χ using assms(4) by auto
  interpret τoχ: vertical-composite J C χ.A.map D D' χ τ ..
  interpret τoχ: cone J C D' a τoχ.map ..
  interpret χf: cone J C D a' ‹D.cones-map f χ›
    using f χ.cone-axioms D.cones-map-mapsto by blast
  interpret τoχf: vertical-composite J C χf.A.map D D' ‹D.cones-map f χ› τ ..
  interpret τoχ-f: cone J C D' a' ‹D'.cones-map f τoχ.map›
    using f τoχ.cone-axioms D'.cones-map-mapsto [of f] by blast
  write C (infixr ‹·› 55)
  show D'.cones-map f τoχ.map = τoχf.map
  proof (intro natural-transformation-eqI)
    show natural-transformation J C χf.A.map D' (D'.cones-map f τoχ.map) ..
    show natural-transformation J C χf.A.map D' τoχf.map ..
    show ⋀j. D.J.ide j ⟹ D'.cones-map f τoχ.map j = τoχf.map j
    proof −
      fix j
      assume j: D.J.ide j
      have D'.cones-map f τoχ.map j = τoχ.map j · f
        using f τoχ.cone-axioms τoχ.map-simp-2 τoχ.extensionality by auto
      also have ... = (τ j · χ (D.J.dom j)) · f
        using j τoχ.map-simp-2 by simp
      also have ... = τ j · χ (D.J.dom j) · f
        using D.C.comp-assoc by simp
      also have ... = τoχf.map j
        using j f χ.cone-axioms τoχf.map-simp-2 by auto
      finally show D'.cones-map f τoχ.map j = τoχf.map j by auto
```

```
    qed
  qed
qed
```

Given a diagram D, we can construct a contravariant set-valued functor, which takes each object a of C to the set of cones over D with apex a, and takes each arrow f of C to the function on cones over D induced by pre-composition with f. For this, we need to introduce a set category S whose universe is large enough to contain all the cones over D, and we need to have an explicit correspondence between cones and elements of the universe of S. A replete set category S equipped with an injective mapping $\iota::('j \Rightarrow 'c) \Rightarrow 's$ serves this purpose.

```
locale cones-functor =
  C: category C +
  Cop: dual-category C +
  J: category J +
  D: diagram J C D +
  S: replete-concrete-set-category S UNIV ι
for J :: 'j comp      (infixr ‹·J› 55)
and C :: 'c comp      (infixr ‹·› 55)
and D :: 'j ⇒ 'c
and S :: 's comp      (infixr ‹·S› 55)
and ι :: ('j ⇒ 'c) ⇒ 's
begin

  notation S.in-hom     (‹«- : - →S -»›)

  abbreviation o where o ≡ S.DN

  definition map :: 'c ⇒ 's
  where map = (λf. if C.arr f then
                 S.mkArr (ι ‘ D.cones (C.cod f)) (ι ‘ D.cones (C.dom f))
                         (ι o D.cones-map f o o)
               else S.null)

  lemma map-simp [simp]:
  assumes C.arr f
  shows map f = S.mkArr (ι ‘ D.cones (C.cod f)) (ι ‘ D.cones (C.dom f))
                        (ι o D.cones-map f o o)
    using assms map-def by auto

  lemma arr-map:
  assumes C.arr f
  shows S.arr (map f)
  proof −
    have ι o D.cones-map f o o ∈ ι ‘ D.cones (C.cod f) → ι ‘ D.cones (C.dom f)
      using assms D.cones-map-mapsto by force
    thus ?thesis using assms S.UP-mapsto by auto
  qed
```

lemma *map-ide*:
assumes *C.ide a*
shows *map a = S.mkIde (ι ‘ D.cones a)*
proof −
  have *map a = S.mkArr (ι ‘ D.cones a) (ι ‘ D.cones a) (ι o D.cones-map a o o)*
    using *assms map-simp* **by** *force*
  also have *... = S.mkArr (ι ‘ D.cones a) (ι ‘ D.cones a) (λx. x)*
    using *S.UP-mapsto D.cones-map-ide* **by** *force*
  also have *... = S.mkIde (ι ‘ D.cones a)*
    using *assms S.mkIde-as-mkArr S.UP-mapsto* **by** *blast*
  finally show *?thesis* **by** *auto*
qed

lemma *map-preserves-dom*:
assumes *Cop.arr f*
shows *map (Cop.dom f) = S.dom (map f)*
  using *assms arr-map map-ide* **by** *auto*

lemma *map-preserves-cod*:
assumes *Cop.arr f*
shows *map (Cop.cod f) = S.cod (map f)*
  using *assms arr-map map-ide* **by** *auto*

lemma *map-preserves-comp*:
assumes *Cop.seq g f*
shows *map (g ·^op f) = map g ·_S map f*
proof −
  have *map (g ·^op f) = S.mkArr (ι ‘ D.cones (C.cod f)) (ι ‘ D.cones (C.dom g))*
                          ((ι o D.cones-map g o o) o (ι o D.cones-map f o o))
  proof −
    have *1*: *S.arr (map (g ·^op f))*
      using *assms arr-map [of C f g]* **by** *simp*
    have *map (g ·^op f) = S.mkArr (ι ‘ D.cones (C.cod f)) (ι ‘ D.cones (C.dom g))*
                            (ι o D.cones-map (C f g) o o)
      using *assms map-simp [of C f g]* **by** *simp*
    also have *... = S.mkArr (ι ‘ D.cones (C.cod f)) (ι ‘ D.cones (C.dom g))*
                          ((ι o D.cones-map g o o) o (ι o D.cones-map f o o))
      using *assms 1 calculation D.cones-map-mapsto D.cones-map-comp* **by** *auto*
    finally show *?thesis* **by** *blast*
  qed
  also have *... = map g ·_S map f*
    using *assms arr-map [of f] arr-map [of g] map-simp S.comp-mkArr* **by** *auto*
  finally show *?thesis* **by** *auto*
qed

lemma *is-functor*:
shows *functor Cop.comp S map*
  apply (*unfold-locales*)
  using *map-def arr-map map-preserves-dom map-preserves-cod map-preserves-comp*

by *auto*

end

sublocale *cones-functor* ⊆ *functor Cop.comp S map* **using** *is-functor* **by** *auto*
sublocale *cones-functor* ⊆ *set-valued-functor Cop.comp S* ‹λ*A*. *A* ⊆ *S.Univ*› *map* **..**

20.3 Limits

20.3.1 Limit Cones

A *limit cone* for a diagram D is a cone χ over D with the universal property that any other cone χ' over the diagram D factors uniquely through χ.

locale *limit-cone* =
  C: *category C* +
  J: *category J* +
  D: *diagram J C D* +
  cone J C D a χ
for *J* :: *'j comp* (**infixr** ‹·$_J$› *55*)
and *C* :: *'c comp* (**infixr** ‹·› *55*)
and *D* :: *'j* ⇒ *'c*
and *a* :: *'c*
and χ :: *'j* ⇒ *'c* +
assumes *is-universal*: *cone J C D a'* χ' ⟹ ∃!*f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ'
begin

  definition *induced-arrow* :: *'c* ⇒ (*'j* ⇒ *'c*) ⇒ *'c*
  where *induced-arrow a'* χ' = (*THE f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ')

  lemma *induced-arrowI*:
  assumes χ': χ' ∈ *D.cones a'*
  shows «*induced-arrow a'* χ' : *a'* → *a*»
  and *D.cones-map* (*induced-arrow a'* χ') χ = χ'
  proof −
    have ∃!*f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ'
      using *assms* χ' *is-universal* **by** *simp*
    hence *1*: «*induced-arrow a'* χ' : *a'* → *a*» ∧ *D.cones-map* (*induced-arrow a'* χ') χ = χ'
      using *theI'* [*of* λ*f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ'] *induced-arrow-def*
      by *presburger*
    show «*induced-arrow a'* χ' : *a'* → *a*» **using** *1* **by** *simp*
    show *D.cones-map* (*induced-arrow a'* χ') χ = χ' **using** *1* **by** *simp*
  qed

  lemma *cones-map-induced-arrow*:
  shows *induced-arrow a'* ∈ *D.cones a'* → *C.hom a' a*
  and ⋀χ'. χ' ∈ *D.cones a'* ⟹ *D.cones-map* (*induced-arrow a'* χ') χ = χ'
    using *induced-arrowI* **by** *auto*

lemma *induced-arrow-cones-map*:
assumes $C.ide\ a'$
shows $(\lambda f.\ D.cones\text{-}map\ f\ \chi) \in C.hom\ a'\ a \to D.cones\ a'$
and $\bigwedge f.\ \langle\langle f : a' \to a \rangle\rangle \Longrightarrow induced\text{-}arrow\ a'\ (D.cones\text{-}map\ f\ \chi) = f$
proof −
  have *a'*: $C.ide\ a'$ **using** *assms* **by** (*simp add*: *cone.ide-apex*)
  have *cone-χ*: *cone J C D a χ* **..**
  show $(\lambda f.\ D.cones\text{-}map\ f\ \chi) \in C.hom\ a'\ a \to D.cones\ a'$
    using *cone-χ D.cones-map-mapsto* **by** *blast*
  fix *f*
  assume *f*: $\langle\langle f : a' \to a \rangle\rangle$
  show $induced\text{-}arrow\ a'\ (D.cones\text{-}map\ f\ \chi) = f$
  proof −
    have $D.cones\text{-}map\ f\ \chi \in D.cones\ a'$
      using *f cone-χ D.cones-map-mapsto* **by** *blast*
    hence $\exists!f'.\ \langle\langle f' : a' \to a \rangle\rangle \wedge D.cones\text{-}map\ f'\ \chi = D.cones\text{-}map\ f\ \chi$
      using *assms is-universal* **by** *auto*
    thus *?thesis*
      using *f induced-arrow-def*
          the1-equality [*of* $\lambda f'.\ \langle\langle f' : a' \to a \rangle\rangle \wedge D.cones\text{-}map\ f'\ \chi = D.cones\text{-}map\ f\ \chi$]
      by *presburger*
  qed
qed

For a limit cone χ with apex a, for each object a' the hom-set $C.hom\ a'\ a$ is in bijective correspondence with the set of cones with apex a'.

lemma *bij-betw-hom-and-cones*:
assumes $C.ide\ a'$
shows $bij\text{-}betw\ (\lambda f.\ D.cones\text{-}map\ f\ \chi)\ (C.hom\ a'\ a)\ (D.cones\ a')$
proof (*intro bij-betwI*)
  show $(\lambda f.\ D.cones\text{-}map\ f\ \chi) \in C.hom\ a'\ a \to D.cones\ a'$
    using *assms induced-arrow-cones-map* **by** *blast*
  show $induced\text{-}arrow\ a' \in D.cones\ a' \to C.hom\ a'\ a$
    using *assms cones-map-induced-arrow* **by** *blast*
  show $\bigwedge f.\ f \in C.hom\ a'\ a \Longrightarrow induced\text{-}arrow\ a'\ (D.cones\text{-}map\ f\ \chi) = f$
    using *assms induced-arrow-cones-map* **by** *blast*
  show $\bigwedge \chi'.\ \chi' \in D.cones\ a' \Longrightarrow D.cones\text{-}map\ (induced\text{-}arrow\ a'\ \chi')\ \chi = \chi'$
    using *assms cones-map-induced-arrow* **by** *blast*
qed

lemma *induced-arrow-eqI*:
assumes $D.cone\ a'\ \chi'$ **and** $\langle\langle f : a' \to a \rangle\rangle$ **and** $D.cones\text{-}map\ f\ \chi = \chi'$
shows $induced\text{-}arrow\ a'\ \chi' = f$
  using *assms is-universal induced-arrow-def*
      the1-equality [*of* $\lambda f.\ f \in C.hom\ a'\ a \wedge D.cones\text{-}map\ f\ \chi = \chi'\ f$]
  by *simp*

lemma *induced-arrow-self*:
shows $induced\text{-}arrow\ a\ \chi = a$

```
  proof −
    have «a : a → a» ∧ D.cones-map a χ = χ
      using ide-apex cone-axioms D.cones-map-ide by force
    thus ?thesis using induced-arrow-eqI cone-axioms by auto
  qed

end

context diagram
begin

  abbreviation limit-cone
  where limit-cone a χ ≡ Limit.limit-cone J C D a χ
```

A diagram D has object a as a limit if a is the apex of some limit cone over D.

```
  abbreviation has-as-limit :: 'c ⇒ bool
  where has-as-limit a ≡ (∃χ. limit-cone a χ)

  abbreviation has-limit
  where has-limit ≡ (∃a χ. limit-cone a χ)

  definition some-limit :: 'c
  where some-limit = (SOME a. ∃χ. limit-cone a χ)

  definition some-limit-cone :: 'j ⇒ 'c
  where some-limit-cone = (SOME χ. limit-cone some-limit χ)

  lemma limit-cone-some-limit-cone:
  assumes has-limit
  shows limit-cone some-limit some-limit-cone
  proof −
    have ∃a. has-as-limit a using assms by simp
    hence has-as-limit some-limit
      using some-limit-def someI-ex [of λa. ∃χ. limit-cone a χ] by simp
    thus limit-cone some-limit some-limit-cone
      using assms some-limit-cone-def someI-ex [of λχ. limit-cone some-limit χ]
      by simp
  qed

  lemma ex-limitE:
  assumes ∃a. has-as-limit a
  obtains a χ where limit-cone a χ
    using assms someI-ex by blast

end
```

20.3.2 Limits by Representation

A limit for a diagram D can also be given by a representation (a, Φ) of the cones functor.

locale *representation-of-cones-functor* =
  C: *category C* +
  Cop: *dual-category C* +
  J: *category J* +
  D: *diagram J C D* +
  S: *replete-concrete-set-category S UNIV ι* +
  Cones: *cones-functor J C D S ι* +
  Hom: *hom-functor C S ‹λA. A ⊆ S.Univ› φ* +
  representation-of-functor C S S.setp φ Cones.map a Φ
for *J* :: *'j comp* (**infixr** *‹·J›* *55*)
and *C* :: *'c comp* (**infixr** *‹·›* *55*)
and *D* :: *'j ⇒ 'c*
and *S* :: *'s comp* (**infixr** *‹·S›* *55*)
and *φ* :: *'c ∗ 'c ⇒ 'c ⇒ 's*
and *ι* :: *('j ⇒ 'c) ⇒ 's*
and *a* :: *'c*
and Φ :: *'c ⇒ 's*

20.3.3 Putting it all Together

A "limit situation" combines and connects the ways of presenting a limit.

locale *limit-situation* =
  C: *category C* +
  Cop: *dual-category C* +
  J: *category J* +
  D: *diagram J C D* +
  S: *replete-concrete-set-category S UNIV ι* +
  Cones: *cones-functor J C D S ι* +
  Hom: *hom-functor C S S.setp φ* +
  Φ: *representation-of-functor C S S.setp φ Cones.map a* Φ +
  χ: *limit-cone J C D a χ*
for *J* :: *'j comp* (**infixr** *‹·J›* *55*)
and *C* :: *'c comp* (**infixr** *‹·›* *55*)
and *D* :: *'j ⇒ 'c*
and *S* :: *'s comp* (**infixr** *‹·S›* *55*)
and *φ* :: *'c ∗ 'c ⇒ 'c ⇒ 's*
and *ι* :: *('j ⇒ 'c) ⇒ 's*
and *a* :: *'c*
and Φ :: *'c ⇒ 's*
and *χ* :: *'j ⇒ 'c* +
assumes *χ-in-terms-of-*Φ: *χ = S.DN (S.Fun* (Φ *a*) *(φ (a, a) a))*
and Φ*-in-terms-of-χ*:
  Cop.ide a′ ⟹ Φ *a′ = S.mkArr (Hom.set (a′, a)) (ι ‘ D.cones a′)*
    (λx. ι (D.cones-map (Hom.ψ (a′, a) x) χ))

The assumption *χ-in-terms-of-*Φ states that the universal cone *χ* is obtained by applying the function *S.Fun* (Φ *a*) to the identity *a* of *C* (after taking into account the necessary coercions).

The assumption Φ*-in-terms-of-χ* states that the component of Φ at *a′* is the arrow

of S corresponding to the function that takes an arrow $f \in C.hom\ a'\ a$ and produces the cone with vertex a' obtained by transforming the universal cone χ by f.

20.3.4 Limit Cones Induce Limit Situations

To obtain a limit situation from a limit cone, we need to introduce a set category that is large enough to contain the hom-sets of C as well as the cones over D. We use the category of all $'c + ('j \Rightarrow 'c)$-sets for this.

context *limit-cone*
begin

interpretation *Cop*: *dual-category C* **..**
interpretation *CopxC*: *product-category Cop.comp C* **..**
interpretation *S*: *replete-setcat* ‹*TYPE*('*c* + ('*j* ⇒ '*c*))› **.**

notation *S.comp* (**infixr** ‹·$_S$› *55*)

interpretation *Sr*: *replete-concrete-set-category S.comp UNIV* ‹*S.UP o Inr*›
apply *unfold-locales*
using *S.UP-mapsto*
apply *auto*[*1*]
using *S.inj-UP inj-Inr inj-compose*
by *metis*

interpretation *Cones*: *cones-functor J C D S.comp* ‹*S.UP o Inr*› **..**

interpretation *Hom*: *hom-functor C S.comp S.setp* ‹λ-. *S.UP o Inl*›
apply (*unfold-locales*)
using *S.UP-mapsto*
apply *auto*[*1*]
using *S.inj-UP injD inj-onI inj-Inl inj-compose*
apply (*metis* (*no-types*, *lifting*))
using *S.UP-mapsto*
by *auto*

interpretation *Y*: *yoneda-functor C S.comp S.setp* ‹λ-. *S.UP o Inl*› **..**
interpretation *Ya*: *yoneda-functor-fixed-object C S.comp S.setp* ‹λ-. *S.UP o Inl*› *a*
apply (*unfold-locales*) **using** *ide-apex* **by** *auto*

abbreviation *inl* :: '*c* ⇒ '*c* + ('*j* ⇒ '*c*) **where** *inl* ≡ *Inl*
abbreviation *inr* :: ('*j* ⇒ '*c*) ⇒ '*c* + ('*j* ⇒ '*c*) **where** *inr* ≡ *Inr*
abbreviation ι **where** ι ≡ *S.UP o inr*
abbreviation o **where** o ≡ *Cones*.o
abbreviation φ **where** φ ≡ λ-. *S.UP o inl*
abbreviation ψ **where** ψ ≡ *Hom*.ψ
abbreviation *Y* **where** *Y* ≡ *Y.Y*

lemma *Ya-ide*:

assumes a': $C.ide\ a'$
shows $Y\ a\ a' = S.mkIde\ (Hom.set\ (a',\ a))$
using *assms ide-apex Y.Y-simp Hom.map-ide* **by** *simp*

lemma *Ya-arr*:
assumes g: $C.arr\ g$
shows $Y\ a\ g = S.mkArr\ (Hom.set\ (C.cod\ g,\ a))\ (Hom.set\ (C.dom\ g,\ a))$
$(\varphi\ (C.dom\ g,\ a)\ o\ Cop.comp\ g\ o\ \psi\ (C.cod\ g,\ a))$
using *ide-apex g Y.Y-ide-arr* [*of a g C.dom g C.cod g*] **by** *auto*

lemma *is-cone* [*simp*]:
shows $\chi \in D.cones\ a$
using *cone-axioms* **by** *simp*

For each object a' of C we have a function mapping $C.hom\ a'\ a$ to the set of cones over D with apex a', which takes $f \in C.hom\ a'\ a$ to χf, where χf is the cone obtained by composing χ with f (after accounting for coercions to and from the universe of S). The corresponding arrows of S are the components of a natural isomorphism from $Y\ a$ to *Cones*.

definition $\Phi o :: 'c \Rightarrow ('c + ('j \Rightarrow 'c))\ setcat.arr$
where
$\Phi o\ a' = S.mkArr\ (Hom.set\ (a',\ a))\ (\iota\ `\ D.cones\ a')\ (\lambda x.\ \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi))$

lemma Φo*-in-hom*:
assumes a': $C.ide\ a'$
shows «$\Phi o\ a' : S.mkIde\ (Hom.set\ (a',\ a)) \rightarrow_S S.mkIde\ (\iota\ `\ D.cones\ a')$»
proof −
have «$S.mkArr\ (Hom.set\ (a',\ a))\ (\iota\ `\ D.cones\ a')\ (\lambda x.\ \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)) :$
$S.mkIde\ (Hom.set\ (a',\ a)) \rightarrow_S S.mkIde\ (\iota\ `\ D.cones\ a')$»
proof −
have $(\lambda x.\ \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)) \in Hom.set\ (a',\ a) \rightarrow \iota\ `\ D.cones\ a'$
proof
fix x
assume x: $x \in Hom.set\ (a',\ a)$
hence «$\psi\ (a',\ a)\ x : a' \rightarrow a$»
using *ide-apex a' Hom.ψ-mapsto* **by** *auto*
hence $D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi \in D.cones\ a'$
using *ide-apex a' x D.cones-map-mapsto is-cone* **by** *force*
thus $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi) \in \iota\ `\ D.cones\ a'$ **by** *simp*
qed
moreover have $Hom.set\ (a',\ a) \subseteq S.Univ$
using *ide-apex a' Hom.set-subset-Univ* **by** *auto*
moreover have $\iota\ `\ D.cones\ a' \subseteq S.Univ$
using *S.UP-mapsto* **by** *auto*
ultimately show *?thesis* **using** *S.mkArr-in-hom* **by** *simp*
qed
thus *?thesis* **using** Φo*-def* [*of a'*] **by** *auto*
qed

```
interpretation Φ: transformation-by-components Cop.comp S.comp ‹Y a› Cones.map Φo
proof
  fix a'
  assume A': Cop.ide a'
  show «Φo a' : Y a a' →S Cones.map a'»
    using A' Ya-ide Φo-in-hom Cones.map-ide by auto
  next
  fix g
  assume g: Cop.arr g
  show Φo (Cop.cod g) ·S Y a g = Cones.map g ·S Φo (Cop.dom g)
  proof −
    let ?A = Hom.set (C.cod g, a)
    let ?B = Hom.set (C.dom g, a)
    let ?B' = ι ‘ D.cones (C.cod g)
    let ?C = ι ‘ D.cones (C.dom g)
    let ?F = φ (C.dom g, a) o Cop.comp g o ψ (C.cod g, a)
    let ?F' = ι o D.cones-map g o o
    let ?G = λx. ι (D.cones-map (ψ (C.dom g, a) x) χ)
    let ?G' = λx. ι (D.cones-map (ψ (C.cod g, a) x) χ)
    have S.arr (Y a g) ∧ Y a g = S.mkArr ?A ?B ?F
      using ide-apex g Ya.preserves-arr Ya-arr by fastforce
    moreover have S.arr (Φo (Cop.cod g))
      using g Φo-in-hom [of Cop.cod g] by auto
    moreover have Φo (Cop.cod g) = S.mkArr ?B ?C ?G
      using g Φo-def [of C.dom g] by auto
    moreover have S.seq (Φo (Cop.cod g)) (Y a g)
      using ide-apex g Φo-in-hom [of Cop.cod g] by auto
    ultimately have 1: S.seq (Φo (Cop.cod g)) (Y a g) ∧
                       Φo (Cop.cod g) ·S Y a g = S.mkArr ?A ?C (?G o ?F)
      using S.comp-mkArr [of ?A ?B ?F ?C ?G] by argo

    have Cones.map g = S.mkArr (ι ‘ D.cones (C.cod g)) (ι ‘ D.cones (C.dom g)) ?F'
      using g Cones.map-simp by fastforce
    moreover have Φo (Cop.dom g) = S.mkArr ?A ?B' ?G'
      using g Φo-def by fastforce
    moreover have S.seq (Cones.map g) (Φo (Cop.dom g))
      using g Cones.preserves-hom [of g C.cod g C.dom g] Φo-in-hom [of Cop.dom g]
      by force
    ultimately have
      2: S.seq (Cones.map g) (Φo (Cop.dom g)) ∧
         Cones.map g ·S Φo (Cop.dom g) = S.mkArr ?A ?C (?F' o ?G')
      using S.seqI' [of Φo (Cop.dom g) Cones.map g] S.comp-mkArr by auto

    have Φo (Cop.cod g) ·S Y a g = S.mkArr ?A ?C (?G o ?F)
      using 1 by auto
    also have ... = S.mkArr ?A ?C (?F' o ?G')
    proof (intro S.mkArr-eqI')
      show S.arr (S.mkArr ?A ?C (?G o ?F)) using 1 by force
      show ⋀x. x ∈ ?A ⟹ (?G o ?F) x = (?F' o ?G') x
```

```
      proof −
        fix x
        assume x: x ∈ ?A
        hence 1: «ψ (C.cod g, a) x : C.cod g → a»
          using ide-apex g Hom.ψ-mapsto [of C.cod g a] by auto
        have (?G o ?F) x = ι (D.cones-map (ψ (C.dom g, a)
                                (φ (C.dom g, a) (ψ (C.cod g, a) x · g))) χ)
        proof −
          have (?G o ?F) x = ?G (?F x) by simp
          also have ... = ι (D.cones-map (ψ (C.dom g, a)
                                  (φ (C.dom g, a) (ψ (C.cod g, a) x · g))) χ)
            by (metis Cop.comp-def comp-apply)
          finally show ?thesis by auto
        qed
        also have ... = ι (D.cones-map (ψ (C.cod g, a) x · g) χ)
        proof −
          have «ψ (C.cod g, a) x · g : C.dom g → a» using g 1 by auto
          thus ?thesis using Hom.ψ-φ by presburger
        qed
        also have ... = ι (D.cones-map g (D.cones-map (ψ (C.cod g, a) x) χ))
          using g x 1 is-cone D.cones-map-comp [of ψ (C.cod g, a) x g] by fastforce
        also have ... = ι (D.cones-map g (o (ι (D.cones-map (ψ (C.cod g, a) x) χ))))
          using 1 is-cone D.cones-map-mapsto Sr.DN-UP by auto
        also have ... = (?F' o ?G') x by simp
        finally show (?G o ?F) x = (?F' o ?G') x by auto
      qed
    qed
    also have ... = Cones.map g ·S Φo (Cop.dom g)
      using 2 by auto
    finally show ?thesis by auto
  qed
qed

interpretation Φ: set-valued-transformation Cop.comp S.comp S.setp
                  ‹Y a› Cones.map Φ.map ..

interpretation Φ: natural-isomorphism Cop.comp S.comp ‹Y a› Cones.map Φ.map
proof
  fix a'
  assume a': Cop.ide a'
  show S.iso (Φ.map a')
  proof −
    let ?F = λx. ι (D.cones-map (ψ (a', a) x) χ)
    have bij: bij-betw ?F (Hom.set (a', a)) (ι ‘ D.cones a')
    proof −
      have ⋀x x'. ⟦ x ∈ Hom.set (a', a); x' ∈ Hom.set (a', a);
                    ι (D.cones-map (ψ (a', a) x) χ) = ι (D.cones-map (ψ (a', a) x') χ) ⟧
                      ⟹ x = x'
      proof −
```

fix $x\ x'$
assume x: $x \in Hom.set\ (a',\ a)$ **and** x': $x' \in Hom.set\ (a',\ a)$
and xx': $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x')\ \chi)$
have ψx: «$\psi\ (a',\ a)\ x : a' \to a$» **using** x *ide-apex* a' *Hom.*ψ*-mapsto* **by** *auto*
have $\psi x'$: «$\psi\ (a',\ a)\ x' : a' \to a$» **using** x' *ide-apex* a' *Hom.*ψ*-mapsto* **by** *auto*
have *1*: $\exists!f.$ «$f : a' \to a$» $\wedge\ \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
proof −
  have $D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi \in D.cones\ a'$
    using $\psi x\ a'$ *is-cone D.cones-map-mapsto* **by** *force*
  hence *2*: $\exists!f.$ «$f : a' \to a$» $\wedge\ D.cones\text{-}map\ f\ \chi = D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi$
    using a' *is-universal* **by** *simp*
  show $\exists!f.$ «$f : a' \to a$» $\wedge\ \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
  proof −
    have $\bigwedge f.\ \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
                $\longleftrightarrow D.cones\text{-}map\ f\ \chi = D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi$
    proof −
      fix $f :: 'c$
      have $D.cones\text{-}map\ f\ \chi = D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi$
              $\longrightarrow \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
        by *simp*
      thus $(\iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi))$
                $= (D.cones\text{-}map\ f\ \chi = D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
        by (*meson Sr.inj-UP injD*)
    qed
    thus *?thesis* **using** *2* **by** *auto*
  qed
qed
have *2*: $\exists!x''.\ x'' \in Hom.set\ (a',\ a)\ \wedge$
              $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x'')\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
proof −
  from *1* **obtain** f'' **where**
    f'': «$f'' : a' \to a$» $\wedge\ \iota\ (D.cones\text{-}map\ f''\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
    by *blast*
  have $\varphi\ (a',\ a)\ f'' \in Hom.set\ (a',\ a)\ \wedge$
      $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ (\varphi\ (a',\ a)\ f''))\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
  proof
    show $\varphi\ (a',\ a)\ f'' \in Hom.set\ (a',\ a)$ **using** f'' *Hom.set-def* **by** *auto*
    show $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ (\varphi\ (a',\ a)\ f''))\ \chi) =$
          $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
      using f'' *Hom.*ψ*-*φ **by** *presburger*
  qed
  moreover have
    $\bigwedge x''.\ x'' \in Hom.set\ (a',\ a)\ \wedge$
        $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x'')\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
          $\Longrightarrow x'' = \varphi\ (a',\ a)\ f''$
  proof −
    fix x''
    assume x'': $x'' \in Hom.set\ (a',\ a)\ \wedge$
              $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x'')\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$

```
    hence «ψ (a′, a) x″ : a′ → a» ∧
          ι (D.cones-map (ψ (a′, a) x″) χ) = ι (D.cones-map (ψ (a′, a) x) χ)
      using ide-apex a′ Hom.set-def Hom.ψ-mapsto [of a′ a] by auto
    hence φ (a′, a) (ψ (a′, a) x″) = φ (a′, a) f″
      using 1 f″ by auto
    thus x″ = φ (a′, a) f″
      using ide-apex a′ x″ Hom.φ-ψ by simp
  qed
  ultimately show ?thesis
    using ex1I [of λx′. x′ ∈ Hom.set (a′, a) ∧
                        ι (D.cones-map (ψ (a′, a) x′) χ) =
                          ι (D.cones-map (ψ (a′, a) x) χ)
                    φ (a′, a) f″]
    by simp
 qed
 thus x = x′ using x x′ xx′ by auto
qed
hence inj-on ?F (Hom.set (a′, a))
 using inj-onI [of Hom.set (a′, a) ?F] by auto
moreover have ?F ‘ Hom.set (a′, a) = ι ‘ D.cones a′
proof
 show ?F ‘ Hom.set (a′, a) ⊆ ι ‘ D.cones a′
 proof
  fix X′
  assume X′: X′ ∈ ?F ‘ Hom.set (a′, a)
  from this obtain x′ where x′: x′ ∈ Hom.set (a′, a) ∧ ?F x′ = X′ by blast
  show X′ ∈ ι ‘ D.cones a′
  proof −
   have X′ = ι (D.cones-map (ψ (a′, a) x′) χ) using x′ by blast
   hence X′ = ι (D.cones-map (ψ (a′, a) x′) χ) using x′ by force
   moreover have «ψ (a′, a) x′ : a′ → a»
     using ide-apex a′ x′ Hom.set-def Hom.ψ-φ by auto
   ultimately show ?thesis
     using x′ is-cone D.cones-map-mapsto by force
  qed
 qed
 show ι ‘ D.cones a′ ⊆ ?F ‘ Hom.set (a′, a)
 proof
  fix X′
  assume X′: X′ ∈ ι ‘ D.cones a′
  hence o X′ ∈ o ‘ ι ‘ D.cones a′ by simp
  with Sr.DN-UP have o X′ ∈ D.cones a′
    by auto
  hence ∃!f. «f : a′ → a» ∧ D.cones-map f χ = o X′
    using a′ is-universal by simp
  from this obtain f where «f : a′ → a» ∧ D.cones-map f χ = o X′
    by auto
  hence f: «f : a′ → a» ∧ ι (D.cones-map f χ) = X′
    using X′ Sr.UP-DN by auto
```

```
        have X' = ?F (φ (a', a) f)
          using f Hom.ψ-φ by presburger
        thus X' ∈ ?F ‘ Hom.set (a', a)
          using f Hom.set-def by force
      qed
    qed
    ultimately show ?thesis
      using bij-betw-def [of ?F Hom.set (a', a) ι ‘ D.cones a'] inj-on-def by auto
  qed
  let ?f = S.mkArr (Hom.set (a', a)) (ι ‘ D.cones a') ?F
  have iso: S.iso ?f
  proof −
    have ?F ∈ Hom.set (a', a) → ι ‘ D.cones a'
      using bij bij-betw-imp-funcset by fast
    hence 1: S.arr ?f
      using ide-apex a' Hom.set-subset-Univ S.UP-mapsto by auto
    thus ?thesis using bij S.iso-char S.set-mkIde by fastforce
  qed
  moreover have ?f = Φ.map a'
    using a' Φ.o-def by force
  finally show ?thesis by auto
 qed
qed

interpretation R: representation-of-functor C S.comp S.setp φ Cones.map a Φ.map ..

lemma χ-in-terms-of-Φ:
shows χ = o (Φ.FUN a (φ (a, a) a))
proof −
  have Φ.FUN a (φ (a, a) a) =
        (λx ∈ Hom.set (a, a). ι (D.cones-map (ψ (a, a) x) χ)) (φ (a, a) a)
    using ide-apex S.Fun-mkArr Φ.map-simp-ide Φ.o-def Φ.preserves-reflects-arr [of a]
    by simp
  also have ... = ι (D.cones-map a χ)
  proof −
    have (λx ∈ Hom.set (a, a). ι (D.cones-map (ψ (a, a) x) χ)) (φ (a, a) a)
          = ι (D.cones-map (ψ (a, a) (φ (a, a) a)) χ)
    proof −
      have φ (a, a) a ∈ Hom.set (a, a)
        using ide-apex Hom.φ-mapsto by fastforce
      thus ?thesis
        using restrict-apply' [of φ (a, a) a Hom.set (a, a)] by blast
    qed
    also have ... = ι (D.cones-map a χ)
    proof −
      have ψ (a, a) (φ (a, a) a) = a
        using ide-apex Hom.ψ-φ [of a a a] by fastforce
      thus ?thesis by metis
    qed
```

```
    finally show ?thesis by auto
  qed
  finally have Φ.FUN a (φ (a, a) a) = ι (D.cones-map a χ) by auto
  also have ... = ι χ
    using ide-apex D.cones-map-ide [of χ a] is-cone by simp
  finally have Φ.FUN a (φ (a, a) a) = ι χ by blast
  hence o (Φ.FUN a (φ (a, a) a)) = o (ι χ) by simp
  thus ?thesis using is-cone Sr.DN-UP by simp
qed

abbreviation Hom
where Hom ≡ Hom.map

abbreviation Φ
where Φ ≡ Φ.map

lemma induces-limit-situation:
shows limit-situation J C D S.comp φ ι a Φ χ
  using χ-in-terms-of-Φ Φo-def by unfold-locales auto

no-notation S.comp      (infixr ‹·S› 55)

end

sublocale limit-cone ⊆ limit-situation J C D replete-setcat.comp φ ι a Φ χ
  using induces-limit-situation by auto
```

20.3.5 Representations of the Cones Functor Induce Limit Situations

```
context representation-of-cones-functor
begin

  interpretation Φ: set-valued-transformation Cop.comp S S.setp ‹Y a› Cones.map Φ ..
  interpretation Ψ: inverse-transformation Cop.comp S ‹Y a› Cones.map Φ ..
  interpretation Ψ: set-valued-transformation Cop.comp S S.setp
                    Cones.map ‹Y a› Ψ.map ..

  abbreviation o
  where o ≡ Cones.o

  abbreviation χ
  where χ ≡ o (S.Fun (Φ a) (φ (a, a) a))

  lemma Cones-SET-eq-ι-img-cones:
  assumes C.ide a'
  shows Cones.SET a' = ι ‘ D.cones a'
  proof −
    have ι ‘ D.cones a' ⊆ S.Univ using S.UP-mapsto by auto
    thus ?thesis using assms Cones.map-ide S.set-mkIde by auto
```

qed

lemma $\iota\chi$:
shows $\iota\ \chi = S.Fun\ (\Phi\ a)\ (\varphi\ (a,\ a)\ a)$
proof −
  have $S.Fun\ (\Phi\ a)\ (\varphi\ (a,\ a)\ a) \in Cones.SET\ a$
    using *Ya.ide-a Hom.φ-mapsto S.Fun-mapsto* [*of* $\Phi\ a$] *Hom.set-map* **by** *fastforce*
  thus *?thesis*
    using *Ya.ide-a Cones-SET-eq-ι-img-cones* **by** *auto*
qed

interpretation χ: *cone J C D a* χ
proof −
  have $\iota\ \chi \in \iota\ `\ D.cones\ a$
    using *Ya.ide-a* $\iota\chi$ *S.Fun-mapsto* [*of* $\Phi\ a$] *Hom.φ-mapsto Hom.set-map*
        Cones-SET-eq-ι-img-cones **by** *fastforce*
  thus *D.cone a* χ
    by (*metis* (*no-types*, *lifting*) *S.DN-UP UNIV-I f-inv-into-f inv-into-into mem-Collect-eq*)
qed

lemma *cone-*χ:
shows *D.cone a* χ **..**

lemma Φ*-FUN-simp*:
assumes a': *C.ide* a' **and** x: $x \in Hom.set\ (a',\ a)$
shows $\Phi.FUN\ a'\ x = Cones.FUN\ (\psi\ (a',\ a)\ x)\ (\iota\ \chi)$
proof −
  have ψx: «$\psi\ (a',\ a)\ x : a' \to a$»
    using *Ya.ide-a* a' *x Hom.ψ-mapsto* **by** *blast*
  have φa: $\varphi\ (a,\ a)\ a \in Hom.set\ (a,\ a)$ **using** *Ya.ide-a Hom.φ-mapsto* **by** *fastforce*
  have $\Phi.FUN\ a'\ x = (\Phi.FUN\ a'\ o\ Ya.FUN\ (\psi\ (a',\ a)\ x))\ (\varphi\ (a,\ a)\ a)$
  proof −
    have $\varphi\ (a',\ a)\ (a \cdot \psi\ (a',\ a)\ x) = x$
      using *Ya.ide-a* a' *x* ψx *Hom.φ-ψ C.comp-cod-arr* **by** *fastforce*
    moreover have $S.arr\ (S.mkArr\ (Hom.set\ (a,\ a))\ (Hom.set\ (a',\ a))$
                        $(\varphi\ (a',\ a) \circ Cop.comp\ (\psi\ (a',\ a)\ x) \circ \psi\ (a,\ a)))$
      by (*metis C.arrI Cop.arr-char Ya.Y-ide-arr*(2) *Ya.preserves-arr* χ*.ide-apex* ψx)
    ultimately show *?thesis*
      using *Ya.ide-a* a' *x Ya.Y-ide-arr* ψx φa *C.ide-in-hom* **by** *auto*
  qed
  also have ... = $(Cones.FUN\ (\psi\ (a',\ a)\ x)\ o\ \Phi.FUN\ a)\ (\varphi\ (a,\ a)\ a)$
  proof −
    have $(\Phi.FUN\ a'\ o\ Ya.FUN\ (\psi\ (a',\ a)\ x))\ (\varphi\ (a,\ a)\ a)$
          $= S.Fun\ (\Phi\ a' \cdot_S\ Y\ a\ (\psi\ (a',\ a)\ x))\ (\varphi\ (a,\ a)\ a)$
      using ψx a' φa *Ya.ide-a Ya.map-simp Hom.set-map* **by** (*elim C.in-homE*, *auto*)
    also have ... = $S.Fun\ (S\ (Cones.map\ (\psi\ (a',\ a)\ x))\ (\Phi\ a))\ (\varphi\ (a,\ a)\ a)$
      using ψx *naturality1* [*of* $\psi\ (a',\ a)\ x$] *naturality2* [*of* $\psi\ (a',\ a)\ x$] **by** *auto*
    also have ... = $(Cones.FUN\ (\psi\ (a',\ a)\ x)\ o\ \Phi.FUN\ a)\ (\varphi\ (a,\ a)\ a)$
    proof −

```
    have S.seq (Cones.map (ψ (a', a) x)) (Φ a)
      using Ya.ide-a ψx Cones.map-preserves-dom [of ψ (a', a) x]
      apply (intro S.seqI)
        apply auto[2]
      by fastforce
    thus ?thesis
      using Ya.ide-a φa Hom.set-map by auto
  qed
  finally show ?thesis by simp
qed
also have ... = Cones.FUN (ψ (a', a) x) (ι χ) using ιχ by simp
finally show ?thesis by auto
qed

lemma χ-is-universal:
assumes D.cone a' χ'
shows «ψ (a', a) (Ψ.FUN a' (ι χ')) : a' → a»
and D.cones-map (ψ (a', a) (Ψ.FUN a' (ι χ'))) χ = χ'
and ⟦ «f' : a' → a»; D.cones-map f' χ = χ' ⟧ ⟹ f' = ψ (a', a) (Ψ.FUN a' (ι χ'))
proof −
  interpret χ': cone J C D a' χ' using assms by auto
  have a': C.ide a' using χ'.ide-apex by simp
  have ιχ': ι χ' ∈ Cones.SET a' using assms a' Cones-SET-eq-ι-img-cones by auto
  let ?f = ψ (a', a) (Ψ.FUN a' (ι χ'))
  have A: Ψ.FUN a' (ι χ') ∈ Hom.set (a', a)
  proof −
    have Ψ.FUN a' ∈ Cones.SET a' → Ya.SET a'
      using a' Ψ.preserves-hom [of a' a' a'] S.Fun-mapsto [of Ψ.map a'] by fastforce
    thus ?thesis using a' ιχ' Ya.ide-a Hom.set-map by auto
  qed
  show f: «?f : a' → a» using A a' Ya.ide-a Hom.ψ-mapsto [of a' a] by auto
  have E: ⋀f. «f : a' → a» ⟹ Cones.FUN f (ι χ) = Φ.FUN a' (φ (a', a) f)
  proof −
    fix f
    assume f: «f : a' → a»
    have φ (a', a) f ∈ Hom.set (a', a)
      using a' Ya.ide-a f Hom.φ-mapsto by auto
    thus Cones.FUN f (ι χ) = Φ.FUN a' (φ (a', a) f)
      using a' f Φ-FUN-simp by simp
  qed
  have I: Φ.FUN a' (Ψ.FUN a' (ι χ')) = ι χ'
  proof −
    have Φ.FUN a' (Ψ.FUN a' (ι χ')) =
          compose (Ψ.DOM a') (Φ.FUN a') (Ψ.FUN a') (ι χ')
      using a' ιχ' Cones.map-ide Ψ.preserves-hom [of a' a' a'] by force
    also have ... = (λx ∈ Ψ.DOM a'. x) (ι χ')
      using a' Ψ.inverts-components S.inverse-arrows-char by force
    also have ... = ι χ'
      using a' ιχ' Cones.map-ide Ψ.preserves-hom [of a' a' a'] by force
```

```
    finally show ?thesis by auto
  qed
  show fχ: D.cones-map ?f χ = χ'
  proof −
    have D.cones-map ?f χ = (o o Cones.FUN ?f o ι) χ
      using f Cones.preserves-arr [of ?f] cone-χ
      by (cases D.cone a χ, auto)
    also have ... = χ'
      using f Ya.ide-a a' A E I by auto
    finally show ?thesis by auto
  qed
  show ⟦ «f' : a' → a»; D.cones-map f' χ = χ' ⟧ ⟹ f' = ?f
  proof −
    assume f': «f' : a' → a» and f'χ: D.cones-map f' χ = χ'
    show f' = ?f
    proof −
      have 1: φ (a', a) f' ∈ Hom.set (a', a) ∧ φ (a', a) ?f ∈ Hom.set (a', a)
        using Ya.ide-a a' f f' Hom.φ-mapsto by auto
      have S.iso (Φ a') using χ'.ide-apex components-are-iso by auto
      hence 2: S.arr (Φ a') ∧ bij-betw (Φ.FUN a') (Hom.set (a', a)) (Cones.SET a')
        using Ya.ide-a a' S.iso-char Hom.set-map by auto
      have Φ.FUN a' (φ (a', a) f') = Φ.FUN a' (φ (a', a) ?f)
      proof −
        have Φ.FUN a' (φ (a', a) ?f) = ι χ'
          using A I Hom.φ-ψ Ya.ide-a a' by simp
        also have ... = Cones.FUN f' (ι χ)
          using f f' A E cone-χ Cones.preserves-arr fχ f'χ by (elim C.in-homE, auto)
        also have ... = Φ.FUN a' (φ (a', a) f')
          using f' E by simp
        finally show ?thesis by argo
      qed
      moreover have inj-on (Φ.FUN a') (Hom.set (a', a))
        using 2 bij-betw-imp-inj-on by blast
      ultimately have 3: φ (a', a) f' = φ (a', a) ?f
        using 1 inj-on-def [of Φ.FUN a' Hom.set (a', a)] by blast
      show ?thesis
      proof −
        have f' = ψ (a', a) (φ (a', a) f')
          using Ya.ide-a a' f' Hom.ψ-φ by simp
        also have ... = ψ (a', a) (Ψ.FUN a' (ι χ'))
          using Ya.ide-a a' Hom.ψ-φ A 3 by simp
        finally show ?thesis by blast
      qed
    qed
  qed
qed

interpretation χ: limit-cone J C D a χ
proof
```

```
  show ⋀a' χ'. D.cone a' χ' ⟹ ∃!f. «f : a' → a» ∧ D.cones-map f χ = χ'
  proof −
    fix a' χ'
    assume 1: D.cone a' χ'
    show ∃!f. «f : a' → a» ∧ D.cones-map f χ = χ'
    proof
      show «ψ (a', a) (Ψ.FUN a' (ι χ')) : a' → a» ∧
           D.cones-map (ψ (a', a) (Ψ.FUN a' (ι χ'))) χ = χ'
        using 1 χ-is-universal by blast
      show ⋀f. «f : a' → a» ∧ D.cones-map f χ = χ' ⟹ f = ψ (a', a) (Ψ.FUN a' (ι χ'))
        using 1 χ-is-universal by blast
    qed
  qed
qed

lemma χ-is-limit-cone:
shows D.limit-cone a χ ..

lemma induces-limit-situation:
shows limit-situation J C D S φ ι a Φ χ
proof
  show χ = χ by simp
  fix a'
  assume a': Cop.ide a'
  let ?F = λx. ι (D.cones-map (ψ (a', a) x) χ)
  show Φ a' = S.mkArr (Hom.set (a', a)) (ι ` D.cones a') ?F
  proof −
    have 1: «Φ a' : S.mkIde (Hom.set (a', a)) →S S.mkIde (ι ` D.cones a')»
      using a' Cones.map-ide Ya.ide-a by auto
    moreover have Φ.DOM a' = Hom.set (a', a)
      using 1 Hom.set-subset-Univ a' Ya.ide-a Hom.set-map by simp
    moreover have Φ.COD a' = ι ` D.cones a'
      using a' Cones-SET-eq-ι-img-cones by fastforce
    ultimately have 2: Φ a' = S.mkArr (Hom.set (a', a)) (ι ` D.cones a') (Φ.FUN a')
      using S.mkArr-Fun [of Φ a'] by fastforce
    also have ... = S.mkArr (Hom.set (a', a)) (ι ` D.cones a') ?F
    proof
      show S.arr (S.mkArr (Hom.set (a', a)) (ι ` D.cones a') (Φ.FUN a'))
        using 1 2 by auto
      show ⋀x. x ∈ Hom.set (a', a) ⟹ Φ.FUN a' x = ?F x
      proof −
        fix x
        assume x: x ∈ Hom.set (a', a)
        hence ψx: «ψ (a', a) x : a' → a»
          using a' Ya.ide-a Hom.ψ-mapsto by auto
        show Φ.FUN a' x = ?F x
        proof −
          have Φ.FUN a' x = Cones.FUN (ψ (a', a) x) (ι χ)
            using a' x Φ-FUN-simp by simp
```

```
        also have ... = restrict (ι o D.cones-map (ψ (a', a) x) o o) (ι ‘ D.cones a) (ι χ)
          using ψx Cones.map-simp Cones.preserves-arr [of ψ (a', a) x] S.Fun-mkArr
          by (elim C.in-homE, auto)
        also have ... = ?F x using cone-χ by simp
        ultimately show ?thesis by simp
      qed
    qed
  qed
  finally show Φ a' = S.mkArr (Hom.set (a', a)) (ι ‘ D.cones a') ?F by auto
 qed
qed

end

sublocale representation-of-cones-functor ⊆ limit-situation J C D S φ ι a Φ χ
 using induces-limit-situation by auto
```

20.4 Categories with Limits

```
context category
begin
```

A category C has limits of shape J if every diagram of shape J admits a limit cone.

```
 definition has-limits-of-shape
 where has-limits-of-shape J ≡ ∀ D. diagram J C D ⟶ (∃ a χ. limit-cone J C D a χ)
```

A category has limits at a type $'j$ if it has limits of shape J for every category J whose arrows are of type $'j$.

```
 definition has-limits
 where has-limits (- :: 'j) ≡ ∀ J :: 'j comp. category J ⟶ has-limits-of-shape J
```

Whether a category has limits of shape J truly depends only on the "shape" (*i.e.* isomorphism class) of J and not on details of its construction.

```
 lemma has-limits-preserved-by-isomorphism:
 assumes has-limits-of-shape J and isomorphic-categories J J'
 shows has-limits-of-shape J'
 proof −
  interpret J: category J
   using assms(2) isomorphic-categories-def isomorphic-categories-axioms-def by auto
  interpret J': category J'
   using assms(2) isomorphic-categories-def isomorphic-categories-axioms-def by auto
  from assms(2) obtain φ ψ where IF: inverse-functors J' J φ ψ
   using isomorphic-categories-def isomorphic-categories-axioms-def
         inverse-functors-sym
   by blast
  interpret IF: inverse-functors J' J φ ψ using IF by auto
  have ψφ: ψ o φ = J.map using IF.inv by metis
  have φψ: φ o ψ = J'.map using IF.inv' by metis
```

have $\bigwedge D'$. *diagram* J' C $D' \Longrightarrow \exists\, a\ \chi$. *limit-cone* J' C D' a χ
proof −
fix D'
assume D': *diagram* J' C D'
interpret D': *diagram* J' C D' **using** D' **by** *auto*
interpret D: *composite-functor* J J' C φ D' **..**
interpret D: *diagram* J C ‹D' *o* φ› **..**
have D: *diagram* J C (D' *o* φ) **..**
from *assms*(*1*) **obtain** a χ **where** χ: *D.limit-cone* a χ
using D *has-limits-of-shape-def* **by** *blast*
interpret χ: *limit-cone* J C ‹D' *o* φ› a χ **using** χ **by** *auto*
interpret A': *constant-functor* J' C a
using χ.*ide-apex* **by** (*unfold-locales*, *auto*)
have $\chi o \psi$: *cone* J' C (D' *o* φ *o* ψ) a (χ *o* ψ)
using *comp-cone-functor IF.G.functor-axioms* χ.*cone-axioms* **by** *fastforce*
hence $\chi o \psi$: *cone* J' C D' a (χ *o* ψ)
using $\varphi\psi$ **by** (*metis D'.functor-axioms Fun.comp-assoc comp-functor-identity*)
interpret $\chi o \psi$: *cone* J' C D' a ‹χ *o* ψ› **using** $\chi o \psi$ **by** *auto*
interpret $\chi o \psi$: *limit-cone* J' C D' a ‹χ *o* ψ›
proof
fix a' χ'
assume χ': D'.*cone* a' χ'
interpret χ': *cone* J' C D' a' χ' **using** χ' **by** *auto*
have $\chi' o \varphi$: *cone* J C (D' *o* φ) a' (χ' *o* φ)
using χ' *comp-cone-functor IF.F.functor-axioms* **by** *fastforce*
interpret $\chi' o \varphi$: *cone* J C ‹D' *o* φ› a' ‹χ' *o* φ› **using** $\chi' o \varphi$ **by** *auto*
have *cone* J C (D' *o* φ) a' (χ' *o* φ) **..**
hence *1*: ∃!f. «f : $a' \to a$» ∧ *D.cones-map* f χ = χ' *o* φ
using χ.*is-universal* **by** *simp*
show ∃!f. «f : $a' \to a$» ∧ D'.*cones-map* f (χ *o* ψ) = χ'
proof
let *?f* = *THE* f. «f : $a' \to a$» ∧ *D.cones-map* f χ = χ' *o* φ
have f: «*?f* : $a' \to a$» ∧ *D.cones-map* *?f* χ = χ' *o* φ
using *1 theI'* [*of* λf. «f : $a' \to a$» ∧ *D.cones-map* f χ = χ' *o* φ] **by** *blast*
have *f-in-hom*: «*?f* : $a' \to a$» **using** f **by** *blast*
have D'.*cones-map* *?f* (χ *o* ψ) = χ'
proof
fix j'
have ¬J'.*arr* j' ⟹ D'.*cones-map* *?f* (χ *o* ψ) j' = χ' j'
proof −
assume j': ¬J'.*arr* j'
have D'.*cones-map* *?f* (χ *o* ψ) j' = *null*
using j' *f-in-hom* $\chi o \psi$ **by** *fastforce*
thus *?thesis*
using j' χ'.*extensionality* **by** *simp*
qed
moreover have J'.*arr* j' ⟹ D'.*cones-map* *?f* (χ *o* ψ) j' = χ' j'
proof −
assume j': J'.*arr* j'

```
        have D'.cones-map ?f (χ o ψ) j' = χ (ψ j') · ?f
          using j' f χoψ by fastforce
        also have ... = D.cones-map ?f χ (ψ j')
          using j' f-in-hom χ χ.is-cone by fastforce
        also have ... = χ' j'
          using j' f χ φψ Fun.comp-def J'.map-simp by metis
        finally show D'.cones-map ?f (χ o ψ) j' = χ' j' by auto
      qed
      ultimately show D'.cones-map ?f (χ o ψ) j' = χ' j' by blast
    qed
    thus «?f : a' → a» ∧ D'.cones-map ?f (χ o ψ) = χ' using f by auto
    fix f'
    assume f': «f' : a' → a» ∧ D'.cones-map f' (χ o ψ) = χ'
    have D.cones-map f' χ = χ' o φ
    proof
      fix j
      have ¬J.arr j ⟹ D.cones-map f' χ j = (χ' o φ) j
        using f' χ χ'oφ.extensionality χ.is-cone mem-Collect-eq restrict-apply by auto
      moreover have J.arr j ⟹ D.cones-map f' χ j = (χ' o φ) j
      proof −
        assume j: J.arr j
        have D.cones-map f' χ j = C (χ j) f'
          using j f' χ.is-cone by auto
        also have ... = C ((χ o ψ) (φ j)) f'
          using j f' ψφ by (metis comp-apply J.map-simp)
        also have ... = D'.cones-map f' (χ o ψ) (φ j)
          using j f' χoψ by fastforce
        also have ... = (χ' o φ) j
          using j f' by auto
        finally show D.cones-map f' χ j = (χ' o φ) j by auto
      qed
      ultimately show D.cones-map f' χ j = (χ' o φ) j by blast
    qed
    hence «f' : a' → a» ∧ D.cones-map f' χ = χ' o φ
      using f' by auto
    moreover have ⋀P x x'. (∃!x. P x) ∧ P x ∧ P x' ⟹ x = x'
      by auto
    ultimately show f' = ?f using 1 f by blast
  qed
 qed
 have limit-cone J' C D' a (χ o ψ) ..
 thus ∃a χ. limit-cone J' C D' a χ by blast
qed
thus ?thesis using has-limits-of-shape-def by auto
qed

end
```

20.4.1 Diagonal Functors

The existence of limits can also be expressed in terms of adjunctions: a category C has limits of shape J if the diagonal functor taking each object a in C to the constant-a diagram and each arrow $f \in C.hom\ a\ a'$ to the constant-f natural transformation between diagrams is a left adjoint functor.

```
locale diagonal-functor =
  C: category C +
  J: category J +
  J-C: functor-category J C
for J :: 'j comp      (infixr ‹·_J› 55)
and C :: 'c comp      (infixr ‹·› 55)
begin

  notation J.in-hom     (‹«- : - →_J -»›)
  notation J-C.comp     (infixr ‹·_[J,C]› 55)
  notation J-C.in-hom   (‹«- : - →_[J,C] -»›)

  definition map :: 'c ⇒ ('j, 'c) J-C.arr
  where map f = (if C.arr f then J-C.MkArr (constant-functor.map J C (C.dom f))
                                           (constant-functor.map J C (C.cod f))
                                           (constant-transformation.map J C f)
                 else J-C.null)

  lemma is-functor:
  shows functor C J-C.comp map
  proof
    fix f
    show ¬ C.arr f ⟹ local.map f = J-C.null
      using map-def by simp
    assume f: C.arr f
    interpret Dom-f: constant-functor J C ‹C.dom f›
      using f by (unfold-locales, auto)
    interpret Cod-f: constant-functor J C ‹C.cod f›
      using f by (unfold-locales, auto)
    interpret Fun-f: constant-transformation J C f
      using f by (unfold-locales, auto)
    show 1: J-C.arr (map f)
      using f map-def by (simp add: Fun-f.natural-transformation-axioms)
    show J-C.dom (map f) = map (C.dom f)
    proof −
      have constant-transformation J C (C.dom f)
        using f by unfold-locales auto
      hence constant-transformation.map J C (C.dom f) = Dom-f.map
        using Dom-f.map-def constant-transformation.map-def [of J C C.dom f] by auto
      thus ?thesis using f 1 by (simp add: map-def J-C.dom-char)
    qed
    show J-C.cod (map f) = map (C.cod f)
    proof −
```

```
  have constant-transformation J C (C.cod f)
    using f by unfold-locales auto
  hence constant-transformation.map J C (C.cod f) = Cod-f.map
    using Cod-f.map-def constant-transformation.map-def [of J C C.cod f] by auto
  thus ?thesis using f 1 by (simp add: map-def J-C.cod-char)
qed
next
fix f g
assume g: C.seq g f
have f: C.arr f using g by auto
interpret Dom-f: constant-functor J C ‹C.dom f›
  using f by unfold-locales auto
interpret Cod-f: constant-functor J C ‹C.cod f›
  using f by unfold-locales auto
interpret Fun-f: constant-transformation J C f
  using f by unfold-locales auto
interpret Cod-g: constant-functor J C ‹C.cod g›
  using g by unfold-locales auto
interpret Fun-g: constant-transformation J C g
  using g by unfold-locales auto
interpret Fun-g: natural-transformation J C Cod-f.map Cod-g.map Fun-g.map
  apply unfold-locales
  using f g C.seqE [of g f] C.comp-arr-dom C.comp-cod-arr Fun-g.extensionality by auto
interpret Fun-fg: vertical-composite
                  J C Dom-f.map Cod-f.map Cod-g.map Fun-f.map Fun-g.map ..
have 1: J-C.arr (map f)
  using f map-def by (simp add: Fun-f.natural-transformation-axioms)
show map (g · f) = map g ·[J,C] map f
proof −
  have map (C g f) = J-C.MkArr Dom-f.map Cod-g.map
                       (constant-transformation.map J C (C g f))
    using f g map-def by simp
  also have ... = J-C.MkArr Dom-f.map Cod-g.map (λj. if J.arr j then C g f else C.null)
  proof −
    have constant-transformation J C (g · f)
      using g by unfold-locales auto
    thus ?thesis using constant-transformation.map-def by metis
  qed
  also have ... = J-C.comp (J-C.MkArr Cod-f.map Cod-g.map Fun-g.map)
                           (J-C.MkArr Dom-f.map Cod-f.map Fun-f.map)
  proof −
    have J-C.MkArr Cod-f.map Cod-g.map Fun-g.map ·[J,C]
         J-C.MkArr Dom-f.map Cod-f.map Fun-f.map
           = J-C.MkArr Dom-f.map Cod-g.map Fun-fg.map
      using J-C.comp-char J-C.comp-MkArr Fun-f.natural-transformation-axioms
            Fun-g.natural-transformation-axioms
      by blast
    also have ... = J-C.MkArr Dom-f.map Cod-g.map
                              (λj. if J.arr j then g · f else C.null)
```

```
      using Fun-fg.extensionality Fun-fg.map-simp-2 by auto
    finally show ?thesis by auto
  qed
  also have ... = map g ·[J,C] map f
    using f g map-def by fastforce
  finally show ?thesis by auto
 qed
qed

sublocale functor C J-C.comp map
 using is-functor by auto
```

The objects of $[J,\ C]$ correspond bijectively to diagrams of shape $(\cdot_J)$ in $(\cdot)$.

```
lemma ide-determines-diagram:
assumes J-C.ide d
shows diagram J C (J-C.Map d) and J-C.MkIde (J-C.Map d) = d
proof −
 interpret δ: natural-transformation J C ‹J-C.Map d› ‹J-C.Map d› ‹J-C.Map d›
   using assms J-C.ide-char J-C.arr-MkArr by fastforce
 interpret D: functor J C ‹J-C.Map d› ..
 show diagram J C (J-C.Map d) ..
 show J-C.MkIde (J-C.Map d) = d
   using assms J-C.ide-char by (metis J-C.ideD(1) J-C.MkArr-Map)
qed

lemma diagram-determines-ide:
assumes diagram J C D
shows J-C.ide (J-C.MkIde D) and J-C.Map (J-C.MkIde D) = D
proof −
 interpret D: diagram J C D using assms by auto
 show J-C.ide (J-C.MkIde D) using J-C.ide-char
   using D.functor-axioms J-C.ide-MkIde by auto
 thus J-C.Map (J-C.MkIde D) = D
   using J-C.in-homE by simp
qed

lemma bij-betw-ide-diagram:
shows bij-betw J-C.Map (Collect J-C.ide) (Collect (diagram J C))
proof (intro bij-betwI)
 show J-C.Map ∈ Collect J-C.ide → Collect (diagram J C)
   using ide-determines-diagram by blast
 show J-C.MkIde ∈ Collect (diagram J C) → Collect J-C.ide
   using diagram-determines-ide by blast
 show ⋀d. d ∈ Collect J-C.ide ⟹ J-C.MkIde (J-C.Map d) = d
   using ide-determines-diagram by blast
 show ⋀D. D ∈ Collect (diagram J C) ⟹ J-C.Map (J-C.MkIde D) = D
   using diagram-determines-ide by blast
qed
```

Arrows from from the diagonal functor correspond bijectively to cones.

lemma *arrow-determines-cone*:
assumes *J-C.ide d* **and** *arrow-from-functor C J-C.comp map a d x*
shows *cone J C* (*J-C.Map d*) *a* (*J-C.Map x*)
and *J-C.MkArr* (*constant-functor.map J C a*) (*J-C.Map d*) (*J-C.Map x*) = *x*
proof −
  interpret *D*: *diagram J C* ‹*J-C.Map d*›
    using *assms ide-determines-diagram* **by** *auto*
  interpret *x*: *arrow-from-functor C J-C.comp map a d x*
    using *assms* **by** *auto*
  interpret *A*: *constant-functor J C a*
    using *x.arrow* **by** (*unfold-locales*, *auto*)
  interpret *α*: *constant-transformation J C a*
    using *x.arrow* **by** (*unfold-locales*, *auto*)
  have *Dom-x*: *J-C.Dom x* = *A.map*
    using *J-C.in-hom-char map-def x.arrow* **by** *force*
  have *Cod-x*: *J-C.Cod x* = *J-C.Map d*
    using *x.arrow* **by** *auto*
  interpret *χ*: *natural-transformation J C A.map* ‹*J-C.Map d*› ‹*J-C.Map x*›
    using *x.arrow J-C.arr-char* [*of x*] *Dom-x Cod-x* **by** *force*
  show *D.cone a* (*J-C.Map x*) **..**
  show *J-C.MkArr A.map* (*J-C.Map d*) (*J-C.Map x*) = *x*
    using *x.arrow Dom-x Cod-x χ.natural-transformation-axioms*
    by (*intro J-C.arr-eqI*, *auto*)
qed

lemma *cone-determines-arrow*:
assumes *J-C.ide d* **and** *cone J C* (*J-C.Map d*) *a χ*
shows *arrow-from-functor C J-C.comp map a d*
        (*J-C.MkArr* (*constant-functor.map J C a*) (*J-C.Map d*) *χ*)
and *J-C.Map* (*J-C.MkArr* (*constant-functor.map J C a*) (*J-C.Map d*) *χ*) = *χ*
proof −
  interpret *χ*: *cone J C* ‹*J-C.Map d*› *a χ* **using** *assms*(*2*) **by** *auto*
  let *?x* = *J-C.MkArr χ.A.map* (*J-C.Map d*) *χ*
  interpret *x*: *arrow-from-functor C J-C.comp map a d ?x*
  proof
    have «*J-C.MkArr χ.A.map* (*J-C.Map d*) *χ* :
          J-C.MkIde χ.A.map →[*J*,*C*] *J-C.MkIde* (*J-C.Map d*)»
      using *χ.natural-transformation-axioms* **by** *auto*
    moreover have *J-C.MkIde χ.A.map* = *map a*
      using *χ.A.value-is-ide map-def χ.A.map-def C.ide-char*
      by (*metis* (*no-types*, *lifting*) *J-C.dom-MkArr preserves-arr preserves-dom*)
    moreover have *J-C.MkIde* (*J-C.Map d*) = *d*
      using *assms ide-determines-diagram*(*2*) **by** *simp*
    ultimately show *C.ide a* ∧ «*J-C.MkArr χ.A.map* (*J-C.Map d*) *χ* : *map a* →[*J*,*C*] *d*»
      using *χ.A.value-is-ide* **by** *simp*
  qed
  show *arrow-from-functor C J-C.comp map a d ?x* **..**
  show *J-C.Map* (*J-C.MkArr* (*constant-functor.map J C a*) (*J-C.Map d*) *χ*) = *χ*
    by (*simp add*: *χ.natural-transformation-axioms*)

qed

Transforming a cone by composing at the apex with an arrow g corresponds, via the preceding bijections, to composition in $[J,\ C]$ with the image of g under the diagonal functor.

lemma *cones-map-is-composition*:
assumes «$g : a' \to a$» **and** *cone J C D a* χ
shows *J-C.MkArr* (*constant-functor.map J C a'*) *D* (*diagram.cones-map J C D g* χ)
  = *J-C.MkArr* (*constant-functor.map J C a*) *D* χ $\cdot_{[J,C]}$ *map g*
proof −
  interpret *A*: *constant-transformation J C a*
    using *assms*(*1*) **by** (*unfold-locales*, *auto*)
  interpret χ: *cone J C D a* χ **using** *assms*(*2*) **by** *auto*
  have *cone-*χ: *cone J C D a* χ **..**
  interpret *A'*: *constant-transformation J C a'*
    using *assms*(*1*) **by** (*unfold-locales*, *auto*)
  let *?*χ' = χ*.D.cones-map g* χ
  interpret χ': *cone J C D a' ?*χ'
    using *assms*(*1*) *cone-*χ χ*.D.cones-map-mapsto* **by** *blast*
  let *?x* = *J-C.MkArr* χ*.A.map D* χ
  let *?x'* = *J-C.MkArr* χ'*.A.map D ?*χ'
  show *?x'* = *J-C.comp ?x* (*map g*)
  proof (*intro J-C.arr-eqI*)
    have *x*: *J-C.arr ?x*
      using χ*.natural-transformation-axioms J-C.arr-char* [*of ?x*] **by** *simp*
    show *x'*: *J-C.arr ?x'*
      using χ'*.natural-transformation-axioms J-C.arr-char* [*of ?x'*] **by** *simp*
    have *3*: «*?x* : *map a* $\to_{[J,C]}$ *J-C.MkIde D*»
      using χ*.D.diagram-axioms arrow-from-functor.arrow cone-*χ *cone-determines-arrow*(*1*)
          diagram-determines-ide(*1*)
      by *fastforce*
    have *4*: «*?x'* : *map a'* $\to_{[J,C]}$ *J-C.MkIde D*»
      by (*metis* (*no-types*, *lifting*) *J-C.Dom.simps*(*1*) *J-C.Dom-cod J-C.Map-cod*
          J-C.cod-MkArr χ'*.cone-axioms arrow-from-functor.arrow category.ide-cod*
          cone-determines-arrow(*1*) *functor-def is-functor x*)
    have *seq-xg*: *J-C.seq ?x* (*map g*)
      using *assms*(*1*) *3 preserves-hom* [*of g*] **by** (*intro J-C.seqI'*, *auto*)
    show *2*: *J-C.seq ?x* (*map g*)
      using *seq-xg J-C.seqI'* **by** *blast*
    show *J-C.Dom ?x'* = *J-C.Dom* (*?x* $\cdot_{[J,C]}$ *map g*)
    proof −
      have *J-C.Dom ?x'* = *J-C.Dom* (*J-C.dom ?x'*)
        using *x' J-C.Dom-dom* **by** *simp*
      also have ... = *J-C.Dom* (*map a'*)
        using *4* **by** *force*
      also have ... = *J-C.Dom* (*J-C.dom* (*?x* $\cdot_{[J,C]}$ *map g*))
        using *assms*(*1*) *2* **by** *auto*
      also have ... = *J-C.Dom* (*?x* $\cdot_{[J,C]}$ *map g*)
        using *seq-xg J-C.Dom-dom J-C.seqI'* **by** *blast*

```
    finally show ?thesis by auto
  qed
  show J-C.Cod ?x' = J-C.Cod (?x ·[J,C] map g)
  proof −
    have J-C.Cod ?x' = J-C.Cod (J-C.cod ?x')
      using x' J-C.Cod-cod by simp
    also have ... = J-C.Cod (J-C.MkIde D)
      using 4 by force
    also have ... = J-C.Cod (J-C.cod (?x ·[J,C] map g))
      using 2 3 J-C.cod-comp J-C.in-homE by metis
    also have ... = J-C.Cod (?x ·[J,C] map g)
      using seq-xg J-C.Cod-cod J-C.seqI' by blast
    finally show ?thesis by auto
  qed
  show J-C.Map ?x' = J-C.Map (?x ·[J,C] map g)
  proof −
    interpret g: constant-transformation J C g
      apply unfold-locales using assms(1) by auto
    interpret χog: vertical-composite J C A'.map χ.A.map D g.map χ
      using assms(1) C.comp-arr-dom C.comp-cod-arr A'.extensionality g.extensionality
      apply (unfold-locales, auto)
      by (elim J.seqE, auto)
    have J-C.Map (?x ·[J,C] map g) = χog.map
      using assms(1) 2 J-C.comp-char map-def by auto
    also have ... = J-C.Map ?x'
      using x' χog.map-def J-C.arr-char [of ?x'] natural-transformation.extensionality
            assms(1) cone-χ χog.map-simp-2
      by fastforce
    finally show ?thesis by auto
  qed
 qed
qed
```

Coextension along an arrow from a functor is equivalent to a transformation of cones.

```
lemma coextension-iff-cones-map:
assumes x: arrow-from-functor C J-C.comp map a d x
and g: «g : a' → a»
and x': «x' : map a' →[J,C] d»
shows arrow-from-functor.is-coext C J-C.comp map a x a' x' g
        ⟷ J-C.Map x' = diagram.cones-map J C (J-C.Map d) g (J-C.Map x)
proof −
  interpret x: arrow-from-functor C J-C.comp map a d x
    using assms by auto
  interpret A': constant-functor J C a'
    using assms(2) by (unfold-locales, auto)
  have x': arrow-from-functor C J-C.comp map a' d x'
    using A'.value-is-ide assms(3) by (unfold-locales, blast)
  have d: J-C.ide d using J-C.ide-cod x.arrow by blast
  let ?D = J-C.Map d
```

let *?χ = J-C.Map x*
let *?χ′ = J-C.Map x′*
interpret *D*: *diagram J C ?D*
using *ide-determines-diagram J-C.ide-cod x.arrow* **by** *blast*
interpret *χ*: *cone J C ?D a ?χ*
using *assms*(*1*) *d arrow-determines-cone* **by** *simp*
interpret *γ*: *constant-transformation J C g*
using *g χ.ide-apex* **by** (*unfold-locales*, *auto*)
interpret *χog*: *vertical-composite J C A′.map χ.A.map ?D γ.map ?χ*
using *g C.comp-arr-dom C.comp-cod-arr γ.extensionality* **by** (*unfold-locales*, *auto*)
show *?thesis*
proof
assume *0*: *x.is-coext a′ x′ g*
show *?χ′ = D.cones-map g ?χ*
proof −
have *1*: *x′ = x* ⋅$_{[J,C]}$ *map g*
using *0 x.is-coext-def* **by** *blast*
hence *?χ′ = J-C.Map x′*
using *0 x.is-coext-def* **by** *fast*
moreover have *... = D.cones-map g ?χ*
proof −
have *J-C.MkArr A′.map* (*J-C.Map d*) (*D.cones-map g* (*J-C.Map x*)) *= x′*
proof −
have *J-C.MkArr A′.map* (*J-C.Map d*) (*D.cones-map g* (*J-C.Map x*)) =
x ⋅$_{[J,C]}$ *map g*
using *d g cones-map-is-composition arrow-determines-cone*(*2*) *χ.cone-axioms*
x.arrow-from-functor-axioms
by *auto*
thus *?thesis* **by** (*metis 1*)
qed
moreover have *J-C.arr* (*J-C.MkArr A′.map* (*J-C.Map d*) (*D.cones-map g* (*J-C.Map x*)))
using *1 d g cones-map-is-composition preserves-arr arrow-determines-cone*(*2*)
χ.cone-axioms x.arrow-from-functor-axioms assms(*3*)
by *auto*
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *blast*
qed
next
assume *X′*: *?χ′ = D.cones-map g ?χ*
show *x.is-coext a′ x′ g*
proof −
have *4*: *J-C.seq x* (*map g*)
using *g x.arrow mem-Collect-eq preserves-arr preserves-cod*
by (*elim C.in-homE*, *auto*)
hence *1*: *x* ⋅$_{[J,C]}$ *map g =*
J-C.MkArr (*J-C.Dom* (*map g*)) (*J-C.Cod x*)
(*vertical-composite.map J C* (*J-C.Map* (*map g*)) *?χ*)

```
      using J-C.comp-char [of x map g] by simp
    have 2: vertical-composite.map J C (J-C.Map (map g)) ?χ = χog.map
      by (simp add: map-def γ.value-is-arr γ.natural-transformation-axioms)
    have 3: ... = D.cones-map g ?χ
      using g χog.map-simp-2 χ.cone-axioms χog.extensionality by auto
    have J-C.MkArr A'.map ?D ?χ' = J-C.comp x (map g)
    proof −
      have f1: A'.map = J-C.Dom (map g)
        using γ.natural-transformation-axioms map-def g by auto
      have J-C.Map d = J-C.Cod x
        using x.arrow by auto
      thus ?thesis using f1 X' 1 2 3 by argo
    qed
    moreover have J-C.MkArr A'.map ?D ?χ' = x'
      using d x' arrow-determines-cone by blast
    ultimately show ?thesis
      using g x.is-coext-def by simp
  qed
 qed
qed

end

locale right-adjoint-to-diagonal-functor =
  C: category C +
  J: category J +
  J-C: functor-category J C +
  Δ: diagonal-functor J C +
  functor J-C.comp C G +
  Adj: meta-adjunction J-C.comp C Δ.map G φ ψ
for J :: 'j comp      (infixr ‹·J› 55)
and C :: 'c comp      (infixr ‹·› 55)
and G :: ('j, 'c) functor-category.arr ⇒ 'c
and φ :: 'c ⇒ ('j, 'c) functor-category.arr ⇒ 'c
and ψ :: ('j, 'c) functor-category.arr ⇒ 'c ⇒ ('j, 'c) functor-category.arr +
assumes adjoint: adjoint-functors J-C.comp C Δ.map G
begin

  interpretation S: replete-setcat .
  interpretation Adj: adjunction J-C.comp C S.comp S.setp Adj.φC Adj.φD Δ.map G
                    φ ψ Adj.η Adj.ε Adj.Φ Adj.Ψ
    using Adj.induces-adjunction by simp
```

A right adjoint G to a diagonal functor maps each object d of $[J, C]$ (corresponding to a diagram D of shape $(\cdot_J)$ in $(\cdot)$ to an object of $(\cdot)$. This object is the limit object, and the component at d of the counit of the adjunction determines the limit cone.

```
  lemma gives-limit-cones:
  assumes diagram J C D
  shows limit-cone J C D (G (J-C.MkIde D)) (J-C.Map (Adj.ε (J-C.MkIde D)))
```

proof −
interpret *D*: *diagram J C D* **using** *assms* **by** *auto*
let *?d* = *J-C.MkIde D*
let *?a* = *G ?d*
let *?x* = *Adj.ε ?d*
let *?χ* = *J-C.Map ?x*
have *diagram J C D* **..**
hence *1*: *J-C.ide ?d* **using** Δ.*diagram-determines-ide* **by** *auto*
hence *2*: *J-C.Map* (*J-C.MkIde D*) = *D*
using *assms 1 J-C.in-homE* Δ.*diagram-determines-ide*(*2*) **by** *simp*
interpret *x*: *terminal-arrow-from-functor C J-C.comp* Δ.*map ?a ?d ?x*
apply *unfold-locales*
apply (*metis* (*no-types*, *lifting*) *1 preserves-ide Adj.ε-in-terms-of-ψ*
Adj.εo-def Adj.εo-in-hom)
by (*metis 1 Adj.has-terminal-arrows-from-functor*(*1*)
terminal-arrow-from-functor.is-terminal)
have *3*: *arrow-from-functor C J-C.comp* Δ.*map ?a ?d ?x* **..**
interpret *χ*: *cone J C D ?a ?χ*
using *1 2 3* Δ.*arrow-determines-cone* [*of ?d*] **by** *auto*
have *cone-χ*: *D.cone ?a ?χ* **..**
interpret *χ*: *limit-cone J C D ?a ?χ*
proof
fix *a′ χ′*
assume *cone-χ′*: *D.cone a′ χ′*
interpret *χ′*: *cone J C D a′ χ′* **using** *cone-χ′* **by** *auto*
let *?x′* = *J-C.MkArr χ′.A.map D χ′*
interpret *x′*: *arrow-from-functor C J-C.comp* Δ.*map a′ ?d ?x′*
using *1 2* **by** (*metis* Δ.*cone-determines-arrow*(*1*) *cone-χ′*)
have *arrow-from-functor C J-C.comp* Δ.*map a′ ?d ?x′* **..**
hence *4*: ∃!*g*. *x.is-coext a′ ?x′ g*
using *x.is-terminal* **by** *simp*
have *5*: ⋀*g*. «*g* : *a′* →*C* *?a*» ⟹ *x.is-coext a′ ?x′ g* ⟷ *D.cones-map g ?χ* = *χ′*
using Δ.*coextension-iff-cones-map x′.arrow x.arrow-from-functor-axioms* **by** *auto*
have *6*: ⋀*g*. *x.is-coext a′ ?x′ g* ⟹ «*g* : *a′* →*C* *?a*»
using *x.is-coext-def* **by** *simp*
show ∃!*g*. «*g* : *a′* →*C* *?a*» ∧ *D.cones-map g ?χ* = *χ′*
proof −
have ∃ *g*. «*g* : *a′* →*C* *?a*» ∧ *D.cones-map g ?χ* = *χ′*
using *4 5 6* **by** *meson*
thus *?thesis*
using *4 5 6* **by** *blast*
qed
qed
show *D.limit-cone ?a ?χ* **..**
qed

corollary *gives-limits*:
assumes *diagram J C D*
shows *diagram.has-as-limit J C D* (*G* (*J-C.MkIde D*))

```
  using assms gives-limit-cones by fastforce

end

lemma (in category) limits-are-isomorphic:
fixes J :: 'j comp
assumes limit-cone J C D a χ and limit-cone J C D a' χ'
shows isomorphic a a' and iso (limit-cone.induced-arrow J C D a χ a' χ')
proof −
  interpret J: category J
    using assms(1) limit-cone.axioms(2) by metis
  interpret C: category C
    using assms(1) limit-cone.axioms(1) by metis
  interpret D: diagram J C D
    using assms(1) limit-cone.axioms(3) by metis
  interpret χ: limit-cone J C D a χ
    using assms(1) by blast
  interpret χ': limit-cone J C D a' χ'
    using assms(2) by blast
  have 1: ∃!f. «f : a → a'» ∧ D.cones-map f χ' = χ
    using χ'.is-universal [of a χ] χ.cone-axioms by simp
  have 2: ∃!g. «g : a' → a» ∧ D.cones-map g χ = χ'
    using χ.is-universal [of a' χ'] χ'.cone-axioms by simp
  define f where f = χ'.induced-arrow a χ
  define g where g = χ.induced-arrow a' χ'
  have f: «f : a → a'» ∧ D.cones-map f χ' = χ
    using f-def χ'.induced-arrowI(1−2) χ.is-cone by blast
  have g: «g : a' → a» ∧ D.cones-map g χ = χ'
    using g-def χ.induced-arrowI(1−2) χ'.is-cone by blast
  have ∗: inverse-arrows f g
  proof
    show ide (g · f)
    proof −
      have g · f = a
      proof −
        have ∃!h. «h : a → a» ∧ D.cones-map h χ = χ
          using χ.is-universal [of a χ] χ.cone-axioms by blast
        moreover have «g · f : a → a»
          using f g by blast
        moreover have D.cones-map (g · f) χ = χ
        proof
          fix j :: 'j
          show D.cones-map (g · f) χ j = χ j
          proof (cases J.arr j)
            show ¬ J.arr j ⟹ ?thesis
              using f g χ.cone-axioms χ.extensionality by fastforce
            assume j: J.arr j
            have D.cone (dom g) (D.cones-map g χ)
              using g D.cones-map-mapsto χ.cone-axioms by blast
```

```
        thus ?thesis
          using f g χ.cone-axioms D.cones-map-comp [of g f] by fastforce
      qed
    qed
    moreover have «a : a → a»
      using χ.ide-apex by auto
    moreover have D.cones-map a χ = χ
      using f χ.cone-axioms D.cones-map-ide by blast
    ultimately show ?thesis by blast
  qed
  thus ?thesis
    using χ.ide-apex by blast
qed
show ide (f · g)
proof −
  have f · g = a'
  proof −
    have ∃!h. «h : a' → a'» ∧ D.cones-map h χ' = χ'
      using χ'.is-universal [of a' χ'] χ'.cone-axioms by blast
    moreover have «f · g : a' → a'»
      using f g by blast
    moreover have D.cones-map (f · g) χ' = χ'
    proof
      fix j :: 'j
      show D.cones-map (f · g) χ' j = χ' j
      proof (cases J.arr j)
        show ¬ J.arr j ⟹ ?thesis
          using f g χ'.cone-axioms χ'.extensionality by fastforce
        assume j: J.arr j
        have D.cone (dom f) (D.cones-map f χ')
          using f D.cones-map-mapsto χ'.cone-axioms by blast
        thus ?thesis
          using f g χ'.cone-axioms D.cones-map-comp [of f g] by fastforce
      qed
    qed
    moreover have «a' : a' → a'»
      using χ'.ide-apex by auto
    moreover have D.cones-map a' χ' = χ'
      using g χ'.cone-axioms D.cones-map-ide by blast
    ultimately show ?thesis by blast
  qed
  thus ?thesis
    using χ'.ide-apex by blast
qed
qed
show isomorphic a a'
  using ∗ f g by blast
show iso (χ.induced-arrow a' χ')
  using ∗ g-def by blast
```

qed

lemma (**in** *category*) *has-limits-iff-left-adjoint-diagonal*:
assumes *category J*
shows *has-limits-of-shape J* ⟷
left-adjoint-functor C (*functor-category.comp J C*) (*diagonal-functor.map J C*)
proof −
interpret *J*: *category J* **using** *assms* **by** *auto*
interpret *J-C*: *functor-category J C* **..**
interpret Δ: *diagonal-functor J C* **..**
show *?thesis*
proof
assume *A*: *left-adjoint-functor C J-C.comp* Δ.*map*
interpret Δ: *left-adjoint-functor C J-C.comp* Δ.*map* **using** *A* **by** *auto*
interpret *Adj*: *meta-adjunction J-C.comp C* Δ.*map* Δ.*G* Δ.φ Δ.ψ
using Δ.*induces-meta-adjunction* **by** *auto*
have *1*: *adjoint-functors J-C.comp C* Δ.*map* Δ.*G*
using *adjoint-functors-def* Δ.*induces-meta-adjunction* **by** *blast*
interpret *G*: *right-adjoint-to-diagonal-functor J C* Δ.*G* Δ.φ Δ.ψ
using *1* **by** *unfold-locales auto*
show *has-limits-of-shape J*
using *A G.gives-limits has-limits-of-shape-def* **by** *blast*
next

If *has-limits J*, then every diagram D from J to C has a limit cone. This means that, for every object d of the functor category $[J,\ C]$, there exists an object a of $(\cdot)$ and a terminal arrow from $\Delta\ a$ to d in $[J,\ C]$. The terminal arrow is given by the limit cone.

assume *A*: *has-limits-of-shape J*
show *left-adjoint-functor C J-C.comp* Δ.*map*
proof
fix *d*
assume *D*: *J-C.ide d*
interpret *D*: *diagram J C* ‹*J-C.Map d*›
using *D* Δ.*ide-determines-diagram* **by** *auto*
let *?D* = *J-C.Map d*
have *diagram J C* (*J-C.Map d*) **..**
from *this* **obtain** *a* χ **where** *limit*: *limit-cone J C ?D a* χ
using *A has-limits-of-shape-def* **by** *blast*
interpret *A*: *constant-functor J C a*
using *limit* **by** (*simp add*: *Limit.cone-def limit-cone-def*)
interpret χ: *limit-cone J C ?D a* χ **using** *limit* **by** *simp*
have *cone-*χ: *cone J C ?D a* χ **..**
let *?x* = *J-C.MkArr A.map ?D* χ
interpret *x*: *arrow-from-functor C J-C.comp* Δ.*map a d ?x*
using *D cone-*χ Δ.*cone-determines-arrow* **by** *auto*
have *terminal-arrow-from-functor C J-C.comp* Δ.*map a d ?x*
proof
show ⋀*a' x'*. *arrow-from-functor C J-C.comp* Δ.*map a' d x'* ⟹ ∃!*g*. *x.is-coext a' x' g*
proof −

```
fix a' x'
assume x': arrow-from-functor C J-C.comp Δ.map a' d x'
interpret x': arrow-from-functor C J-C.comp Δ.map a' d x' using x' by auto
interpret A': constant-functor J C a'
  by (unfold-locales, simp add: x'.arrow)
let ?χ' = J-C.Map x'
interpret χ': cone J C ?D a' ?χ'
  using D x' Δ.arrow-determines-cone by auto
have cone-χ': cone J C ?D a' ?χ' ..
let ?g = χ.induced-arrow a' ?χ'
show ∃!g. x.is-coext a' x' g
proof
  show x.is-coext a' x' ?g
  proof (unfold x.is-coext-def)
    have 1: «?g : a' → a» ∧ D.cones-map ?g χ = ?χ'
      using χ.induced-arrow-def χ.is-universal cone-χ'
            theI' [of λf. «f : a' → a» ∧ D.cones-map f χ = ?χ']
      by presburger
    hence 2: x' = ?x ·[J,C] Δ.map ?g
    proof −
      have x' = J-C.MkArr A'.map ?D ?χ'
        using D Δ.arrow-determines-cone(2) x'.arrow-from-functor-axioms by auto
      thus ?thesis
        using 1 cone-χ Δ.cones-map-is-composition [of ?g a' a ?D χ] by simp
    qed
    show «?g : a' → a» ∧ x' = ?x ·[J,C] Δ.map ?g
      using 1 2 by auto
  qed
  next
  fix g
  assume X: x.is-coext a' x' g
  show g = ?g
  proof −
    have «g : a' → a» ∧ D.cones-map g χ = ?χ'
    proof
      show G: «g : a' → a» using X x.is-coext-def by blast
      show D.cones-map g χ = ?χ'
      proof −
        have ?χ' = J-C.Map (?x ·[J,C] Δ.map g)
          using X x.is-coext-def [of a' x' g] by fast
        also have ... = D.cones-map g χ
        proof −
          interpret map-g: constant-transformation J C g
            using G by (unfold-locales, auto)
          interpret χ': vertical-composite J C
                        map-g.F.map A.map ‹χ.Φ.Ya.Cop-S.Map d›
                        map-g.map χ
          proof (intro-locales)
            have map-g.G.map = A.map
```

```
            using G by blast
          thus natural-transformation-axioms J (·) map-g.F.map A.map map-g.map
            using map-g.natural-transformation-axioms
            by (simp add: natural-transformation-def)
        qed
      have J-C.Map (?x ·[J,C] Δ.map g) = vertical-composite.map J C map-g.map χ
        proof −
          have J-C.seq ?x (Δ.map g)
            using G x.arrow by auto
          thus ?thesis
            using G Δ.map-def J-C.Map-comp' [of ?x Δ.map g] by auto
        qed
        also have ... = D.cones-map g χ
          using G cone-χ χ'.map-def map-g.map-def χ.naturality2 χ'.map-simp-2
          by auto
        finally show ?thesis by blast
      qed
      finally show ?thesis by auto
    qed
  qed
  thus ?thesis
    using cone-χ' χ.is-universal χ.induced-arrow-def
        theI-unique [of λg. «g : a' → a» ∧ D.cones-map g χ = ?χ' g]
    by presburger
 qed
 qed
 qed
 qed
 thus ∃ a x. terminal-arrow-from-functor C J-C.comp Δ.map a d x by auto
 qed
 qed
qed
```

20.5 Right Adjoint Functors Preserve Limits

```
context right-adjoint-functor
begin

 lemma preserves-limits:
 fixes J :: 'j comp
 assumes diagram J C E and diagram.has-as-limit J C E a
 shows diagram.has-as-limit J D (G o E) (G a)
 proof −
```

From the assumption that E has a limit, obtain a limit cone χ.

```
  interpret J: category J using assms(1) diagram-def by auto
  interpret E: diagram J C E using assms(1) by auto
  from assms(2) obtain χ where χ: limit-cone J C E a χ by auto
  interpret χ: limit-cone J C E a χ using χ by auto
```

have *a*: *C.ide a* **using** *χ.ide-apex* **by** *auto*

Form the E-image GE of the diagram E.

interpret *GE*: *composite-functor J C D E G* **..**
interpret *GE*: *diagram J D GE.map* **..**

Let $G\chi$ be the G-image of the cone χ, and note that it is a cone over GE.

let *?Gχ* = *G o χ*
interpret *Gχ*: *cone J D GE.map ‹G a› ?Gχ*
using *χ.cone-axioms preserves-cones* **by** *blast*

Claim that $G\chi$ is a limit cone for diagram GE.

interpret *Gχ*: *limit-cone J D GE.map ‹G a› ?Gχ*
proof

Let κ be an arbitrary cone over GE.

fix *b κ*
assume *κ*: *GE.cone b κ*
interpret *κ*: *cone J D GE.map b κ* **using** *κ* **by** *auto*
interpret *Fb*: *constant-functor J C ‹F b›*
apply *unfold-locales*
by (*meson F-is-functor κ.ide-apex functor.preserves-ide*)
interpret *Adj*: *meta-adjunction C D F G φ ψ*
using *induces-meta-adjunction* **by** *auto*
interpret *S*: *replete-setcat* **.**
interpret *Adj*: *adjunction C D S.comp S.setp*
Adj.φC Adj.φD F G φ ψ Adj.η Adj.ε Adj.Φ Adj.Ψ
using *Adj.induces-adjunction* **by** *simp*

For each arrow j of J, let $\chi'\ j$ be defined to be the adjunct of $\chi\ j$. We claim that χ' is a cone over E.

let *?χ′* = *λj. if J.arr j then Adj.ε* (*C.cod* (*E j*)) ·$_C$ *F* (*κ j*) *else C.null*
have *cone-χ′*: *E.cone* (*F b*) *?χ′*
proof
show ⋀*j*. ¬*J.arr j* ⟹ *?χ′ j* = *C.null* **by** *simp*
fix *j*
assume *j*: *J.arr j*
show *C.arr* (*?χ′ j*) **using** *j ψ-in-hom* **by** *simp*
show *E j* ·$_C$ *?χ′* (*J.dom j*) = *?χ′ j*
proof −
have *E j* ·$_C$ *?χ′* (*J.dom j*) = (*E j* ·$_C$ *Adj.ε* (*E* (*J.dom j*))) ·$_C$ *F* (*κ* (*J.dom j*))
using *j C.comp-assoc* **by** *simp*
also have ... = *Adj.ε* (*E* (*J.cod j*)) ·$_C$ *F* (*κ j*)
proof −
have (*E j* ·$_C$ *Adj.ε* (*E* (*J.dom j*))) ·$_C$ *F* (*κ* (*J.dom j*))
= (*Adj.ε* (*C.cod* (*E j*)) ·$_C$ *Adj.FG.map* (*E j*)) ·$_C$ *F* (*κ* (*J.dom j*))
using *j Adj.ε.naturality* [*of E j*] **by** *fastforce*
also have ... = *Adj.ε* (*C.cod* (*E j*)) ·$_C$ *Adj.FG.map* (*E j*) ·$_C$ *F* (*κ* (*J.dom j*))
using *C.comp-assoc* **by** *simp*

```
      also have ... = Adj.ε (E (J.cod j)) ·C F (κ j)
      proof −
        have Adj.FG.map (E j) ·C F (κ (J.dom j)) = F (GE.map j ·D κ (J.dom j))
          using j by simp
        hence Adj.FG.map (E j) ·C F (κ (J.dom j)) = F (κ j)
          using j κ.naturality1 by metis
        thus ?thesis using j by simp
      qed
      finally show ?thesis by auto
    qed
    also have ... = ?χ′ j
      using j by simp
    finally show ?thesis by auto
  qed
  show ?χ′ (J.cod j) ·C Fb.map j = ?χ′ j
  proof −
    have ?χ′ (J.cod j) ·C Fb.map j = Adj.ε (E (J.cod j)) ·C F (κ (J.cod j))
      using j Fb.value-is-ide Adj.ε.preserves-hom C.comp-arr-dom [of F (κ (J.cod j))]
            C.comp-assoc
      by simp
    also have ... = Adj.ε (E (J.cod j)) ·C F (κ j)
      using j κ.naturality1 κ.naturality2 Adj.ε.naturality J.arr-cod-iff-arr
      by (metis J.cod-cod κ.A.map-simp)
    also have ... = ?χ′ j using j by simp
    finally show ?thesis by auto
  qed
qed
```

Using the universal property of the limit cone χ, obtain the unique arrow f that transforms χ into χ'.

```
from this χ.is-universal [of F b ?χ′] obtain f
  where f: «f : F b →C a» ∧ E.cones-map f χ = ?χ′
  by auto
```

Let g be the adjunct of f, and show that g transforms $G\chi$ into κ.

```
let ?g = G f ·D Adj.η b
have 1: «?g : b →D G a» using f κ.ide-apex by fastforce
moreover have GE.cones-map ?g ?Gχ = κ
proof
  fix j
  have ¬J.arr j ⟹ GE.cones-map ?g ?Gχ j = κ j
    using 1 Gχ.cone-axioms κ.extensionality by auto
  moreover have J.arr j ⟹ GE.cones-map ?g ?Gχ j = κ j
  proof −
    fix j
    assume j: J.arr j
    have GE.cones-map ?g ?Gχ j = G (χ j) ·D ?g
      using j 1 Gχ.cone-axioms mem-Collect-eq restrict-apply by auto
    also have ... = G (χ j ·C f) ·D Adj.η b
```

```
    using j f χ.preserves-hom [of j J.dom j J.cod j] D.comp-assoc by fastforce
  also have ... = G (E.cones-map f χ j) ·_D Adj.η b
  proof −
    have χ j ·_C f = Adj.ε (C.cod (E j)) ·_C F (κ j)
    proof −
      have χ j ·_C f = E.cones-map f χ j
      proof −
        have E.cone (C.cod f) χ
          using f χ.cone-axioms by blast
        thus ?thesis
          using χ.extensionality by simp
      qed
      also have ... = Adj.ε (C.cod (E j)) ·_C F (κ j)
        using j f by simp
      finally show ?thesis by blast
    qed
    thus ?thesis
      using f mem-Collect-eq restrict-apply Adj.F.extensionality by simp
  qed
  also have ... = (G (Adj.ε (C.cod (E j))) ·_D Adj.η (D.cod (GE.map j))) ·_D κ j
    using j f Adj.η.naturality [of κ j] D.comp-assoc by auto
  also have ... = D.cod (κ j) ·_D κ j
    using j Adj.ηε.triangle-G Adj.ε-in-terms-of-ψ Adj.εo-def
          Adj.η-in-terms-of-φ Adj.ηo-def Adj.unit-counit-G
    by fastforce
  also have ... = κ j
    using j D.comp-cod-arr by simp
  finally show GE.cones-map ?g ?Gχ j = κ j by metis
qed
ultimately show GE.cones-map ?g ?Gχ j = κ j by auto
qed
ultimately have «?g : b →_D G a» ∧ GE.cones-map ?g ?Gχ = κ by auto
```

It remains to be shown that g is the unique such arrow. Given any g' that transforms $G\chi$ into κ, its adjunct transforms χ into χ'. The adjunct of g' is therefore equal to f, which implies $g' = g$.

```
moreover have ⋀g'. «g' : b →_D G a» ∧ GE.cones-map g' ?Gχ = κ ⟹ g' = ?g
proof −
  fix g'
  assume g': «g' : b →_D G a» ∧ GE.cones-map g' ?Gχ = κ
  have 1: «ψ a g' : F b →_C a»
    using g' a ψ-in-hom by simp
  have 2: E.cones-map (ψ a g') χ = ?χ'
  proof
    fix j
    have ¬J.arr j ⟹ E.cones-map (ψ a g') χ j = ?χ' j
      using 1 χ.cone-axioms by auto
    moreover have J.arr j ⟹ E.cones-map (ψ a g') χ j = ?χ' j
    proof −
```

```
      fix j
      assume j: J.arr j
      have E.cones-map (ψ a g′) χ j = χ j ·C ψ a g′
        using 1 χ.cone-axioms χ.extensionality by auto
      also have ... = (χ j ·C Adj.ε a) ·C F g′
        using j a g′ Adj.ψ-in-terms-of-ε C.comp-assoc Adj.ε-def by auto
      also have ... = (Adj.ε (C.cod (E j)) ·C F (G (χ j))) ·C F g′
        using j a g′ Adj.ε.naturality [of χ j] by simp
      also have ... = Adj.ε (C.cod (E j)) ·C F (κ j)
        using j a g′ Gχ.cone-axioms C.comp-assoc by auto
      finally show E.cones-map (ψ a g′) χ j = ?χ′ j by (simp add: j)
    qed
    ultimately show E.cones-map (ψ a g′) χ j = ?χ′ j by auto
  qed
  have ψ a g′ = f
  proof −
    have ∃!f. «f : F b →C a» ∧ E.cones-map f χ = ?χ′
      using cone-χ′ χ.is-universal by simp
    moreover have «ψ a g′ : F b →C a» ∧ E.cones-map (ψ a g′) χ = ?χ′
      using 1 2 by simp
    ultimately show ?thesis
      using ex1E [of λf. «f : F b →C a» ∧ E.cones-map f χ = ?χ′ ψ a g′ = f]
      using 1 2 Adj.ε.extensionality C.null-is-zero(2) C.ex-un-null χ.cone-axioms f
            mem-Collect-eq restrict-apply
      by blast
  qed
  hence φ b (ψ a g′) = φ b f by auto
  hence g′ = φ b f using χ.ide-apex g′ by (simp add: φ-ψ)
  moreover have ?g = φ b f using f Adj.φ-in-terms-of-η κ.ide-apex Adj.η-def by auto
  ultimately show g′ = ?g by argo
 qed
 ultimately show ∃!g. «g : b →D G a» ∧ GE.cones-map g ?Gχ = κ by blast
qed
have GE.limit-cone (G a) ?Gχ ..
thus ?thesis by auto
qed

end
```

20.6 Special Kinds of Limits

20.6.1 Terminal Objects

An object of a category C is a terminal object if and only if it is a limit of the empty diagram in C.

```
locale empty-diagram =
  diagram J C D
for J :: 'j comp      (infixr ‹·J› 55)
```

```
and C :: 'c comp      (infixr ‹⋅› 55)
and D :: 'j ⇒ 'c +
assumes is-empty: ¬J.arr j
begin

  lemma has-as-limit-iff-terminal:
  shows has-as-limit a ⟷ C.terminal a
  proof
    assume a: has-as-limit a
    show C.terminal a
    proof
      have ∃χ. limit-cone a χ using a by auto
      from this obtain χ where χ: limit-cone a χ by blast
      interpret χ: limit-cone J C D a χ using χ by auto
      have cone-χ: cone a χ ..
      show C.ide a using χ.ide-apex by auto
      have 1: χ = (λj. C.null) using is-empty χ.extensionality by auto
      show ⋀a'. C.ide a' ⟹ ∃!f. «f : a' → a»
      proof −
        fix a'
        assume a': C.ide a'
        interpret A': constant-functor J C a'
          apply unfold-locales using a' by auto
        let ?χ' = λj. C.null
        have cone-χ': cone a' ?χ'
          using a' is-empty apply unfold-locales by auto
        hence ∃!f. «f : a' → a» ∧ cones-map f χ = ?χ'
          using χ.is-universal by force
        moreover have ⋀f. «f : a' → a» ⟹ cones-map f χ = ?χ'
          using 1 cone-χ by auto
        ultimately show ∃!f. «f : a' → a» by blast
      qed
    qed
    next
    assume a: C.terminal a
    show has-as-limit a
    proof −
      let ?χ = λj. C.null
      have C.ide a using a C.terminal-def by simp
      interpret A: constant-functor J C a
        apply unfold-locales using ‹C.ide a› by simp
      interpret χ: cone J C D a ?χ
        using ‹C.ide a› is-empty by (unfold-locales, auto)
      have cone-χ: cone a ?χ ..
      have 1: ⋀a' χ'. cone a' χ' ⟹ χ' = (λj. C.null)
      proof −
        fix a' χ'
        assume χ': cone a' χ'
        interpret χ': cone J C D a' χ' using χ' by auto
```

```
      show χ' = (λj. C.null)
        using is-empty χ'.extensionality by metis
    qed
    have limit-cone a ?χ
    proof
      fix a' χ'
      assume χ': cone a' χ'
      have 2: χ' = (λj. C.null) using 1 χ' by simp
      interpret χ': cone J C D a' χ' using χ' by auto
      have ∃!f. «f : a' → a» using a C.terminal-def χ'.ide-apex by simp
      moreover have ⋀f. «f : a' → a» ⟹ cones-map f ?χ = χ'
       using 1 2 cones-map-mapsto cone-χ χ'.cone-axioms mem-Collect-eq by blast
      ultimately show ∃!f. «f : a' → a» ∧ cones-map f (λj. C.null) = χ'
        by blast
    qed
    thus ?thesis by auto
  qed
qed

end
```

20.6.2 Products

A *product* in a category C is a limit of a discrete diagram in C.

```
locale discrete-diagram =
  J: category J +
  diagram J C D
for J :: 'j comp      (infixr ‹·J› 55)
and C :: 'c comp      (infixr ‹·› 55)
and D :: 'j ⇒ 'c +
assumes is-discrete: J.arr = J.ide
begin

  abbreviation mkCone
  where mkCone F ≡ (λj. if J.arr j then F j else C.null)

  lemma cone-mkCone:
  assumes C.ide a and ⋀j. J.arr j ⟹ «F j : a → D j»
  shows cone a (mkCone F)
  proof −
    interpret A: constant-functor J C a
      using assms(1) by unfold-locales auto
    show cone a (mkCone F)
      using assms(2) is-discrete
      apply unfold-locales
        apply auto
       apply (metis C.in-homE C.comp-cod-arr)
      using C.comp-arr-ide by fastforce
  qed
```

```
  lemma mkCone-cone:
  assumes cone a π
  shows mkCone π = π
  proof −
    interpret π: cone J C D a π
      using assms by auto
    show mkCone π = π using π.extensionality by auto
  qed

end
```

The following locale defines a discrete diagram in a category C, given an index set I and a function D mapping I to objects of C. Here we obtain the diagram shape J using a discrete category construction that allows us to directly identify the objects of J with the elements of I, however this construction can only be applied in case the set I is not the universe of its element type.

```
locale discrete-diagram-from-map =
  J: discrete-category I null +
  C: category C
for I :: 'i set
and C :: 'c comp      (infixr ⟨·⟩ 55)
and D :: 'i ⇒ 'c
and null :: 'i +
assumes maps-to-ide: i ∈ I ⟹ C.ide (D i)
begin

  definition map
  where map j ≡ if J.arr j then D j else C.null

end

sublocale discrete-diagram-from-map ⊆ discrete-diagram J.comp C map
  using map-def maps-to-ide J.arr-char J.Null-not-in-Obj J.null-char
  by unfold-locales auto

locale product-cone =
  J: category J +
  C: category C +
  D: discrete-diagram J C D +
  limit-cone J C D a π
for J :: 'j comp      (infixr ⟨·J⟩ 55)
and C :: 'c comp      (infixr ⟨·⟩ 55)
and D :: 'j ⇒ 'c
and a :: 'c
and π :: 'j ⇒ 'c
begin

  lemma is-cone:
```

shows *D.cone a π* **..**

The following versions of *is-universal* and *induced-arrowI* from the *limit-cone* locale are specialized to the case in which the underlying diagram is a product diagram.

lemma *is-universal′*:
assumes *C.ide b* **and** $\bigwedge j.\ J.arr\ j \Longrightarrow$ «*F j*: *b* → *D j*»
shows $\exists!f.$ «*f* : *b* → *a*» ∧ (∀ *j*. *J.arr j* ⟶ *π j* · *f* = *F j*)
proof −
let *?χ* = *D.mkCone F*
interpret *B*: *constant-functor J C b*
using *assms*(*1*) **by** *unfold-locales auto*
have *cone-χ*: *D.cone b ?χ*
using *assms D.is-discrete D.cone-mkCone* **by** *blast*
interpret *χ*: *cone J C D b ?χ* **using** *cone-χ* **by** *auto*
have $\exists!f.$ «*f* : *b* → *a*» ∧ *D.cones-map f π* = *?χ*
using *cone-χ is-universal* **by** *force*
moreover have
$\bigwedge f.$ «*f* : *b* → *a*» ⟹ *D.cones-map f π* = *?χ* ⟷ (∀ *j*. *J.arr j* ⟶ *π j* · *f* = *F j*)
proof −
fix *f*
assume *f*: «*f* : *b* → *a*»
show *D.cones-map f π* = *?χ* ⟷ (∀ *j*. *J.arr j* ⟶ *π j* · *f* = *F j*)
proof
assume *1*: *D.cones-map f π* = *?χ*
show ∀ *j*. *J.arr j* ⟶ *π j* · *f* = *F j*
proof −
have $\bigwedge j.$ *J.arr j* ⟹ *π j* · *f* = *F j*
proof −
fix *j*
assume *j*: *J.arr j*
have *π j* · *f* = *D.cones-map f π j*
using *j f cone-axioms* **by** *force*
also have ... = *F j* **using** *j 1* **by** *simp*
finally show *π j* · *f* = *F j* **by** *auto*
qed
thus *?thesis* **by** *auto*
qed
next
assume *1*: ∀ *j*. *J.arr j* ⟶ *π j* · *f* = *F j*
show *D.cones-map f π* = *?χ*
using *1 f is-cone χ.extensionality D.is-discrete is-cone cone-χ* **by** *auto*
qed
qed
ultimately show *?thesis* **by** *blast*
qed

abbreviation *induced-arrow′* :: *′c* ⇒ (*′j* ⇒ *′c*) ⇒ *′c*
where *induced-arrow′ b F* ≡ *induced-arrow b* (*D.mkCone F*)

```
  lemma induced-arrowI':
  assumes C.ide b and ⋀j. J.arr j ⟹ «F j : b → D j»
  shows ⋀j. J.arr j ⟹ π j · induced-arrow' b F = F j
  proof −
    interpret B: constant-functor J C b
      using assms(1) by unfold-locales auto
    interpret χ: cone J C D b ‹D.mkCone F›
      using assms D.cone-mkCone by blast
    have cone-χ: D.cone b (D.mkCone F) ..
    hence 1: D.cones-map (induced-arrow' b F) π = D.mkCone F
      using induced-arrowI by blast
    fix j
    assume j: J.arr j
    have π j · induced-arrow' b F = D.cones-map (induced-arrow' b F) π j
      using induced-arrowI(1) cone-χ is-cone extensionality by force
    also have ... = F j
      using j 1 by auto
    finally show π j · induced-arrow' b F = F j
      by auto
  qed

end

context discrete-diagram
begin

  lemma product-coneI:
  assumes limit-cone a π
  shows product-cone J C D a π
    by (meson assms discrete-diagram-axioms functor-axioms functor-def product-cone.intro)

end

context category
begin

  definition has-as-product
  where has-as-product J D a ≡ (∃π. product-cone J C D a π)

  lemma product-is-ide:
  assumes has-as-product J D a
  shows ide a
  proof −
    obtain π where π: product-cone J C D a π
      using assms has-as-product-def by blast
    interpret π: product-cone J C D a π
      using π by auto
    show ?thesis using π.ide-apex by auto
  qed
```

A category has I-indexed products for an $'i$-set I if every I-indexed discrete diagram has a product. In order to reap the benefits of being able to directly identify the elements of a set I with the objects of discrete category it generates (thereby avoiding the use of coercion maps), it is necessary to assume that $I \neq UNIV$. If we want to assert that a category has products indexed by the universe of some type $'i$, we have to pass to a larger type, such as $'i\ option$.

definition *has-products*
where *has-products* $(I :: 'i\ set) \equiv$
$I \neq UNIV \wedge$
$(\forall J\ D.\ discrete\text{-}diagram\ J\ C\ D \wedge Collect\ (partial\text{-}composition.arr\ J) = I$
$\longrightarrow (\exists a.\ has\text{-}as\text{-}product\ J\ D\ a))$

lemma *ex-productE*:
assumes $\exists a.\ has\text{-}as\text{-}product\ J\ D\ a$
obtains $a\ \pi$ **where** *product-cone J C D a* π
using *assms has-as-product-def someI-ex* [*of* $\lambda a.\ has\text{-}as\text{-}product\ J\ D\ a$] **by** *metis*

lemma *has-products-if-has-limits*:
assumes *has-limits* (*undefined* :: $'j$) **and** $I \neq (UNIV :: 'j\ set)$
shows *has-products I*
proof (*unfold has-products-def*, *intro conjI allI impI*)
show $I \neq UNIV$ **by** *fact*
fix *J D*
assume *D*: *discrete-diagram J C D* $\wedge$ *Collect* (*partial-composition.arr J*) = *I*
interpret *D*: *discrete-diagram J C D*
using *D* **by** *simp*
have *1*: $\exists a.$ *D.has-as-limit a*
using *assms D D.diagram-axioms D.J.category-axioms*
by (*simp add*: *has-limits-of-shape-def has-limits-def*)
show $\exists a.$ *has-as-product J D a*
using *1 has-as-product-def D.product-coneI* **by** *blast*
qed

lemma *has-finite-products-if-has-finite-limits*:
assumes $\bigwedge J :: 'j\ comp.\ (finite\ (Collect\ (partial\text{-}composition.arr\ J))) \Longrightarrow has\text{-}limits\text{-}of\text{-}shape\ J$
and *finite* ($I :: 'j\ set$) **and** $I \neq UNIV$
shows *has-products I*
proof (*unfold has-products-def*, *intro conjI allI impI*)
show $I \neq UNIV$ **by** *fact*
fix *J D*
assume *D*: *discrete-diagram J C D* $\wedge$ *Collect* (*partial-composition.arr J*) = *I*
interpret *D*: *discrete-diagram J C D*
using *D* **by** *simp*
have *1*: $\exists a.$ *D.has-as-limit a*
using *assms D has-limits-of-shape-def D.diagram-axioms* **by** *auto*
show $\exists a.$ *has-as-product J D a*
using *1 has-as-product-def D.product-coneI* **by** *blast*

qed

lemma *has-products-preserved-by-bijection*:
assumes *has-products I* **and** *bij-betw φ I I'* **and** $I' \neq UNIV$
shows *has-products I'*
proof (*unfold has-products-def*, *intro conjI allI impI*)
show $I' \neq UNIV$ **by** *fact*
show $\bigwedge J'\ D'.$ *discrete-diagram J' C D'* $\wedge$ *Collect* (*partial-composition.arr J'*) $= I'$
$\Longrightarrow \exists a.$ *has-as-product J' D' a*
proof −
fix *J' D'*
assume *1*: *discrete-diagram J' C D'* $\wedge$ *Collect* (*partial-composition.arr J'*) $= I'$
interpret *J'*: *category J'*
using *1* **by** (*simp add*: *discrete-diagram-def*)
interpret *D'*: *discrete-diagram J' C D'*
using *1* **by** *simp*
interpret *J*: *discrete-category I* ‹*SOME x. x* $\notin$ *I*›
using *assms has-products-def* [*of I*] *someI-ex* [*of* $\lambda x.\ x \notin I$]
by *unfold-locales auto*
have *2*: *Collect J.arr* $= I \wedge$ *Collect J'.arr* $= I'$
using *1* **by** *auto*
have *φ*: *bij-betw φ* (*Collect J.arr*) (*Collect J'.arr*)
using *2 assms*(*2*) **by** *simp*
let *?φ* $= \lambda j.$ *if J.arr j then φ j else J'.null*
let *?φ'* $= \lambda j'.$ *if J'.arr j' then the-inv-into I φ j' else J.null*
interpret *φ*: *functor J.comp J' ?φ*
proof −
have *φ* ‘ *I* $= I'$
using *φ 2 bij-betw-def* [*of φ I I'*] **by** *simp*
hence $\bigwedge j.$ *J.arr j* $\Longrightarrow$ *J'.arr* (*?φ j*)
using *1 D'.is-discrete* **by** *auto*
thus *functor J.comp J' ?φ*
using *D'.is-discrete J.is-discrete J.seqE*
by *unfold-locales auto*
qed
interpret *φ'*: *functor J' J.comp ?φ'*
proof −
have *the-inv-into I φ* ‘ *I'* $= I$
using *assms*(*2*) *φ bij-betw-the-inv-into bij-betw-imp-surj-on* **by** *metis*
hence $\bigwedge j'.$ *J'.arr j'* $\Longrightarrow$ *J.arr* (*?φ' j'*)
using *2 D'.is-discrete J.is-discrete* **by** *auto*
thus *functor J' J.comp ?φ'*
using *D'.is-discrete J.is-discrete J'.seqE*
by *unfold-locales auto*
qed
let *?D* $= \lambda i.$ *D'* (*φ i*)
interpret *D*: *discrete-diagram-from-map I C ?D* ‹*SOME j. j* $\notin$ *I*›
using *assms 1 D'.is-discrete bij-betw-imp-surj-on φ.preserves-ide*
by *unfold-locales auto*

obtain a **where** a: *has-as-product J.comp D.map a*
using *assms D.discrete-diagram-axioms has-products-def* [*of I*] **by** *auto*
obtain π **where** π: *product-cone J.comp C D.map a* π
using *a has-as-product-def* **by** *blast*
interpret π: *product-cone J.comp C D.map a* π
using π **by** *simp*
let *?π′* = π *o ?φ′*
interpret A: *constant-functor J′ C a*
using π.*ide-apex* **by** *unfold-locales simp*
interpret π': *natural-transformation J′ C A.map D′ ?π′*
proof −
have π.*A.map* ∘ *?φ′* = *A.map*
using φ *A.map-def* φ'.*preserves-arr* π.*A.extensionality J.not-arr-null* **by** *auto*
moreover have *D.map* ∘ *?φ′* = *D′*
proof
fix *j′*
have *J′.arr j′* ⟹ (*D.map* ∘ *?φ′*) *j′* = *D′ j′*
proof −
assume *2*: *J′.arr j′*
have *3*: *inj-on* φ *I*
using *assms*(*2*) *bij-betw-imp-inj-on* **by** *auto*
have φ ‘ *I* = *I′*
by (*metis* (*no-types*) *assms*(*2*) *bij-betw-imp-surj-on*)
hence φ ‘ *I* = *Collect J′.arr*
using *1* **by** *force*
thus *?thesis*
using *2 3 D.map-def* φ'.*preserves-arr f-the-inv-into-f* **by** *fastforce*
qed
moreover have ¬ *J′.arr j′* ⟹ (*D.map* ∘ *?φ′*) *j′* = *D′ j′*
using *D.extensionality D′.extensionality*
by (*simp add*: *J.Null-not-in-Obj J.null-char*)
ultimately show (*D.map* ∘ *?φ′*) *j′* = *D′ j′* **by** *blast*
qed
ultimately show *natural-transformation J′ C A.map D′ ?π′*
using π.*natural-transformation-axioms* φ'.*as-nat-trans.natural-transformation-axioms*
horizontal-composite [*of J′ J.comp ?φ′ ?φ′ ?φ′ C* π.*A.map D.map* π]
by *simp*
qed
interpret π': *cone J′ C D′ a ?π′* **..**
interpret π': *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** *simp*
show ∃!*f*. «*f* : *a′* → *a*» ∧ *D′.cones-map f* (π ∘ *?φ′*) = χ'
proof −
let *?χ* = χ' *o ?φ*
interpret *A′*: *constant-functor J.comp C a′*

```
  using χ'.ide-apex by unfold-locales simp
interpret χ: natural-transformation J.comp C A'.map D.map ?χ
proof −
  have χ'.A.map ∘ ?φ = A'.map
    using φ φ.preserves-arr A'.map-def χ'.A.extensionality by auto
  moreover have D' ∘ ?φ = D.map
    using φ D.map-def D'.extensionality by auto
  ultimately show natural-transformation J.comp C A'.map D.map ?χ
    using χ'.natural-transformation-axioms
          φ.as-nat-trans.natural-transformation-axioms
          horizontal-composite [of J.comp J' ?φ ?φ ?φ C χ'.A.map D' χ']
    by simp
qed
interpret χ: cone J.comp C D.map a' ?χ ..
have ∗: ∃!f. «f : a' → a» ∧ D.cones-map f π = ?χ
  using π.is-universal χ.cone-axioms by simp
show ∃!f. «f : a' → a» ∧ D'.cones-map f ?π' = χ'
proof −
  have ∃f. «f : a' → a» ∧ D'.cones-map f ?π' = χ'
  proof −
    obtain f where f: «f : a' → a» ∧ D.cones-map f π = ?χ
      using ∗ by blast
    have D'.cones-map f ?π' = χ'
    proof
      fix j'
      show D'.cones-map f ?π' j' = χ' j'
      proof (cases J'.arr j')
        assume j': ¬ J'.arr j'
        show D'.cones-map f ?π' j' = χ' j'
          using f j' χ'.extensionality π'.cone-axioms by auto
        next
        assume j': J'.arr j'
        show D'.cones-map f ?π' j' = χ' j'
        proof −
          have D'.cones-map f ?π' j' = π (the-inv-into I φ j') · f
            using f j' π'.cone-axioms by auto
          also have ... = D.cones-map f π (the-inv-into I φ j')
          proof −
            have arr f ∧ dom f = a' ∧ cod f = a
              using f by blast
            thus ?thesis
              using φ'.preserves-arr π.is-cone j' by auto
          qed
          also have ... = (χ' ∘ ?φ) (the-inv-into I φ j')
            using f by simp
          also have ... = χ' j'
            using assms(2) j' 2 bij-betw-def [of φ I I'] bij-betw-imp-inj-on
                  φ'.preserves-arr f-the-inv-into-f
            by fastforce
```

```
      finally show ?thesis by simp
    qed
  qed
 qed
 thus ?thesis using f by blast
qed
moreover have ⋀f f'. ⟦ «f : a' → a»; D'.cones-map f ?π' = χ';
                       «f' : a' → a»; D'.cones-map f' ?π' = χ' ⟧
                         ⟹ f = f'
proof −
 fix f f'
 assume f: «f : a' → a» and f': «f' : a' → a»
 and fχ': D'.cones-map f ?π' = χ' and f'χ': D'.cones-map f' ?π' = χ'
 have D.cones-map f π = χ' ∘ ?φ ∧ D.cones-map f' π = χ' o ?φ
 proof (intro conjI)
  show D.cones-map f π = χ' ∘ ?φ
  proof
   fix j
   have ¬ J.arr j ⟹ D.cones-map f π j = (χ' ∘ ?φ) j
    using f f χ' π.cone-axioms χ.extensionality by auto
   moreover have J.arr j ⟹ D.cones-map f π j = (χ' ∘ ?φ) j
   proof −
    assume j: J.arr j
    have 1: j = the-inv-into I φ (φ j)
     using assms(2) j φ the-inv-into-f-f bij-betw-imp-inj-on J.arr-char
     by metis
    have D.cones-map f π j = D.cones-map f π (the-inv-into I φ (φ j))
     using 1 by simp
    also have ... = (χ' ∘ ?φ) j
     using f j f χ' 1 π.cone-axioms π'.cone-axioms φ.preserves-arr by auto
    finally show D.cones-map f π j = (χ' ∘ ?φ) j by blast
   qed
   ultimately show D.cones-map f π j = (χ' ∘ ?φ) j by blast
  qed
  show D.cones-map f' π = χ' ∘ ?φ
  proof
   fix j
   have ¬ J.arr j ⟹ D.cones-map f' π j = (χ' ∘ ?φ) j
    using f' f χ' π.cone-axioms χ.extensionality by auto
   moreover have J.arr j ⟹ D.cones-map f' π j = (χ' ∘ ?φ) j
   proof −
    assume j: J.arr j
    have 1: j = the-inv-into I φ (φ j)
     using assms(2) j φ the-inv-into-f-f bij-betw-imp-inj-on J.arr-char
     by metis
    have D.cones-map f' π j = D.cones-map f' π (the-inv-into I φ (φ j))
     using 1 by simp
    also have ... = (χ' ∘ ?φ) j
     using f' j f'χ' 1 π.cone-axioms π'.cone-axioms φ.preserves-arr by auto
```

```
            finally show D.cones-map f' π j = (χ' ∘ ?φ) j by blast
          qed
          ultimately show D.cones-map f' π j = (χ' ∘ ?φ) j by blast
        qed
      qed
      thus f = f'
        using f f' * by auto
    qed
    ultimately show ?thesis by blast
  qed
 qed
 qed

lemma ide-is-unary-product:
assumes ide a
shows ⋀m n :: nat. m ≠ n ⟹ has-as-product (discrete-category.comp {m :: nat} (n :: nat))
                                          (λi. if i = m then a else null) a
proof −
  fix m n :: nat
  assume neq: m ≠ n
  have {m :: nat} ≠ UNIV
  proof −
    have finite {m :: nat} by simp
    moreover have ¬finite (UNIV :: nat set) by simp
    ultimately show ?thesis by fastforce
  qed
  interpret J: discrete-category {m :: nat} ‹n :: nat›
    using neq ‹{m :: nat} ≠ UNIV› by unfold-locales auto
  let ?D = λi. if i = m then a else null
  interpret D: discrete-diagram J.comp C ?D
    apply unfold-locales
    using assms J.null-char neq
        apply auto
    by metis
  interpret A: constant-functor J.comp C a
    using assms by unfold-locales auto
  show has-as-product J.comp ?D a
  proof (unfold has-as-product-def)
    let ?π = λi :: nat. if i = m then a else null
    interpret π: natural-transformation J.comp C A.map ?D ?π
      using assms J.arr-char J.dom-char J.cod-char
      by unfold-locales auto
    interpret π: cone J.comp C ?D a ?π ..
    interpret π: product-cone J.comp C ?D a ?π
```

```
    proof
      fix a' χ'
      assume χ': D.cone a' χ'
      interpret χ': cone J.comp C ?D a' χ' using χ' by auto
      show ∃!f. «f : a' → a» ∧ D.cones-map f ?π = χ'
      proof
        show «χ' m : a' → a» ∧ D.cones-map (χ' m) ?π = χ'
        proof
          show 1: «χ' m : a' → a»
            using χ'.preserves-hom χ'.A.map-def J.arr-char J.dom-char J.cod-char
            by auto
          show D.cones-map (χ' m) ?π = χ'
          proof
            fix j
            show D.cones-map (χ' m) (λi. if i = m then a else null) j = χ' j
              using J.arr-char J.dom-char J.cod-char J.ide-char π.cone-axioms comp-cod-arr
              apply (cases j = m)
               apply simp
              using χ'.extensionality by simp
          qed
        qed
        show ⋀f. «f : a' → a» ∧ D.cones-map f ?π = χ' ⟹ f = χ' m
        proof −
          fix f
          assume f: «f : a' → a» ∧ D.cones-map f ?π = χ'
          show f = χ' m
            using assms f χ'.preserves-hom J.arr-char J.dom-char J.cod-char π.cone-axioms
                  comp-cod-arr
            by auto
        qed
      qed
    qed
    show ∃π. product-cone J.comp C (λi. if i = m then a else null) a π
      using π.product-cone-axioms by blast
  qed
qed

lemma has-unary-products:
assumes card I = 1 and I ≠ UNIV
shows has-products I
proof −
  obtain i where i: I = {i}
    using assms card-1-singletonE by blast
  obtain n where n: n ∉ I
    using assms by auto
  have has-products {1 :: nat}
  proof (unfold has-products-def, intro conjI)
    show {1 :: nat} ≠ UNIV by auto
    show ∀(J :: nat comp) D.
```

```
        discrete-diagram J (·) D ∧ Collect (partial-composition.arr J) = {1}
          ⟶ (∃ a. has-as-product J D a)
    proof
      fix J :: nat comp
      show ∀ D. discrete-diagram J (·) D ∧ Collect (partial-composition.arr J) = {1}
              ⟶ (∃ a. has-as-product J D a)
      proof (intro allI impI)
        fix D
        assume D: discrete-diagram J (·) D ∧ Collect (partial-composition.arr J) = {1}
        interpret D: discrete-diagram J C D
          using D by auto
        interpret J: discrete-category ‹{1 :: nat}› D.J.null
          by (metis D D.J.not-arr-null discrete-category.intro mem-Collect-eq)
        have 1: has-as-product J.comp D (D 1)
        proof −
          have has-as-product J.comp (λi. if i = 1 then D 1 else null) (D 1)
            using ide-is-unary-product D.preserves-ide D.is-discrete D J.null-char
            by (metis J.Null-not-in-Obj insertCI mem-Collect-eq)
          moreover have D = (λi. if i = 1 then D 1 else null)
          proof
            fix j
            show D j = (if j = 1 then D 1 else null)
              using D D.extensionality by auto
          qed
          ultimately show ?thesis by simp
        qed
        moreover have J = J.comp
        proof −
          have ⋀j j'. J j j' = J.comp j j'
          proof −
            fix j j'
            show J j j' = J.comp j j'
              using J.comp-char D.is-discrete D
              by (metis D.J.category-axioms D.J.ext D.J.ide-def J.null-char
                        category.seqE mem-Collect-eq)
          qed
          thus ?thesis by auto
        qed
        ultimately show ∃ a. has-as-product J D a by auto
      qed
    qed
  qed
  moreover have bij-betw (λk. if k = 1 then i else n) {1 :: nat} I
    using i by (intro bij-betwI') auto
  ultimately show has-products I
    using assms has-products-preserved-by-bijection by blast
qed

end
```

20.6.3 Equalizers

An *equalizer* in a category C is a limit of a parallel pair of arrows in C.

```
locale parallel-pair-diagram =
  J: parallel-pair +
  C: category C
for C :: 'c comp      (infixr ‹·› 55)
and f0 :: 'c
and f1 :: 'c +
assumes is-parallel: C.par f0 f1
begin

  no-notation J.comp   (infixr ‹·› 55)
  notation J.comp      (infixr ‹·J› 55)

  definition map
  where map ≡ (λj. if j = J.Zero then C.dom f0
                   else if j = J.One then C.cod f0
                   else if j = J.j0 then f0
                   else if j = J.j1 then f1
                   else C.null)

  lemma map-simp:
  shows map J.Zero = C.dom f0
  and map J.One = C.cod f0
  and map J.j0 = f0
  and map J.j1 = f1
  proof −
    show map J.Zero = C.dom f0
      using map-def by metis
    show map J.One = C.cod f0
      using map-def J.Zero-not-eq-One by metis
    show map J.j0 = f0
      using map-def J.Zero-not-eq-j0 J.One-not-eq-j0 by metis
    show map J.j1 = f1
      using map-def J.Zero-not-eq-j1 J.One-not-eq-j1 J.j0-not-eq-j1 by metis
  qed

end

sublocale parallel-pair-diagram ⊆ diagram J.comp C map
  apply unfold-locales
      apply (simp add: J.arr-char map-def)
  using map-def is-parallel J.arr-char J.cod-simp J.dom-simp
     apply auto[2]
proof −
  show 1: ⋀j. J.arr j ⟹ C.cod (map j) = map (J.cod j)
    using is-parallel map-simp(1−4) J.arr-char by auto
  next
```

```
  fix j j'
  assume jj': J.seq j' j
  show map (j' ·_J j) = map j' · map j
  proof −
    have 1: (j = J.Zero ∧ j' ≠ J.One) ∨ (j ≠ J.Zero ∧ j' = J.One)
      using jj' J.seq-char by blast
    moreover have j = J.Zero ∧ j' ≠ J.One ⟹ ?thesis
      by (metis (no-types, lifting) C.arr-dom-iff-arr C.comp-arr-dom C.dom-dom
          J.comp-simp(1) jj' map-simp(1,3−4) J.arr-char is-parallel)
    moreover have j ≠ J.Zero ∧ j' = J.One ⟹ ?thesis
      by (metis (no-types, lifting) C.comp-arr-dom C.comp-cod-arr C.seqE J.comp-char jj'
          map-simp(2−4) J.arr-char is-parallel)
    ultimately show ?thesis by blast
  qed
qed

context parallel-pair-diagram
begin

  definition mkCone
  where mkCone e ≡ λj. if J.arr j then if j = J.Zero then e else f0 · e else C.null

  abbreviation is-equalized-by
  where is-equalized-by e ≡ C.seq f0 e ∧ f0 · e = f1 · e

  abbreviation has-as-equalizer
  where has-as-equalizer e ≡ limit-cone (C.dom e) (mkCone e)

  lemma cone-mkCone:
  assumes is-equalized-by e
  shows cone (C.dom e) (mkCone e)
  proof −
    interpret E: constant-functor J.comp C ‹C.dom e›
      using assms by unfold-locales auto
    show cone (C.dom e) (mkCone e)
    proof (unfold-locales)
      show ⋀j. ¬ J.arr j ⟹ mkCone e j = C.null
        using assms mkCone-def by auto
      show ⋀j. J.arr j ⟹ C.arr (mkCone e j)
        using assms mkCone-def by auto
      show ⋀j. J.arr j ⟹ map j · mkCone e (J.dom j) = mkCone e j
        using assms mkCone-def C.comp-cod-arr extensionality map-def is-parallel
        apply auto
        using parallel-pair.arr-char by auto
      show ⋀j. J.arr j ⟹ mkCone e (J.cod j) · E.map j = mkCone e j
      proof −
        fix j
        assume j: J.arr j
        have 1: j = J.Zero ∨ map j · mkCone e (J.dom j) = mkCone e j
```

```
      using assms j mkCone-def C.cod-comp
      by (metis (no-types, lifting) C.comp-cod-arr J.arr-char J.dom-char map-def
          J.dom-simp(2))
    thus mkCone e (J.cod j) · E.map j = mkCone e j
      using j C.comp-arr-dom assms mkCone-def apply auto
      by (metis (no-types, lifting) J.Zero-not-eq-One parallel-pair.arr-char
        parallel-pair.cod-simp(2−4))
   qed
  qed
qed

lemma is-equalized-by-cone:
assumes cone a χ
shows is-equalized-by (χ (J.Zero))
proof −
  interpret χ: cone J.comp C map a χ
    using assms by auto
  show ?thesis
    by (metis (no-types, lifting) J.arr-char J.cod-char cone-def
        χ.component-in-hom χ.naturality1 χ.naturality assms C.in-homE
        constant-functor.map-simp J.dom-simp(3−4) map-simp(3−4))
qed

lemma mkCone-cone:
assumes cone a χ
shows mkCone (χ J.Zero) = χ
proof −
  interpret χ: cone J.comp C map a χ
    using assms by auto
  have 1: is-equalized-by (χ J.Zero)
    using assms is-equalized-by-cone by blast
  show ?thesis
  proof
    fix j
    have j = J.Zero ⟹ mkCone (χ J.Zero) j = χ j
      using mkCone-def χ.extensionality by simp
    moreover have j = J.One ∨ j = J.j0 ∨ j = J.j1 ⟹ mkCone (χ J.Zero) j = χ j
      using J.arr-char J.cod-char J.dom-char J.seq-char mkCone-def
          χ.naturality1 χ.naturality2 χ.A.map-simp map-def
      by (metis (no-types, lifting) J.Zero-not-eq-j0 J.dom-simp(2))
    ultimately have J.arr j ⟹ mkCone (χ J.Zero) j = χ j
      using J.arr-char by auto
    thus mkCone (χ J.Zero) j = χ j
      using mkCone-def χ.extensionality by fastforce
  qed
qed

end
```

```
locale equalizer-cone =
  J: parallel-pair +
  C: category C +
  D: parallel-pair-diagram C f0 f1 +
  limit-cone J.comp C D.map C.dom e D.mkCone e
for C :: 'c comp      (infixr ‹·› 55)
and f0 :: 'c
and f1 :: 'c
and e :: 'c
begin

  lemma equalizes:
  shows D.is-equalized-by e
  proof
    show C.seq f0 e
    proof (intro C.seqI)
      show C.arr e using ide-apex C.arr-dom-iff-arr by fastforce
      show C.arr f0
        using D.map-simp D.preserves-arr J.arr-char by metis
      show C.dom f0 = C.cod e
        using J.arr-char J.ide-char D.mkCone-def D.map-simp preserves-cod [of J.Zero]
        by auto
    qed
    show f0 · e = f1 · e
      using D.map-simp D.mkCone-def J.arr-char naturality [of J.j0] naturality [of J.j1]
      by force
  qed

  lemma is-universal':
  assumes D.is-equalized-by e'
  shows ∃!h. «h : C.dom e' → C.dom e» ∧ e · h = e'
  proof −
    have D.cone (C.dom e') (D.mkCone e')
      using assms D.cone-mkCone by blast
    moreover have 0: D.cone (C.dom e) (D.mkCone e) ..
    ultimately have 1: ∃!h. «h : C.dom e' → C.dom e» ∧
                            D.cones-map h (D.mkCone e) = D.mkCone e'
      using is-universal [of C.dom e' D.mkCone e'] by auto
    have 2: ⋀h. «h : C.dom e' → C.dom e» ⟹
                D.cones-map h (D.mkCone e) = D.mkCone e' ⟷ e · h = e'
    proof −
      fix h
      assume h: «h : C.dom e' → C.dom e»
      show D.cones-map h (D.mkCone e) = D.mkCone e' ⟷ e · h = e'
      proof
        assume 3: D.cones-map h (D.mkCone e) = D.mkCone e'
        show e · h = e'
        proof −
          have e' = D.mkCone e' J.Zero
```

```
        using D.mkCone-def J.arr-char by simp
      also have ... = D.cones-map h (D.mkCone e) J.Zero
        using 3 by simp
      also have ... = e ⋅ h
        using 0 h D.mkCone-def J.arr-char by auto
      finally show ?thesis by auto
    qed
    next
    assume e': e ⋅ h = e'
    show D.cones-map h (D.mkCone e) = D.mkCone e'
    proof
      fix j
      have ¬J.arr j ⟹ D.cones-map h (D.mkCone e) j = D.mkCone e' j
        using h cone-axioms D.mkCone-def by auto
      moreover have j = J.Zero ⟹ D.cones-map h (D.mkCone e) j = D.mkCone e' j
        using h e' is-cone D.mkCone-def J.arr-char [of J.Zero] by force
      moreover have
          J.arr j ∧ j ≠ J.Zero ⟹ D.cones-map h (D.mkCone e) j = D.mkCone e' j
        using C.comp-assoc D.mkCone-def is-cone e' h by auto
      ultimately show D.cones-map h (D.mkCone e) j = D.mkCone e' j by blast
    qed
  qed
 qed
 thus ?thesis using 1 by blast
qed

lemma induced-arrowI':
assumes D.is-equalized-by e'
shows «induced-arrow (C.dom e') (D.mkCone e') : C.dom e' → C.dom e»
and e ⋅ induced-arrow (C.dom e') (D.mkCone e') = e'
proof −
 interpret A': constant-functor J.comp C ‹C.dom e'›
   using assms by (unfold-locales, auto)
 have cone: D.cone (C.dom e') (D.mkCone e')
   using assms D.cone-mkCone [of e'] by blast
 have e ⋅ induced-arrow (C.dom e') (D.mkCone e') =
       D.cones-map (induced-arrow (C.dom e') (D.mkCone e')) (D.mkCone e) J.Zero
   using cone induced-arrowI(1) D.mkCone-def J.arr-char is-cone by force
 also have ... = e'
 proof −
   have D.cones-map (induced-arrow (C.dom e') (D.mkCone e')) (D.mkCone e) =
        D.mkCone e'
     using cone induced-arrowI by blast
   thus ?thesis
     using J.arr-char D.mkCone-def by simp
 qed
 finally have 1: e ⋅ induced-arrow (C.dom e') (D.mkCone e') = e'
   by auto
 show «induced-arrow (C.dom e') (D.mkCone e') : C.dom e' → C.dom e»
```

```
    using 1 cone induced-arrowI by simp
  show e · induced-arrow (C.dom e') (D.mkCone e') = e'
    using 1 cone induced-arrowI by simp
qed

end

context category
begin

  definition has-as-equalizer
  where has-as-equalizer f0 f1 e ≡ par f0 f1 ∧ parallel-pair-diagram.has-as-equalizer C f0 f1 e

  definition has-equalizers
  where has-equalizers = (∀ f0 f1. par f0 f1 ⟶ (∃ e. has-as-equalizer f0 f1 e))

  lemma has-as-equalizerI [intro]:
  assumes par f g and seq f e and f · e = g · e
  and ⋀e'. ⟦seq f e'; f · e' = g · e'⟧ ⟹ ∃!h. e · h = e'
  shows has-as-equalizer f g e
  proof (unfold has-as-equalizer-def, intro conjI)
    show arr f and arr g and dom f = dom g and cod f = cod g
      using assms(1) by auto
    interpret J: parallel-pair .
    interpret D: parallel-pair-diagram C f g
      using assms(1) by unfold-locales
    show D.has-as-equalizer e
    proof −
      let ?χ = D.mkCone e
      let ?a = dom e
      interpret χ: cone J.comp C D.map ?a ?χ
        using assms(2−3) D.cone-mkCone [of e] by simp
      interpret χ: limit-cone J.comp C D.map ?a ?χ
      proof
        fix a' χ'
        assume χ': D.cone a' χ'
        interpret χ': cone J.comp C D.map a' χ'
          using χ' by blast
        have seq f (χ' J.Zero)
          using J.ide-char J.arr-char χ'.preserves-hom
          by (metis (no-types, lifting) D.map-simp(3) χ'.naturality1
            χ'.natural-transformation-axioms natural-transformation.preserves-reflects-arr
            parallel-pair.dom-simp(3))
        moreover have f · (χ' J.Zero) = g · (χ' J.Zero)
          using χ' D.is-equalized-by-cone by blast
        ultimately have 1: ∃!h. e · h = χ' J.Zero
          using assms by blast
        obtain h where h: e · h = χ' J.Zero
          using 1 by blast
```

```
      have 2: D.is-equalized-by e
      using assms(2−3) by blast
      have «h : a′ → dom e» ∧ D.cones-map h (D.mkCone e) = χ′
      proof
        show 3: «h : a′ → dom e»
          using h χ′.preserves-dom
          by (metis (no-types, lifting) χ′.component-in-hom ‹seq f (χ′ J.Zero)›
            category.has-codomain-iff-arr category.seqE category-axioms cod-in-codomains
            domains-comp in-hom-def parallel-pair.arr-char)
        show D.cones-map h (D.mkCone e) = χ′
        proof
          fix j
          have D.cone (cod h) (D.mkCone e)
            using 2 3 D.cone-mkCone by auto
          thus D.cones-map h (D.mkCone e) j = χ′ j
            using h 2 3 D.cone-mkCone [of e] J.arr-char D.mkCone-def comp-assoc
            apply (cases J.arr j)
             apply simp-all
             apply (metis (no-types, lifting) D.mkCone-cone χ′)
            using χ′.extensionality
            by presburger
        qed
      qed
      hence ∃h. «h : a′ → dom e» ∧ D.cones-map h (D.mkCone e) = χ′
        by blast
      moreover have ⋀h′. «h′ : a′ → dom e» ∧ D.cones-map h′ (D.mkCone e) = χ′ ⟹ h′
= h
      proof (elim conjE)
        fix h′
        assume h′: «h′ : a′ → dom e»
        assume eq: D.cones-map h′ (D.mkCone e) = χ′
        have e · h′ = χ′ J.Zero
          using eq D.cone-mkCone D.mkCone-def χ′.preserves-reflects-arr χ.cone-axioms
              ‹seq f (χ′ J.Zero)› eq h′ in-homE mem-Collect-eq restrict-apply seqE
          apply simp
          by fastforce
        moreover have ∃!h. e · h = χ′ J.Zero
          using assms(2,4) 1 category.seqI by blast
        ultimately show h′ = h
          using h by auto
      qed
      ultimately show ∃!h. «h : a′ → dom e» ∧ D.cones-map h (D.mkCone e) = χ′
        by blast
    qed
    show D.has-as-equalizer e
      using assms χ.limit-cone-axioms by blast
  qed
qed
```

```
  lemma has-as-equalizerE [elim]:
  assumes has-as-equalizer f g e
  and ⟦seq f e; f · e = g · e; ⋀e'. ⟦seq f e'; f · e' = g · e'⟧ ⟹ ∃!h. e · h = e'⟧ ⟹ T
  shows T
  proof −
    interpret D: parallel-pair-diagram C f g
      using assms
      by (simp add: category-axioms has-as-equalizer-def parallel-pair-diagram.intro
          parallel-pair-diagram-axioms-def)
    have D.has-as-equalizer e
      using assms has-as-equalizer-def by blast
    interpret equalizer-cone C f g e
      by (simp add: ‹D.has-as-equalizer e› category-axioms equalizer-cone-def
          D.parallel-pair-diagram-axioms)
    show T
      by (metis arr-iff-in-hom assms(2) dom-comp equalizes is-universal' seqE)
  qed

end
```

20.7 Limits by Products and Equalizers

A category with equalizers has limits of shape J if it has products indexed by the set of arrows of J and the set of objects of J. The proof is patterned after [4], Theorem 2, page 109:

> "The limit of F: $J \to C$ is the equalizer e of f, g: $\Pi_i\ F_i \to \Pi_u\ F_{cod\ u}$ ($u \in arr\ J$, $i \in J$) where $p_u\ f = p_{cod\ u}$, $p_u\ g = F_u\ o\ p_{dom\ u}$; the limiting cone μ is $\mu_j = p_j\ e$, for $j \in J$."

```
locale category-with-equalizers =
  category C
for C :: 'c comp      (infixr ‹·› 55) +
assumes has-equalizers: has-equalizers
begin

  lemma has-limits-if-has-products:
  fixes J :: 'j comp  (infixr ‹·J› 55)
  assumes category J and has-products (Collect (partial-composition.ide J))
  and has-products (Collect (partial-composition.arr J))
  shows has-limits-of-shape J
  proof (unfold has-limits-of-shape-def)
    interpret J: category J using assms(1) by auto
    have ⋀D. diagram J C D ⟹ (∃a χ. limit-cone J C D a χ)
    proof −
      fix D
      assume D: diagram J C D
      interpret D: diagram J C D using D by auto
```

First, construct the two required products and their cones.

interpret *Obj*: *discrete-category* ‹*Collect J.ide*› *J.null*
  using *J.not-arr-null J.ideD*(*1*) *mem-Collect-eq* **by** (*unfold-locales*, *blast*)
interpret Δo: *discrete-diagram-from-map* ‹*Collect J.ide*› *C D J.null*
  using *D.preserves-ide* **by** (*unfold-locales*, *auto*)
have $\exists p$. *has-as-product Obj.comp* $\Delta o.map\ p$
  using *assms*(*2*) $\Delta o.diagram\text{-}axioms$ *has-products-def Obj.arr-char*
  by (*metis* (*no-types*, *lifting*) *Collect-cong* $\Delta o.discrete\text{-}diagram\text{-}axioms$ *mem-Collect-eq*)
from *this* **obtain** $\Pi o\ \pi o$ **where** πo: *product-cone Obj.comp C* $\Delta o.map\ \Pi o\ \pi o$
  using *ex-productE* [*of Obj.comp* $\Delta o.map$] **by** *auto*
interpret πo: *product-cone Obj.comp C* $\Delta o.map\ \Pi o\ \pi o$ **using** πo **by** *auto*
have $\pi o\text{-}in\text{-}hom$: $\bigwedge j.$ *Obj.arr j* $\Longrightarrow$ «$\pi o\ j : \Pi o \to D\ j$»
  using $\pi o.preserves\text{-}dom$ $\pi o.preserves\text{-}cod$ $\Delta o.map\text{-}def$ **by** *auto*

interpret *Arr*: *discrete-category* ‹*Collect J.arr*› *J.null*
  using *J.not-arr-null* **by** (*unfold-locales*, *blast*)
interpret Δa: *discrete-diagram-from-map* ‹*Collect J.arr*› *C* ‹*D o J.cod*› *J.null*
  by (*unfold-locales*, *auto*)
have $\exists p$. *has-as-product Arr.comp* $\Delta a.map\ p$
  using *assms*(*3*) *has-products-def* [*of Collect J.arr*] $\Delta a.discrete\text{-}diagram\text{-}axioms$
  by *blast*
from *this* **obtain** $\Pi a\ \pi a$ **where** πa: *product-cone Arr.comp C* $\Delta a.map\ \Pi a\ \pi a$
  using *ex-productE* [*of Arr.comp* $\Delta a.map$] **by** *auto*
interpret πa: *product-cone Arr.comp C* $\Delta a.map\ \Pi a\ \pi a$ **using** πa **by** *auto*
have $\pi a\text{-}in\text{-}hom$: $\bigwedge j.$ *Arr.arr j* $\Longrightarrow$ «$\pi a\ j : \Pi a \to D\ (J.cod\ j)$»
  using $\pi a.preserves\text{-}cod$ $\pi a.preserves\text{-}dom$ $\Delta a.map\text{-}def$ **by** *auto*

Next, construct a parallel pair of arrows f, g: $\Pi o \to \Pi a$ that expresses the commutativity constraints imposed by the diagram.

interpret Πo: *constant-functor Arr.comp C* Πo
  using $\pi o.ide\text{-}apex$ **by** (*unfold-locales*, *auto*)
let $?\chi = \lambda j.$ *if Arr.arr j then* $\pi o\ (J.cod\ j)$ *else null*
interpret χ: *cone Arr.comp C* $\Delta a.map\ \Pi o\ ?\chi$
  using $\pi o.ide\text{-}apex$ $\pi o\text{-}in\text{-}hom$ $\Delta a.map\text{-}def$ $\Delta o.map\text{-}def$ $\Delta o.is\text{-}discrete$ $\pi o.naturality2$
    comp-cod-arr
  by (*unfold-locales*, *auto*)

let $?f = \pi a.induced\text{-}arrow\ \Pi o\ ?\chi$
have *f-in-hom*: «$?f : \Pi o \to \Pi a$»
  using $\chi.cone\text{-}axioms$ $\pi a.induced\text{-}arrowI$ **by** *blast*
have *f-map*: $\Delta a.cones\text{-}map\ ?f\ \pi a = ?\chi$
  using $\chi.cone\text{-}axioms$ $\pi a.induced\text{-}arrowI$ **by** *blast*
have *ff*: $\bigwedge j.$ *J.arr j* $\Longrightarrow$ $\pi a\ j \cdot ?f = \pi o\ (J.cod\ j)$
  using $\chi.component\text{-}in\text{-}hom$ $\pi a.induced\text{-}arrowI'$ $\pi o.ide\text{-}apex$ **by** *auto*

let $?\chi' = \lambda j.$ *if Arr.arr j then D j* $\cdot$ $\pi o\ (J.dom\ j)$ *else null*
interpret χ': *cone Arr.comp C* $\Delta a.map\ \Pi o\ ?\chi'$
  using $\pi o.ide\text{-}apex$ $\pi o\text{-}in\text{-}hom$ $\Delta o.map\text{-}def$ $\Delta a.map\text{-}def$ *comp-arr-dom comp-cod-arr*
  by (*unfold-locales*, *auto*)

```
let ?g = πa.induced-arrow Πo ?χ'
have g-in-hom: «?g : Πo → Πa»
  using χ'.cone-axioms πa.induced-arrowI by blast
have g-map: Δa.cones-map ?g πa = ?χ'
  using χ'.cone-axioms πa.induced-arrowI by blast
have gg: ⋀j. J.arr j ⟹ πa j · ?g = D j · πo (J.dom j)
  using χ'.component-in-hom πa.induced-arrowI' πo.ide-apex by force

interpret PP: parallel-pair-diagram C ?f ?g
  using f-in-hom g-in-hom
  by (elim in-homE, unfold-locales, auto)
from PP.is-parallel obtain e where equ: PP.has-as-equalizer e
  using has-equalizers has-equalizers-def has-as-equalizer-def by blast
interpret EQU: limit-cone PP.J.comp C PP.map ‹dom e› ‹PP.mkCone e›
  using equ by auto
interpret EQU: equalizer-cone C ?f ?g e ..
```

An arrow h with $cod\ h = \Pi o$ equalizes f and g if and only if it satisfies the commutativity condition required for a cone over D.

```
have E: ⋀h. «h : dom h → Πo» ⟹
          ?f · h = ?g · h ⟷ (∀j. J.arr j ⟶ ?χ j · h = ?χ' j · h)
proof
  fix h
  assume h: «h : dom h → Πo»
  show ?f · h = ?g · h ⟹ ∀j. J.arr j ⟶ ?χ j · h = ?χ' j · h
  proof −
    assume E: ?f · h = ?g · h
    have ⋀j. J.arr j ⟹ ?χ j · h = ?χ' j · h
    proof −
      fix j
      assume j: J.arr j
      have ?χ j · h = Δa.cones-map ?f πa j · h
        using j f-map by fastforce
      also have ... = πa j · ?f · h
        using j f-in-hom Δa.map-def πa.is-cone comp-assoc by auto
      also have ... = πa j · ?g · h
        using j E by simp
      also have ... = Δa.cones-map ?g πa j · h
        using j g-in-hom Δa.map-def πa.is-cone comp-assoc by auto
      also have ... = ?χ' j · h
        using j g-map by force
      finally show ?χ j · h = ?χ' j · h by auto
    qed
    thus ∀j. J.arr j ⟶ ?χ j · h = ?χ' j · h by blast
  qed
  show ∀j. J.arr j ⟶ ?χ j · h = ?χ' j · h ⟹ ?f · h = ?g · h
  proof −
    assume 1: ∀j. J.arr j ⟶ ?χ j · h = ?χ' j · h
    have 2: ⋀j. j ∈ Collect J.arr ⟹ πa j · ?f · h = πa j · ?g · h
```

```
    proof −
      fix j
      assume j: j ∈ Collect J.arr
      have πa j · ?f · h = (πa j · ?f) · h
        using comp-assoc by simp
      also have ... = ?χ j · h
        using ff j by force
      also have ... = ?χ' j · h
        using 1 j by auto
      also have ... = (πa j · ?g) · h
        using gg j by force
      also have ... = πa j · ?g · h
        using comp-assoc by simp
      finally show πa j · ?f · h = πa j · ?g · h
        by auto
    qed
    show C ?f h = C ?g h
    proof −
      have ⋀j. Arr.arr j ⟹ «πa j · ?f · h : dom h → Δa.map j»
        using f-in-hom h πa-in-hom by (elim in-homE, auto)
      hence 3: ∃!k. «k : dom h → Πa» ∧ (∀j. Arr.arr j ⟶ πa j · k = πa j · ?f · h)
        using h πa πa.is-universal' [of dom h λj. πa j · ?f · h] Δa.map-def
              ide-dom [of h]
        by blast
      have 4: ⋀P x x'. ∃!k. P k x ⟹ P x x ⟹ P x' x ⟹ x' = x by auto
      let ?P = λ k x. «k : dom h → Πa» ∧
                      (∀j. j ∈ Collect J.arr ⟶ πa j · k = πa j · x)
      have ?P (?g · h) (?g · h)
        using g-in-hom h by force
      moreover have ?P (?f · h) (?g · h)
        using 2 f-in-hom g-in-hom h by force
      ultimately show ?thesis
        using 3 4 [of ?P ?f · h ?g · h] by auto
    qed
  qed
qed
have E': ⋀e. «e : dom e → Πo» ⟹
         ?f · e = ?g · e ⟷
         (∀j. J.arr j ⟶
              (D (J.cod j) · πo (J.cod j) · e) · dom e = D j · πo (J.dom j) · e)
proof −
  have 1: ⋀e j. «e : dom e → Πo» ⟹ J.arr j ⟹
                ?χ j · e = (D (J.cod j) · πo (J.cod j) · e) · dom e
  proof −
    fix e j
    assume e: «e : dom e → Πo»
    assume j: J.arr j
    have «πo (J.cod j) · e : dom e → D (J.cod j)»
      using e j πo-in-hom by auto
```

```
    thus ?χ j · e = (D (J.cod j) · πo (J.cod j) · e) · dom e
      using j comp-arr-dom comp-cod-arr by (elim in-homE, auto)
  qed
  have 2: ⋀e j. «e : dom e → Πo» ⟹ J.arr j ⟹ ?χ' j · e = D j · πo (J.dom j) · e
    by (simp add: comp-assoc)
  show ⋀e. «e : dom e → Πo» ⟹
         ?f · e = ?g · e ⟷
          (∀j. J.arr j ⟶
               (D (J.cod j) · πo (J.cod j) · e) · dom e = D j · πo (J.dom j) · e)
    using 1 2 E by presburger
qed
```

The composites of e with the projections from the product Πo determine a limit cone μ for D. The component of μ at an object j of J is the composite $\pi o\ j \cdot e$. However, we need to extend μ to all arrows j of J, so the correct definition is $\mu\ j = D\ j \cdot \pi o\ (J.dom\ j) \cdot e$.

```
have e-in-hom: «e : dom e → Πo»
  using EQU.equalizes f-in-hom in-homI
  by (metis (no-types, lifting) seqE in-homE)
have e-map: C ?f e = C ?g e
  using EQU.equalizes f-in-hom in-homI by fastforce
interpret domE: constant-functor J C ‹dom e›
  using e-in-hom by (unfold-locales, auto)
let ?μ = λj. if J.arr j then D j · πo (J.dom j) · e else null
have μ: ⋀j. J.arr j ⟹ «?μ j : dom e → D (J.cod j)»
proof −
  fix j
  assume j: J.arr j
  show «?μ j : dom e → D (J.cod j)»
    using j e-in-hom πo-in-hom [of J.dom j] by auto
qed
interpret μ: cone J C D ‹dom e› ?μ
  using μ comp-cod-arr e-in-hom e-map E'
    apply unfold-locales
      apply auto
  by (metis D.as-nat-trans.naturality1 comp-assoc)
```

If τ is any cone over D then τ restricts to a cone over Δo for which the induced arrow to Πo equalizes f and g.

```
have R: ⋀a τ. cone J C D a τ ⟹
            cone Obj.comp C Δo.map a (Δo.mkCone τ) ∧
            ?f · πo.induced-arrow a (Δo.mkCone τ)
              = ?g · πo.induced-arrow a (Δo.mkCone τ)
proof −
  fix a τ
  assume cone-τ: cone J C D a τ
  interpret τ: cone J C D a τ using cone-τ by auto
  interpret A: constant-functor Obj.comp C a
    using τ.ide-apex by (unfold-locales, auto)
```

```
    interpret τo: cone Obj.comp C Δo.map a ‹Δo.mkCone τ›
      using A.value-is-ide Δo.map-def comp-cod-arr comp-arr-dom
      by (unfold-locales, auto)
    let ?e = πo.induced-arrow a (Δo.mkCone τ)
    have mkCone-τ: Δo.mkCone τ ∈ Δo.cones a
      using τo.cone-axioms by auto
    have e: «?e : a → Πo»
      using mkCone-τ πo.induced-arrowI by simp
    have ee: ⋀j. J.ide j ⟹ πo j · ?e = τ j
    proof −
      fix j
      assume j: J.ide j
      have πo j · ?e = Δo.cones-map ?e πo j
        using j e πo.cone-axioms by force
      also have ... = Δo.mkCone τ j
        using j mkCone-τ πo.induced-arrowI [of Δo.mkCone τ a] by fastforce
      also have ... = τ j
        using j by simp
      finally show πo j · ?e = τ j by auto
    qed
    have ⋀j. J.arr j ⟹
              (D (J.cod j) · πo (J.cod j) · ?e) · dom ?e = D j · πo (J.dom j) · ?e
    proof −
      fix j
      assume j: J.arr j
      have 1: «πo (J.cod j) : Πo → D (J.cod j)» using j πo-in-hom by simp
      have 2: (D (J.cod j) · πo (J.cod j) · ?e) · dom ?e
                = D (J.cod j) · πo (J.cod j) · ?e
      proof −
        have seq (D (J.cod j)) (πo (J.cod j))
          using j 1 by auto
        moreover have seq (πo (J.cod j)) ?e
          using j e by fastforce
        ultimately show ?thesis using comp-arr-dom by auto
      qed
      also have 3: ... = πo (J.cod j) · ?e
        using j e 1 comp-cod-arr by (elim in-homE, auto)
      also have ... = D j · πo (J.dom j) · ?e
        using j e ee 2 3 τ.naturality τ.A.map-simp τ.ide-apex comp-cod-arr by auto
      finally show (D (J.cod j) · πo (J.cod j) · ?e) · dom ?e = D j · πo (J.dom j) · ?e
        by auto
    qed
    hence C ?f ?e = C ?g ?e
      using E' πo.induced-arrowI τo.cone-axioms mem-Collect-eq by blast
    thus cone Obj.comp C Δo.map a (Δo.mkCone τ) ∧ C ?f ?e = C ?g ?e
      using τo.cone-axioms by auto
  qed
```

Finally, show that μ is a limit cone.

```
interpret μ: limit-cone J C D ‹dom e› ?μ
proof
  fix a τ
  assume cone-τ: cone J C D a τ
  interpret τ: cone J C D a τ using cone-τ by auto
  interpret A: constant-functor Obj.comp C a
    using τ.ide-apex by unfold-locales auto
  have cone-τo: cone Obj.comp C Δo.map a (Δo.mkCone τ)
    using A.value-is-ide Δo.map-def D.preserves-ide comp-cod-arr comp-arr-dom
        τ.preserves-hom
    by unfold-locales auto
  show ∃!h. «h : a → dom e» ∧ D.cones-map h ?μ = τ
  proof
    let ?e' = πo.induced-arrow a (Δo.mkCone τ)
    have e'-in-hom: «?e' : a → Πo»
      using cone-τ R πo.induced-arrowI by auto
    have e'-map: ?f · ?e' = ?g · ?e' ∧ Δo.cones-map ?e' πo = Δo.mkCone τ
      using cone-τ R πo.induced-arrowI [of Δo.mkCone τ a] by auto
    have equ: PP.is-equalized-by ?e'
      using e'-map e'-in-hom f-in-hom seqI' by blast
    let ?h = EQU.induced-arrow a (PP.mkCone ?e')
    have h-in-hom: «?h : a → dom e»
      using EQU.induced-arrowI PP.cone-mkCone [of ?e'] e'-in-hom equ by fastforce
    have h-map: PP.cones-map ?h (PP.mkCone e) = PP.mkCone ?e'
      using EQU.induced-arrowI [of PP.mkCone ?e' a] PP.cone-mkCone [of ?e']
          e'-in-hom equ
      by fastforce
    have 3: D.cones-map ?h ?μ = τ
    proof
      fix j
      have ¬J.arr j ⟹ D.cones-map ?h ?μ j = τ j
        using h-in-hom μ.cone-axioms cone-τ τ.extensionality by force
      moreover have J.arr j ⟹ D.cones-map ?h ?μ j = τ j
      proof −
        fix j
        assume j: J.arr j
        have 1: «πo (J.dom j) · e : dom e → D (J.dom j)»
          using j e-in-hom πo-in-hom [of J.dom j] by auto
        have D.cones-map ?h ?μ j = ?μ j · ?h
          using h-in-hom j μ.cone-axioms by auto
        also have ... = D j · (πo (J.dom j) · e) · ?h
          using j comp-assoc by simp
        also have ... = D j · τ (J.dom j)
        proof −
          have (πo (J.dom j) · e) · ?h = τ (J.dom j)
          proof −
            have (πo (J.dom j) · e) · ?h = πo (J.dom j) · e · ?h
              using j 1 e-in-hom h-in-hom πo arrI comp-assoc by auto
            also have ... = πo (J.dom j) · ?e'
```

```
        using equ e'-in-hom EQU.induced-arrowI' [of ?e'] by auto
      also have ... = Δo.cones-map ?e' πo (J.dom j)
        using j e'-in-hom πo.cone-axioms by auto
      also have ... = τ (J.dom j)
        using j e'-map by simp
      finally show ?thesis by auto
    qed
    thus ?thesis by simp
  qed
  also have ... = τ j
    using j τ.naturality1 by simp
  finally show D.cones-map ?h ?μ j = τ j by auto
 qed
 ultimately show D.cones-map ?h ?μ j = τ j by auto
qed
show «?h : a → dom e» ∧ D.cones-map ?h ?μ = τ
 using h-in-hom 3 by simp
show ⋀h'. «h' : a → dom e» ∧ D.cones-map h' ?μ = τ ⟹ h' = ?h
proof −
 fix h'
 assume h': «h' : a → dom e» ∧ D.cones-map h' ?μ = τ
 have h'-in-hom: «h' : a → dom e» using h' by simp
 have h'-map: D.cones-map h' ?μ = τ using h' by simp
 show h' = ?h
 proof −
  have 1: «e · h' : a → Πo» ∧ ?f · e · h' = ?g · e · h' ∧
          Δo.cones-map (C e h') πo = Δo.mkCone τ
  proof −
   have 2: «e · h' : a → Πo» using h'-in-hom e-in-hom by auto
   moreover have ?f · e · h' = ?g · e · h'
    by (metis (no-types, lifting) EQU.equalizes comp-assoc)
   moreover have Δo.cones-map (e · h') πo = Δo.mkCone τ
   proof
    have Δo.cones-map (e · h') πo = Δo.cones-map h' (Δo.cones-map e πo)
     using πo.cone-axioms e-in-hom h'-in-hom Δo.cones-map-comp [of e h']
     by fastforce
    fix j
    have ¬Obj.arr j ⟹ Δo.cones-map (e · h') πo j = Δo.mkCone τ j
     using 2 e-in-hom h'-in-hom πo.cone-axioms by auto
    moreover have Obj.arr j ⟹ Δo.cones-map (e · h') πo j = Δo.mkCone τ j
    proof −
     assume j: Obj.arr j
     have Δo.cones-map (e · h') πo j = πo j · e · h'
      using 2 j πo.cone-axioms by auto
     also have ... = (πo j · e) · h'
      using comp-assoc by auto
     also have ... = Δo.mkCone ?μ j · h'
      using j e-in-hom πo-in-hom comp-ide-arr [of D j πo j · e]
      by fastforce
```

```
            also have ... = Δo.mkCone τ j
              using j h' μ.cone-axioms mem-Collect-eq by auto
            finally show Δo.cones-map (e · h') πo j = Δo.mkCone τ j by auto
          qed
          ultimately show Δo.cones-map (e · h') πo j = Δo.mkCone τ j by auto
        qed
        ultimately show ?thesis by auto
      qed
      have «e · h' : a → Πo» using 1 by simp
      moreover have e · h' = ?e'
        using 1 cone-τo e'-in-hom e'-map πo.is-universal πo by blast
      ultimately show h' = ?h
        using 1 h'-in-hom h'-map EQU.is-universal' [of e · h']
              EQU.induced-arrowI' [of ?e'] equ
        by (elim in-homE) auto
    qed
   qed
  qed
 qed
 have limit-cone J C D (dom e) ?μ ..
 thus ∃a μ. limit-cone J C D a μ by auto
qed
thus ∀D. diagram J C D ⟶ (∃a μ. limit-cone J C D a μ) by blast
qed

end
```

20.8 Limits in a Set Category

In this section, we consider the special case of limits in a set category.

```
locale diagram-in-set-category =
  J: category J +
  S: set-category S is-set +
  diagram J S D
for J :: 'j comp      (infixr ‹·J› 55)
and S :: 's comp      (infixr ‹·› 55)
and is-set :: 's set ⇒ bool
and D :: 'j ⇒ 's
begin

  notation S.in-hom (‹«- : - → -»›)
```

An object a of a set category S is a limit of a diagram in S if and only if there is a bijection between the set $S.hom\ S.unity\ a$ of points of a and the set of cones over the diagram that have apex $S.unity$.

```
  lemma limits-are-sets-of-cones:
  shows has-as-limit a ⟷ S.ide a ∧ (∃φ. bij-betw φ (S.hom S.unity a) (cones S.unity))
  proof
```

If *has-limit a*, then by the universal property of the limit cone, composition in S yields a bijection between *S.hom S.unity a* and *cones S.unity*.

assume *a*: *has-as-limit a*
hence *S.ide a*
using *limit-cone-def cone.ide-apex* **by** *metis*
from *a* **obtain** χ **where** χ: *limit-cone a* χ **by** *auto*
interpret χ: *limit-cone J S D a* χ **using** χ **by** *auto*
have *bij-betw* (λf. *cones-map f* χ) (*S.hom S.unity a*) (*cones S.unity*)
using χ*.bij-betw-hom-and-cones S.ide-unity* **by** *simp*
thus *S.ide a* $\wedge$ ($\exists \varphi$. *bij-betw* φ (*S.hom S.unity a*) (*cones S.unity*))
using ‹*S.ide a*› **by** *blast*
next

Conversely, an arbitrary bijection φ between *S.hom S.unity a* and cones unity extends pointwise to a natural bijection Φ a' between *S.hom* a' *a* and *cones* a', showing that a is a limit.

In more detail, the hypotheses give us a correspondence between points of a and cones with apex *S.unity*. We extend this to a correspondence between functions to a and general cones, with each arrow from a' to a determining a cone with apex a'. If $f \in hom\ a'\ a$ then composition with f takes each point y of a' to the point $f \cdot y$ of a. To this we may apply the given bijection φ to obtain $\varphi\ (f \cdot y) \in cones\ S.unity$. The component $\varphi\ (f \cdot y)\ j$ at j of this cone is a point of *S.cod* (*D j*). Thus, $f \in hom\ a'\ a$ determines a cone χf with apex a' whose component at j is the unique arrow $\chi f\ j$ of S such that $\chi f\ j \in hom\ a'\ (cod\ (D\ j))$ and $\chi f\ j \cdot y = \varphi\ (f \cdot y)\ j$ for all points y of a'. The cone χa corresponding to $a \in S.hom\ a\ a$ is then a limit cone.

assume *a*: *S.ide a* $\wedge$ ($\exists \varphi$. *bij-betw* φ (*S.hom S.unity a*) (*cones S.unity*))
hence *ide-a*: *S.ide a* **by** *auto*
show *has-as-limit a*
proof −
from *a* **obtain** φ **where** φ: *bij-betw* φ (*S.hom S.unity a*) (*cones S.unity*) **by** *blast*
have *X*: $\bigwedge f\ j\ y$. ⟦ «*f* : *S.dom f* → *a*»; *J.arr j*; «*y* : *S.unity* → *S.dom f*» ⟧
⟹ «$\varphi\ (f \cdot y)\ j$: *S.unity* → *S.cod* (*D j*)»
proof −
fix *f j y*
assume *f*: «*f* : *S.dom f* → *a*» **and** *j*: *J.arr j* **and** *y*: «*y* : *S.unity* → *S.dom f*»
interpret χ: *cone J S D S.unity* ‹φ (*S f y*)›
using *f y* φ *bij-betw-imp-funcset funcset-mem* **by** *blast*
show «$\varphi\ (f \cdot y)\ j$: *S.unity* → *S.cod* (*D j*)» **using** *j* **by** *auto*
qed

We want to define the component $\chi j \in S.hom\ (S.dom\ f)\ (S.cod\ (D\ j))$ at j of a cone by specifying how it acts by composition on points $y \in S.hom\ S.unity\ (S.dom\ f)$. We can do this because S is a set category.

let *?P* = $\lambda f\ j\ \chi j$. «χj : *S.dom f* → *S.cod* (*D j*)» $\wedge$
($\forall y$. «*y* : *S.unity* → *S.dom f*» ⟶ $\chi j \cdot y = \varphi\ (f \cdot y)\ j$)
let *?*χ = $\lambda f\ j$. *if J.arr j then* (*THE* χj. *?P f j* χj) *else S.null*
have χ: $\bigwedge f\ j$. ⟦ «*f* : *S.dom f* → *a*»; *J.arr j* ⟧ ⟹ *?P f j* (*?*χ *f j*)

```
proof −
  fix b f j
  assume f: «f : S.dom f → a» and j: J.arr j
  interpret B: constant-functor J S ‹S.dom f›
    using f by (unfold-locales) auto
  have (λy. φ (f · y) j) ∈ S.hom S.unity (S.dom f) → S.hom S.unity (S.cod (D j))
    using f j X Pi-I′ by simp
  hence ∃!χj. ?P f j χj
    using f j S.fun-complete′ by (elim S.in-homE) auto
  thus ?P f j (?χ f j) using j theI′ [of ?P f j] by simp
qed
```

The arrows $\chi\ f\ j$ are in fact the components of a cone with apex $S.dom\ f$.

```
have cone: ⋀f. «f : S.dom f → a» ⟹ cone (S.dom f) (?χ f)
proof −
  fix f
  assume f: «f : S.dom f → a»
  interpret B: constant-functor J S ‹S.dom f›
    using f by unfold-locales auto
  show cone (S.dom f) (?χ f)
  proof
    show ⋀j. ¬J.arr j ⟹ ?χ f j = S.null by simp
    fix j
    assume j: J.arr j
    have 0: «?χ f j : S.dom f → S.cod (D j)» using f j χ by simp
    show S.arr (?χ f j) using f j χ by auto
    have par2: S.par (?χ f (J.cod j) · B.map j) (?χ f j)
      using f j 0 χ [of f J.cod j] by (elim S.in-homE, auto)
    have nat: ⋀y. «y : S.unity → S.dom f» ⟹
                    (D j · ?χ f (J.dom j)) · y = ?χ f j · y ∧
                    (?χ f (J.cod j) · B.map j) · y = ?χ f j · y
    proof −
      fix y
      assume y: «y : S.unity → S.dom f»
      show (D j · ?χ f (J.dom j)) · y = ?χ f j · y ∧
           (?χ f (J.cod j) · B.map j) · y = ?χ f j · y
      proof
        have 1: φ (f · y) ∈ cones S.unity
          using f y φ bij-betw-imp-funcset PiE
                S.seqI S.cod-comp S.dom-comp mem-Collect-eq
          by fastforce
        interpret χ: cone J S D S.unity ‹φ (f · y)›
          using 1 by simp
        show (D j · ?χ f (J.dom j)) · y = ?χ f j · y
          using J.arr-dom S.comp-assoc χ χ.naturality1 f j y by presburger
        have (?χ f (J.cod j) · B.map j) · y = ?χ f (J.cod j) · y
          using j B.map-simp par2 B.value-is-ide S.comp-arr-ide
          by (metis (no-types, lifting))
        also have ... = φ (f · y) (J.cod j)
```

```
        using f y χ χ.extensionality by simp
      also have ... = φ (f · y) j
        using j χ.naturality2
        by (metis J.arr-cod χ.A.map-simp J.cod-cod)
      also have ... = ?χ f j · y
        using f y χ χ.extensionality by simp
      finally show (?χ f (J.cod j) · B.map j) · y = ?χ f j · y by auto
    qed
  qed
  show D j · ?χ f (J.dom j) = ?χ f j
  proof −
    have S.par (D j · ?χ f (J.dom j)) (?χ f j)
      using f j 0 χ [of f J.dom j] by (elim S.in-homE, auto)
    thus ?thesis
      using nat 0
      apply (intro S.arr-eqI'SC [of D j · ?χ f (J.dom j) ?χ f j])
       apply force
      by auto
  qed
  show ?χ f (J.cod j) · B.map j = ?χ f j
    using par2 nat 0 f j χ
    apply (intro S.arr-eqI'SC [of ?χ f (J.cod j) · B.map j ?χ f j])
     apply force
    by (metis (no-types, lifting) S.in-homE)
  qed
qed
interpret χa: cone J S D a ‹?χ a› using a cone [of a] by fastforce
```

Finally, show that $\chi\ a$ is a limit cone.

```
interpret χa: limit-cone J S D a ‹?χ a›
proof
  fix a' χ'
  assume cone-χ': cone a' χ'
  interpret χ': cone J S D a' χ' using cone-χ' by auto
  show ∃!f. «f : a' → a» ∧ cones-map f (?χ a) = χ'
  proof
    let ?ψ = inv-into (S.hom S.unity a) φ
    have ψ: ?ψ ∈ cones S.unity → S.hom S.unity a
      using φ bij-betw-inv-into bij-betwE by blast
    let ?P = λf. «f : a' → a» ∧
                 (∀ y. y ∈ S.hom S.unity a' ⟶ f · y = ?ψ (cones-map y χ'))
    have 1: ∃!f. ?P f
    proof −
      have (λy. ?ψ (cones-map y χ')) ∈ S.hom S.unity a' → S.hom S.unity a
      proof
        fix x
        assume x ∈ S.hom S.unity a'
        hence «x : S.unity → a'» by simp
        hence cones-map x ∈ cones a' → cones S.unity
```

```
      using cones-map-mapsto [of x] by (elim S.in-homE) auto
    hence cones-map x χ' ∈ cones S.unity
      using cone-χ' by blast
    thus ?ψ (cones-map x χ') ∈ S.hom S.unity a
      using ψ by auto
  qed
  thus ?thesis
    using S.fun-complete' a χ'.ide-apex by simp
qed
let ?f = THE f. ?P f
have f: ?P ?f using 1 theI' [of ?P] by simp
have f-in-hom: «?f : a' → a» using f by simp
have f-map: cones-map ?f (?χ a) = χ'
proof −
  have 1: cone a' (cones-map ?f (?χ a))
  proof −
    have cones-map ?f ∈ cones a → cones a'
      using f-in-hom cones-map-mapsto [of ?f] by (elim S.in-homE) auto
    hence cones-map ?f (?χ a) ∈ cones a'
      using χa.cone-axioms by blast
    thus ?thesis by simp
  qed
  interpret fχa: cone J S D a' ‹cones-map ?f (?χ a)›
    using 1 by simp
  show ?thesis
  proof
    fix j
    have ¬J.arr j ⟹ cones-map ?f (?χ a) j = χ' j
      using 1 χ'.extensionality fχa.extensionality by presburger
    moreover have J.arr j ⟹ cones-map ?f (?χ a) j = χ' j
    proof −
      assume j: J.arr j
      show cones-map ?f (?χ a) j = χ' j
      proof (intro S.arr-eqI'SC [of cones-map ?f (?χ a) j χ' j])
        show par: S.par (cones-map ?f (?χ a) j) (χ' j)
          using j χ'.preserves-cod χ'.preserves-dom χ'.preserves-reflects-arr
                fχa.preserves-cod fχa.preserves-dom fχa.preserves-reflects-arr
          by presburger
        fix y
        assume «y : S.unity → S.dom (cones-map ?f (?χ a) j)»
        hence y: «y : S.unity → a'»
          using j fχa.preserves-dom by simp
        have 1: «?χ a j : a → D (J.cod j)»
          using j χa.preserves-hom by force
        have 2: «?f · y : S.unity → a»
          using f-in-hom y by blast
        have cones-map ?f (?χ a) j · y = (?χ a j · ?f) · y
        proof −
          have S.cod ?f = a using f-in-hom by blast
```

```
        thus ?thesis using j χa.cone-axioms by simp
      qed
      also have ... = ?χ a j · ?f · y
        using 1 j y f-in-hom S.comp-assoc S.seqI' by blast
      also have ... = φ (a · ?f · y) j
        using 1 2 ide-a f j y χ [of a] by (simp add: S.ide-in-hom)
      also have ... = φ (?f · y) j
        using a 2 y S.comp-cod-arr by (elim S.in-homE, auto)
      also have ... = φ (?ψ (cones-map y χ')) j
        using j y f by simp
      also have ... = cones-map y χ' j
      proof −
        have cones-map y χ' ∈ cones S.unity
          using cone-χ' y cones-map-mapsto by force
        hence φ (?ψ (cones-map y χ')) = cones-map y χ'
          using φ bij-betw-inv-into-right [of φ] by simp
        thus ?thesis by auto
      qed
      also have ... = χ' j · y
        using cone-χ' j y by auto
      finally show cones-map ?f (?χ a) j · y = χ' j · y
        by auto
    qed
  qed
  ultimately show cones-map ?f (?χ a) j = χ' j by blast
 qed
qed
show «?f : a' → a» ∧ cones-map ?f (?χ a) = χ'
 using f-in-hom f-map by simp
show ⋀f'. «f' : a' → a» ∧ cones-map f' (?χ a) = χ' ⟹ f' = ?f
proof −
 fix f'
 assume f': «f' : a' → a» ∧ cones-map f' (?χ a) = χ'
 have f'-in-hom: «f' : a' → a» using f' by simp
 have f'-map: cones-map f' (?χ a) = χ' using f' by simp
 show f' = ?f
 proof (intro S.arr-eqI'SC [of f' ?f])
  show S.par f' ?f
    using f-in-hom f'-in-hom by (elim S.in-homE, auto)
  show ⋀y'. «y' : S.unity → S.dom f'» ⟹ f' · y' = ?f · y'
  proof −
    fix y'
    assume y': «y' : S.unity → S.dom f'»
    have 0: φ (f' · y') = cones-map y' χ'
    proof
      fix j
      have 1: «f' · y' : S.unity → a» using f'-in-hom y' by auto
      hence 2: φ (f' · y') ∈ cones S.unity
        using φ bij-betw-imp-funcset [of φ S.hom S.unity a cones S.unity]
```

```
            by auto
          interpret χ'': cone J S D S.unity ‹φ (f' · y')› using 2 by auto
          have ¬J.arr j ⟹ φ (f' · y') j = cones-map y' χ' j
            using f' y' cone-χ' χ''.extensionality mem-Collect-eq restrict-apply
            by (elim S.in-homE, auto)
          moreover have J.arr j ⟹ φ (f' · y') j = cones-map y' χ' j
          proof −
            assume j: J.arr j
            have 3: « ?χ a j : a → D (J.cod j)»
              using j χa.preserves-hom by force
            have φ (f' · y') j = φ (a · f' · y') j
              using a f' y' j S.comp-cod-arr by (elim S.in-homE, auto)
            also have ... = ?χ a j · f' · y'
              using 1 3 χ [of a] a f' y' j by fastforce
            also have ... = (?χ a j · f') · y'
              using S.comp-assoc by simp
            also have ... = cones-map f' (?χ a) j · y'
              using f' y' j χa.cone-axioms by auto
            also have ... = χ' j · y'
              using f' by blast
            also have ... = cones-map y' χ' j
              using y' j cone-χ' f' mem-Collect-eq restrict-apply by force
            finally show φ (f' · y') j = cones-map y' χ' j by auto
          qed
          ultimately show φ (f' · y') j = cones-map y' χ' j by auto
        qed
        hence f' · y' = ?ψ (cones-map y' χ')
          using φ f'-in-hom y' S.comp-in-homI
                bij-betw-inv-into-left [of φ S.hom S.unity a cones S.unity f' · y']
          by (elim S.in-homE, auto)
        moreover have ?f · y' = ?ψ (cones-map y' χ')
          using φ 0 1 f f-in-hom f'-in-hom y' S.comp-in-homI
                bij-betw-inv-into-left [of φ S.hom S.unity a cones S.unity ?f · y']
          by (elim S.in-homE, auto)
        ultimately show f' · y' = ?f · y' by auto
      qed
    qed
  qed
  have limit-cone a (?χ a) ..
  thus ?thesis by auto
 qed
qed

end

locale diagram-in-replete-set-category =
  J: category J +
```

```
  S: replete-set-category S +
  diagram J S D
for J :: 'j comp      (infixr ‹·J› 55)
and S :: 's comp      (infixr ‹·› 55)
and D :: 'j ⇒ 's
begin

  sublocale diagram-in-set-category J S S.setp D
    ..

end

context set-category
begin
```

A set category has an equalizer for any parallel pair of arrows.

```
  lemma has-equalizers_SC:
  shows has-equalizers
  proof (unfold has-equalizers-def)
    have ⋀f0 f1. par f0 f1 ⟹ ∃e. has-as-equalizer f0 f1 e
    proof −
      fix f0 f1
      assume par: par f0 f1
      interpret J: parallel-pair .
      interpret PP: parallel-pair-diagram S f0 f1
        using par by unfold-locales auto
      interpret PP: diagram-in-set-category J.comp S setp PP.map ..
```

Let a be the object corresponding to the set of all images of equalizing points of *dom f0*, and let e be the inclusion of a in *dom f0*.

```
      let ?a = mkIde (img ‘ {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e})
      have 0: {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e} ⊆ hom unity (dom f0)
        by auto
      hence 1: img ‘ {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e} ⊆ Univ
        using img-point-in-Univ by auto
      have 2: setp (img ‘ {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e})
      proof −
        have setp (img ‘ hom unity (dom f0))
          using ide-dom par setp-img-points by blast
        moreover have img ‘ {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e} ⊆
                      img ‘ hom unity (dom f0)
          by blast
        ultimately show ?thesis
          by (meson setp-respects-subset)
      qed
      have ide-a: ide ?a using 1 2 ide-mkIde by auto
      have set-a: set ?a = img ‘ {e. e ∈ hom unity (dom f0) ∧ f0 · e = f1 · e}
        using 1 2 set-mkIde by simp
      have incl-in-a: incl-in ?a (dom f0)
```

proof −
have *ide* (*dom f0*)
using *PP.is-parallel* **by** *simp*
moreover have *set ?a* ⊆ *set* (*dom f0*)
using *img-point-elem-set set-a* **by** *fastforce*
ultimately show *?thesis*
using *incl-in-def* ‹*ide ?a*› **by** *simp*
qed

Then *set a* is in bijective correspondence with *PP.cones unity*.

let *?φ* = λ*t*. *PP.mkCone* (*mkPoint* (*dom f0*) *t*)
let *?ψ* = λχ. *img* (χ (*J.Zero*))
have *bij*: *bij-betw ?φ* (*set ?a*) (*PP.cones unity*)
proof (*intro bij-betwI*)
show *?φ* ∈ *set ?a* → *PP.cones unity*
proof
fix *t*
assume *t*: *t* ∈ *set ?a*
hence *1*: *t* ∈ *img* ‘ {*e*. *e* ∈ *hom unity* (*dom f0*) ∧ *f0* · *e* = *f1* · *e*}
using *set-a* **by** *blast*
then have *2*: *mkPoint* (*dom f0*) *t* ∈ *hom unity* (*dom f0*)
using *mkPoint-in-hom imageE mem-Collect-eq mkPoint-img*(*2*) **by** *auto*
with *1* **have** *3*: *mkPoint* (*dom f0*) *t* ∈ {*e*. *e* ∈ *hom unity* (*dom f0*) ∧ *f0* · *e* = *f1* · *e*}
using *mkPoint-img*(*2*) **by** *auto*
then have *PP.is-equalized-by* (*mkPoint* (*dom f0*) *t*)
using *CollectD par* **by** *fastforce*
thus *PP.mkCone* (*mkPoint* (*dom f0*) *t*) ∈ *PP.cones unity*
using *2 PP.cone-mkCone* [*of mkPoint* (*dom f0*) *t*] **by** *auto*
qed
show *?ψ* ∈ *PP.cones unity* → *set ?a*
proof
fix χ
assume χ: χ ∈ *PP.cones unity*
interpret χ: *cone J.comp S PP.map unity* χ **using** χ **by** *auto*
have χ (*J.Zero*) ∈ *hom unity* (*dom f0*) ∧ *f0* · χ (*J.Zero*) = *f1* · χ (*J.Zero*)
using χ *PP.map-def PP.is-equalized-by-cone J.arr-char* **by** *auto*
hence *img* (χ (*J.Zero*)) ∈ *set ?a*
using *set-a* **by** *simp*
thus *?ψ* χ ∈ *set ?a* **by** *blast*
qed
show ⋀*t*. *t* ∈ *set ?a* ⟹ *?ψ* (*?φ t*) = *t*
using *set-a J.arr-char PP.mkCone-def imageE mem-Collect-eq mkPoint-img*(*2*)
by *auto*
show ⋀χ. χ ∈ *PP.cones unity* ⟹ *?φ* (*?ψ* χ) = χ
proof −
fix χ
assume χ: χ ∈ *PP.cones unity*
interpret χ: *cone J.comp S PP.map unity* χ **using** χ **by** *auto*
have *1*: χ (*J.Zero*) ∈ *hom unity* (*dom f0*) ∧ *f0* · χ (*J.Zero*) = *f1* · χ (*J.Zero*)

```
        using χ PP.map-def PP.is-equalized-by-cone J.arr-char by auto
      hence img (χ (J.Zero)) ∈ set ?a
        using set-a by simp
      hence img (χ (J.Zero)) ∈ set (dom f0)
        using incl-in-a incl-in-def by auto
      hence mkPoint (dom f0) (img (χ J.Zero)) = χ J.Zero
        using 1 mkPoint-img(2) by blast
      hence ?φ (?ψ χ) = PP.mkCone (χ J.Zero) by simp
      also have ... = χ
        using χ PP.mkCone-cone by simp
      finally show ?φ (?ψ χ) = χ by auto
    qed
  qed
```

It follows that a is a limit of PP, and that the limit cone gives an equalizer of *f0* and *f1*.

```
  have PP.has-as-limit ?a
  proof −
    have ∃μ. bij-betw μ (hom unity ?a) (set ?a)
      using bij-betw-points-and-set ide-a by auto
    from this obtain μ where μ: bij-betw μ (hom unity ?a) (set ?a) by blast
    have bij-betw (?φ o μ) (hom unity ?a) (PP.cones unity)
      using bij μ bij-betw-comp-iff by blast
    hence ∃φ. bij-betw φ (hom unity ?a) (PP.cones unity) by auto
    thus ?thesis
      using ide-a PP.limits-are-sets-of-cones by simp
  qed
  from this obtain ε where ε: limit-cone J.comp S PP.map ?a ε by auto
  have PP.has-as-equalizer (ε J.Zero)
  proof −
    interpret ε: limit-cone J.comp S PP.map ?a ε using ε by auto
    have PP.mkCone (ε (J.Zero)) = ε
      using ε PP.mkCone-cone ε.cone-axioms by simp
    moreover have dom (ε (J.Zero)) = ?a
      using J.ide-char ε.preserves-hom ε.A.map-def by simp
    ultimately show ?thesis
      using ε by simp
  qed
  thus ∃ e. has-as-equalizer f0 f1 e
    using par has-as-equalizer-def by auto
 qed
 thus ∀ f0 f1. par f0 f1 ⟶ (∃ e. has-as-equalizer f0 f1 e) by auto
qed

end

sublocale set-category ⊆ category-with-equalizers S
 apply unfold-locales using has-equalizers_SC by auto
```

context *set-category*
begin

The aim of the next results is to characterize the conditions under which a set category has products. In a traditional development of category theory, one shows that the category **Set** of *all* sets has all small (*i.e.* set-indexed) products. In the present context we do not have a category of *all* sets, but rather only a category of all sets with elements at a particular type. Clearly, we cannot expect such a category to have products indexed by arbitrarily large sets. The existence of I-indexed products in a set category S implies that the universe $S.Univ$ of S must be large enough to admit the formation of I-tuples of its elements. Conversely, for a set category S the ability to form I-tuples in *Univ* implies that S has I-indexed products. Below we make this precise by defining the notion of when a set category S "admits I-indexed tupling" and we show that S has I-indexed products if and only if it admits I-indexed tupling.

The definition of "S admits I-indexed tupling" says that there is an injective map, from the space of extensional functions from I to *Univ*, to *Univ*. However for a convenient statement and proof of the desired result, the definition of extensional function from theory *HOL−Library.FuncSet* needs to be modified. The theory *HOL−Library.FuncSet* uses the definite, but arbitrarily chosen value *undefined* as the value to be assumed by an extensional function outside of its domain. In the context of the *set-category*, though, it is more natural to use $S.unity$, which is guaranteed to be an element of the universe of S, for this purpose. Doing things that way makes it simpler to establish a bijective correspondence between cones over D with apex *unity* and the set of extensional functions d that map each arrow j of J to an element $d\ j$ of *set* $(D\ j)$. Possibly it makes sense to go back and make this change in *set-category*, but that would mean completely abandoning *HOL−Library.FuncSet* and essentially introducing a duplicate version for use with *set-category*. As a compromise, what I have done here is to locally redefine the few notions from *HOL−Library.FuncSet* that I need in order to prove the next set of results. The redefined notions are primed to avoid confusion with the original versions.

definition *extensional′*
where $extensional'\ A \equiv \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = unity\}$

abbreviation *PiE′*
where $PiE'\ A\ B \equiv Pi\ A\ B \cap extensional'\ A$

abbreviation *restrict′*
where $restrict'\ f\ A \equiv \lambda x.\ if\ x \in A\ then\ f\ x\ else\ unity$

lemma *extensional′I* [*intro*]:
assumes $\bigwedge x.\ x \notin A \Longrightarrow f\ x = unity$
shows $f \in extensional'\ A$
  using *assms extensional′-def* **by** *auto*

lemma *extensional′-arb*:
assumes $f \in extensional'\ A$ **and** $x \notin A$
shows $f\ x = unity$

using *assms extensional'-def* **by** *fast*

lemma *extensional'-monotone*:
assumes $A \subseteq B$
shows *extensional'* $A \subseteq$ *extensional'* B
using *assms extensional'-arb* **by** *fastforce*

lemma *PiE'-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow PiE'\ A\ B \subseteq PiE'\ A\ C$
by *auto*

end

locale *discrete-diagram-in-set-category* =
S: *set-category* $S\ \mathfrak{S}$ +
discrete-diagram $J\ S\ D$ +
diagram-in-set-category $J\ S\ \mathfrak{S}\ D$
for $J :: 'j\ comp$ (**infixr** $\langle\cdot_J\rangle$ *55*)
and $S :: 's\ comp$ (**infixr** $\langle\cdot\rangle$ *55*)
and $\mathfrak{S} :: 's\ set \Rightarrow bool$
and $D :: 'j \Rightarrow 's$
begin

For D a discrete diagram in a set category, there is a bijective correspondence between cones over D with apex unity and the set of extensional functions d that map each arrow j of J to an element of $S.set\ (D\ j)$.

abbreviation I
where $I \equiv Collect\ J.arr$

definition *funToCone*
where *funToCone* $F \equiv \lambda j.$ *if* $J.arr\ j$ *then* $S.mkPoint\ (D\ j)\ (F\ j)$ *else* $S.null$

definition *coneToFun*
where *coneToFun* $\chi \equiv \lambda j.$ *if* $J.arr\ j$ *then* $S.img\ (\chi\ j)$ *else* $S.unity$

lemma *funToCone-mapsto*:
shows $funToCone \in S.PiE'\ I\ (S.set\ o\ D) \rightarrow cones\ S.unity$
proof
fix F
assume F: $F \in S.PiE'\ I\ (S.set\ o\ D)$
interpret U: *constant-functor* $J\ S\ S.unity$
apply *unfold-locales* **using** *S.ide-unity* **by** *auto*
have *cone* $S.unity\ (funToCone\ F)$
proof
show $\bigwedge j.\ \neg J.arr\ j \Longrightarrow funToCone\ F\ j = S.null$
using *funToCone-def* **by** *simp*
fix j
assume j: $J.arr\ j$
have $funToCone\ F\ j = S.mkPoint\ (D\ j)\ (F\ j)$
using j *funToCone-def* **by** *simp*

```
    moreover have ... ∈ S.hom S.unity (D j)
      using F j is-discrete S.img-mkPoint(1) [of D j] by force
    ultimately have 2: funToCone F j ∈ S.hom S.unity (D j) by auto
    show S.arr (funToCone F j)
      using 2 j by auto
    show D j · funToCone F (J.dom j) = funToCone F j
      using 2 j is-discrete S.comp-cod-arr by auto
    show funToCone F (J.cod j) · (U.map j) = funToCone F j
      using 2 S.comp-arr-dom is-discrete j by auto
  qed
  thus funToCone F ∈ cones S.unity by auto
qed

lemma coneToFun-mapsto:
shows coneToFun ∈ cones S.unity → S.PiE' I (S.set o D)
proof
  fix χ
  assume χ: χ ∈ cones S.unity
  interpret χ: cone J S D S.unity χ using χ by auto
  show coneToFun χ ∈ S.PiE' I (S.set o D)
  proof
    show coneToFun χ ∈ Pi I (S.set o D)
      using S.mkPoint-img(1) coneToFun-def is-discrete χ.component-in-hom
      by (simp add: S.img-point-elem-set restrict-apply')
    show coneToFun χ ∈ S.extensional' I
      by (metis S.extensional'I coneToFun-def mem-Collect-eq)
  qed
qed

lemma funToCone-coneToFun:
assumes χ ∈ cones S.unity
shows funToCone (coneToFun χ) = χ
proof
  interpret χ: cone J S D S.unity χ using assms by auto
  fix j
  have ¬J.arr j ⟹ funToCone (coneToFun χ) j = χ j
    using funToCone-def χ.extensionality by simp
  moreover have J.arr j ⟹ funToCone (coneToFun χ) j = χ j
    using funToCone-def coneToFun-def S.mkPoint-img(2) is-discrete χ.component-in-hom
    by auto
  ultimately show funToCone (coneToFun χ) j = χ j by blast
qed

lemma coneToFun-funToCone:
assumes F ∈ S.PiE' I (S.set o D)
shows coneToFun (funToCone F) = F
proof
  fix i
  have i ∉ I ⟹ coneToFun (funToCone F) i = F i
```

using *assms coneToFun-def S.extensional'-arb* [*of F I i*] **by** *auto*
moreover have $i \in I \Longrightarrow$ *coneToFun* (*funToCone F*) *i* = *F i*
proof −
assume *i*: $i \in I$
have *coneToFun* (*funToCone F*) *i* = *S.img* (*funToCone F i*)
using *i coneToFun-def* **by** *simp*
also have ... = *S.img* (*S.mkPoint* (*D i*) (*F i*))
using *i funToCone-def* **by** *auto*
also have ... = *F i*
using *assms i is-discrete S.img-mkPoint*(*2*) **by** *force*
finally show *coneToFun* (*funToCone F*) *i* = *F i* **by** *auto*
qed
ultimately show *coneToFun* (*funToCone F*) *i* = *F i* **by** *auto*
qed

lemma *bij-coneToFun*:
shows *bij-betw coneToFun* (*cones S.unity*) (*S.PiE' I* (*S.set o D*))
using *coneToFun-mapsto funToCone-mapsto funToCone-coneToFun coneToFun-funToCone bij-betwI*
by *blast*

lemma *bij-funToCone*:
shows *bij-betw funToCone* (*S.PiE' I* (*S.set o D*)) (*cones S.unity*)
using *coneToFun-mapsto funToCone-mapsto funToCone-coneToFun coneToFun-funToCone bij-betwI*
by *blast*

end

context *set-category*
begin

A set category admits I-indexed tupling if there is an injective map that takes each extensional function from I to *Univ* to an element of *Univ*.

definition *admits-tupling*
where *admits-tupling I* ≡ ∃ π. π ∈ *PiE' I* (λ-. *Univ*) → *Univ* ∧ *inj-on* π (*PiE' I* (λ-. *Univ*))

lemma *admits-tupling-monotone*:
assumes *admits-tupling I* **and** $I' \subseteq I$
shows *admits-tupling I'*
proof −
from *assms*(*1*) **obtain** π
where π: π ∈ *PiE' I* (λ-. *Univ*) → *Univ* ∧ *inj-on* π (*PiE' I* (λ-. *Univ*))
using *admits-tupling-def* **by** *metis*
have π ∈ *PiE' I'* (λ-. *Univ*) → *Univ*
proof
fix *f*
assume *f*: *f* ∈ *PiE' I'* (λ-. *Univ*)
have *f* ∈ *PiE' I* (λ-. *Univ*)

using *assms*(2) *f extensional'-def* [*of I'*] *terminal-unity*$_{SC}$ *extensional'-monotone* **by** *auto*

thus $\pi\ f \in Univ$ **using** π **by** *auto*

qed

moreover have *inj-on* π (*PiE'* I' (λ-. *Univ*))

proof −

have *1*: $\bigwedge F\ A\ A'$. *inj-on* $F\ A \wedge A' \subseteq A \Longrightarrow$ *inj-on* $F\ A'$

using *subset-inj-on* **by** *blast*

moreover have *PiE'* I' (λ-. *Univ*) $\subseteq$ *PiE'* I (λ-. *Univ*)

using *assms*(2) *extensional'-def* [*of I'*] *terminal-unity*$_{SC}$ **by** *auto*

ultimately show *?thesis* **using** π *assms*(2) **by** *blast*

qed

ultimately show *?thesis* **using** *admits-tupling-def* **by** *metis*

qed

lemma *admits-tupling-respects-bij*:

assumes *admits-tupling I* **and** *bij-betw* φ *I I'*

shows *admits-tupling I'*

proof −

obtain π **where** π: $\pi \in (I \to Univ) \cap extensional'\ I \to Univ \wedge$
inj-on π (($I \to Univ$) $\cap$ *extensional'* I)

using *assms*(1) *admits-tupling-def* **by** *metis*

have *inv*: *bij-betw* (*inv-into I* φ) I' I

using *assms*(2) *bij-betw-inv-into* **by** *blast*

let *?C* = $\lambda f\ x$. *if* $x \in I$ *then* f ($\varphi\ x$) *else unity*

let *?*π' = λf. π (*?C* f)

have *1*: $\bigwedge f$. $f \in (I' \to Univ) \cap extensional'\ I' \Longrightarrow$ *?C* $f \in (I \to Univ) \cap extensional'\ I$

using *assms bij-betw-apply* **by** *fastforce*

have *?*$\pi' \in (I' \to Univ) \cap extensional'\ I' \to Univ \wedge$
inj-on ?π' (($I' \to Univ$) $\cap$ *extensional'* I')

proof

show (λf. π (*?C* f)) $\in (I' \to Univ) \cap extensional'\ I' \to Univ$

using *1* π **by** *blast*

show *inj-on ?*π' (($I' \to Univ$) $\cap$ *extensional'* I')

proof

fix $f\ g$

assume *f*: $f \in (I' \to Univ) \cap extensional'\ I'$

assume *g*: $g \in (I' \to Univ) \cap extensional'\ I'$

assume *eq*: *?*$\pi'\ f$ = *?*$\pi'\ g$

have *f'*: *?C* $f \in (I \to Univ) \cap extensional'\ I$

using *f 1* **by** *simp*

have *g'*: *?C* $g \in (I \to Univ) \cap extensional'\ I$

using *g 1* **by** *simp*

have *2*: *?C* f = *?C* g

using $f'\ g'\ \pi$ *eq* **by** (*simp add*: *inj-on-def*)

show $f = g$

proof

fix x

show $f\ x = g\ x$

```
        proof (cases x ∈ I')
          show x ∈ I' ⟹ ?thesis
            using f g
            by (metis (no-types, opaque-lifting) 2 assms(2) bij-betw-apply
                bij-betw-inv-into-right inv)
          show x ∉ I' ⟹ ?thesis
            using f g by (metis IntD2 extensional'-arb)
        qed
      qed
    qed
  qed
  thus ?thesis
    using admits-tupling-def by blast
qed

end

context replete-set-category
begin

  lemma has-products-iff-admits-tupling:
  fixes I :: 'i set
  shows has-products I ⟷ I ≠ UNIV ∧ admits-tupling I
  proof
```

If S has I-indexed products, then for every I-indexed discrete diagram D in S there is an object ΠD of S whose points are in bijective correspondence with the set of cones over D with apex *unity*. In particular this is true for the diagram D that assigns to each element of I the "universal object" *mkIde Univ*.

```
    assume has-products: has-products I
    have I: I ≠ UNIV using has-products has-products-def by auto
    interpret J: discrete-category I ‹SOME x. x ∉ I›
      using I someI-ex [of λx. x ∉ I] by (unfold-locales, auto)
    let ?D = λi. mkIde Univ
    interpret D: discrete-diagram-from-map I S ?D ‹SOME j. j ∉ I›
      using J.not-arr-null J.arr-char ide-mkIde
      by (unfold-locales, auto)
    interpret D: discrete-diagram-in-set-category J.comp S ‹λA. A ⊆ Univ› D.map ..
    have discrete-diagram J.comp S D.map ..
    from this obtain ΠD χ where χ: product-cone J.comp S D.map ΠD χ
      using has-products has-products-def [of I] ex-productE [of J.comp D.map]
            D.diagram-axioms
      by blast
    interpret χ: product-cone J.comp S D.map ΠD χ
      using χ by auto
    have D.has-as-limit ΠD
      using χ.limit-cone-axioms by auto
    hence ΠD: ide ΠD ∧ (∃φ. bij-betw φ (hom unity ΠD) (D.cones unity))
      using D.limits-are-sets-of-cones by simp
```

from *this* **obtain** φ **where** φ: *bij-betw* φ (*hom unity* ΠD) (*D.cones unity*)
by *blast*
have φ': *inv-into* (*hom unity* ΠD) $\varphi \in$ *D.cones unity* $\to$ *hom unity* $\Pi D \wedge$
inj-on (*inv-into* (*hom unity* ΠD) φ) (*D.cones unity*)
using φ *bij-betw-inv-into bij-betw-imp-inj-on bij-betw-imp-funcset* **by** *metis*
let $?\pi$ = *img o* (*inv-into* (*hom unity* ΠD) φ) *o D.funToCone*
have *1*: *D.funToCone* $\in$ *PiE'* *I* (*set o D.map*) $\to$ *D.cones unity*
using *D.funToCone-mapsto extensional'-def* [*of I*] **by** *auto*
have *2*: *inv-into* (*hom unity* ΠD) $\varphi \in$ *D.cones unity* $\to$ *hom unity* ΠD
using φ' **by** *auto*
have *3*: *img* $\in$ *hom unity* $\Pi D \to$ *Univ*
using *img-point-in-Univ* **by** *blast*
have *4*: *inj-on D.funToCone* (*PiE'* *I* (*set o D.map*))
proof −
have *D.I* = *I* **by** *auto*
thus *?thesis*
using *D.bij-funToCone bij-betw-imp-inj-on* **by** *auto*
qed
have *5*: *inj-on* (*inv-into* (*hom unity* ΠD) φ) (*D.cones unity*)
using φ' **by** *auto*
have *6*: *inj-on img* (*hom unity* ΠD)
using ΠD *bij-betw-points-and-set bij-betw-imp-inj-on* [*of img hom unity* ΠD *set* ΠD]
by *simp*
have $?\pi \in$ *PiE'* *I* (*set o D.map*) $\to$ *Univ*
using *1 2 3* **by** *force*
moreover have *inj-on* $?\pi$ (*PiE'* *I* (*set o D.map*))
proof −
have *7*: $\bigwedge A\ B\ C\ D\ F\ G\ H.\ F \in A \to B \wedge G \in B \to C \wedge H \in C \to D$
$\wedge$ *inj-on F A* $\wedge$ *inj-on G B* $\wedge$ *inj-on H C*
$\Longrightarrow$ *inj-on* (*H o G o F*) *A*
by (*simp add*: *Pi-iff inj-on-def*)
show *?thesis*
using *1 2 3 4 5 6 7* [*of D.funToCone PiE'* *I* (*set o D.map*) *D.cones unity*
inv-into (*hom unity* ΠD) φ *hom unity* ΠD
img Univ]
by *fastforce*
qed
moreover have *PiE'* *I* (*set o D.map*) = *PiE'* *I* ($\lambda x.$ *Univ*)
proof −
have $\bigwedge i.\ i \in I \Longrightarrow$ (*set o D.map*) *i* = *Univ*
using *J.arr-char D.map-def* **by** *simp*
thus *?thesis* **by** *blast*
qed
ultimately have $?\pi \in$ (*PiE'* *I* ($\lambda x.$ *Univ*)) $\to$ *Univ* $\wedge$ *inj-on* $?\pi$ (*PiE'* *I* ($\lambda x.$ *Univ*))
by *auto*
thus *I* $\neq$ *UNIV* $\wedge$ *admits-tupling I*
using *I admits-tupling-def* **by** *auto*
next
assume *ex-π*: *I* $\neq$ *UNIV* $\wedge$ *admits-tupling I*

show *has-products I*
proof (*unfold has-products-def*)
from *ex-π* **obtain** π
where π: π ∈ (*PiE′ I* (λx. *Univ*)) → *Univ* ∧ *inj-on* π (*PiE′ I* (λx. *Univ*))
using *admits-tupling-def* **by** *metis*

Given an I-indexed discrete diagram D, obtain the object ΠD of S corresponding to the set π ‘ Pi_E I D of all π d where $d \in d \in J \to_E$ *Univ* and $d\ i \in D\ i$ for all $i \in I$. The elements of ΠD are in bijective correspondence with the set of cones over D, hence ΠD is a limit of D.

have ⋀*J D*. *discrete-diagram J S D* ∧ *Collect* (*partial-composition.arr J*) = *I*
⟹ ∃ΠD. *has-as-product J D* ΠD
proof
fix *J* :: *′i comp* **and** *D*
assume *D*: *discrete-diagram J S D* ∧ *Collect* (*partial-composition.arr J*) = *I*
interpret *J*: *category J*
using *D discrete-diagram.axioms*(*1*) **by** *blast*
interpret *D*: *discrete-diagram J S D*
using *D* **by** *simp*
interpret *D*: *discrete-diagram-in-set-category J S* ‹λA. *A* ⊆ *Univ*› *D* **..**
let *?ΠD* = *mkIde* (π ‘ *PiE′ I* (*set o D*))
have *0*: *ide ?ΠD*
proof −
have *set o D* ∈ *I* → *Pow Univ*
using *Pow-iff incl-in-def o-apply elem-set-implies-incl-in subsetI Pi-I′*
setp-set-ide
by (*metis* (*mono-tags*, *lifting*))
hence π ‘ *PiE′ I* (*set o D*) ⊆ *Univ*
using π **by** *blast*
thus *?thesis* **using** π *ide-mkIde* **by** *simp*
qed
hence *set-ΠD*: π ‘ *PiE′ I* (*set o D*) = *set ?ΠD*
using *0 ide-in-hom arr-mkIde set-mkIde* **by** *auto*

The elements of ΠD are all values of the form $\pi\ d$, where d satisfies $d\ i \in set\ (D\ i)$ for all $i \in I$. Such d correspond bijectively to cones. Since π is injective, the values $\pi\ d$ correspond bijectively to cones.

let *?φ* = *mkPoint ?ΠD o* π *o D.coneToFun*
let *?φ′* = *D.funToCone o inv-into* (*PiE′ I* (*set o D*)) π *o img*
have *1*: π ∈ *PiE′ I* (*set o D*) → *set ?ΠD* ∧ *inj-on* π (*PiE′ I* (*set o D*))
proof −
have *PiE′ I* (*set o D*) ⊆ *PiE′ I* (λx. *Univ*)
using *setp-set-ide elem-set-implies-incl-in elem-set-implies-set-eq-singleton*
incl-in-def PiE′-mono comp-apply subsetI
by (*metis* (*no-types*, *lifting*))
thus *?thesis* **using** π *subset-inj-on set-ΠD Pi-I′ imageI* **by** *fastforce*
qed
have *2*: *inv-into* (*PiE′ I* (*set o D*)) π ∈ *set ?ΠD* → *PiE′ I* (*set o D*)
proof

fix y
assume y: $y \in set\ ?\Pi D$
have $y \in \pi\ `\ (PiE'\ I\ (set\ o\ D))$ **using** y $set\text{-}\Pi D$ **by** *auto*
thus $inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ y \in PiE'\ I\ (set\ o\ D)$
using $inv\text{-}into\text{-}into\ [of\ y\ \pi\ PiE'\ I\ (set\ o\ D)]$ **by** *simp*
qed
have *3*: $\bigwedge x.\ x \in set\ ?\Pi D \Longrightarrow \pi\ (inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ x) = x$
using $set\text{-}\Pi D$ **by** (*simp add*: *f-inv-into-f*)
have *4*: $\bigwedge d.\ d \in PiE'\ I\ (set\ o\ D) \Longrightarrow inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ (\pi\ d) = d$
using *1* **by** *auto*
have *5*: $D.I = I$
using D **by** *auto*
have $bij\text{-}betw\ ?\varphi\ (D.cones\ unity)\ (hom\ unity\ ?\Pi D)$
proof (*intro bij-betwI*)
show $?\varphi \in D.cones\ unity \to hom\ unity\ ?\Pi D$
proof
fix χ
assume χ: $\chi \in D.cones\ unity$
show $?\varphi\ \chi \in hom\ unity\ ?\Pi D$
using χ *0 1 5* $D.coneToFun\text{-}mapsto\ mkPoint\text{-}in\text{-}hom\ [of\ ?\Pi D]$
by (*simp*, *blast*)
qed
show $?\varphi' \in hom\ unity\ ?\Pi D \to D.cones\ unity$
proof
fix x
assume x: $x \in hom\ unity\ ?\Pi D$
hence $img\ x \in set\ ?\Pi D$
using *img-point-elem-set* **by** *blast*
hence $inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ (img\ x) \in Pi\ I\ (set \circ D) \cap extensional'\ I$
using *2* **by** *blast*
thus $?\varphi'\ x \in D.cones\ unity$
using *5* $D.funToCone\text{-}mapsto$ **by** *auto*
qed
show $\bigwedge x.\ x \in hom\ unity\ ?\Pi D \Longrightarrow ?\varphi\ (?\varphi'\ x) = x$
proof −
fix x
assume x: $x \in hom\ unity\ ?\Pi D$
show $?\varphi\ (?\varphi'\ x) = x$
proof −
have $D.coneToFun\ (D.funToCone\ (inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ (img\ x)))$
$= inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ (img\ x)$
using *x 1 5 img-point-elem-set* $set\text{-}\Pi D$ $D.coneToFun\text{-}funToCone$ **by** *force*
hence $\pi\ (D.coneToFun\ (D.funToCone\ (inv\text{-}into\ (PiE'\ I\ (set\ o\ D))\ \pi\ (img\ x))))$
$= img\ x$
using *x 3 img-point-elem-set* $set\text{-}\Pi D$ **by** *force*
thus *?thesis* **using** *x 0 mkPoint-img* **by** *auto*
qed
qed
show $\bigwedge \chi.\ \chi \in D.cones\ unity \Longrightarrow ?\varphi'\ (?\varphi\ \chi) = \chi$

```
        proof −
          fix χ
          assume χ: χ ∈ D.cones unity
          show ?φ' (?φ χ) = χ
          proof −
            have img (mkPoint ?ΠD (π (D.coneToFun χ))) = π (D.coneToFun χ)
              using χ 0 1 5 D.coneToFun-mapsto img-mkPoint(2) by blast
            hence inv-into (PiE' I (set o D)) π (img (mkPoint ?ΠD (π (D.coneToFun χ))))
                  = D.coneToFun χ
              using χ D.coneToFun-mapsto 4 5
              by (metis (no-types, lifting) PiE)
            hence D.funToCone (inv-into (PiE' I (set o D)) π
                                        (img (mkPoint ?ΠD (π (D.coneToFun χ)))))
                  = χ
              using χ D.funToCone-coneToFun by auto
            thus ?thesis by auto
          qed
        qed
      qed
      hence bij-betw (inv-into (D.cones unity) ?φ) (hom unity ?ΠD) (D.cones unity)
        using bij-betw-inv-into by blast
      hence ∃φ. bij-betw φ (hom unity ?ΠD) (D.cones unity) by blast
      hence D.has-as-limit ?ΠD
        using ‹ide ?ΠD› D.limits-are-sets-of-cones by simp
      from this obtain χ where χ: limit-cone J S D ?ΠD χ by blast
      interpret χ: limit-cone J S D ?ΠD χ using χ by auto
      interpret P: product-cone J S D ?ΠD χ
        using χ D.product-coneI by blast
      have product-cone J S D ?ΠD χ ..
      thus has-as-product J D ?ΠD
        using has-as-product-def by auto
    qed
    thus I ≠ UNIV ∧
         (∀J D. discrete-diagram J S D ∧ Collect (partial-composition.arr J) = I
            ⟶ (∃ΠD. has-as-product J D ΠD))
      using ex-π by blast
  qed
qed

end

context replete-set-category
begin
```

Characterization of the completeness properties enjoyed by a set category: A set category S has all limits at a type $'j$, if and only if S admits I-indexed tupling for all $'j$-sets I such that $I \neq UNIV$.

```
theorem has-limits-iff-admits-tupling:
shows has-limits (undefined :: 'j) ⟷ (∀I :: 'j set. I ≠ UNIV ⟶ admits-tupling I)
```

```
  proof
    assume has-limits: has-limits (undefined :: 'j)
    show ∀ I :: 'j set. I ≠ UNIV ⟶ admits-tupling I
      using has-limits has-products-if-has-limits has-products-iff-admits-tupling by blast
    next
    assume admits-tupling: ∀ I :: 'j set. I ≠ UNIV ⟶ admits-tupling I
    show has-limits (undefined :: 'j)
      using has-limits-def admits-tupling has-products-iff-admits-tupling
      by (metis category.axioms(1) category.ideD(1) has-limits-if-has-products
          iso-tuple-UNIV-I mem-Collect-eq partial-composition.not-arr-null)
  qed

end
```

20.9 Limits in Functor Categories

In this section, we consider the special case of limits in functor categories, with the objective of showing that limits in a functor category $[A, B]$ are given pointwise, and that $[A, B]$ has all limits that B has.

```
locale parametrized-diagram =
  J: category J +
  A: category A +
  B: category B +
  JxA: product-category J A +
  binary-functor J A B D
for J :: 'j comp      (infixr ‹·J› 55)
and A :: 'a comp      (infixr ‹·A› 55)
and B :: 'b comp      (infixr ‹·B› 55)
and D :: 'j * 'a ⇒ 'b
begin


  notation J.in-hom     (‹«- : - →J -»›)
  notation JxA.comp     (infixr ‹·JxA› 55)
  notation JxA.in-hom   (‹«- : - →JxA -»›)
```

A choice of limit cone for each diagram $D\ (-,\ a)$, where a is an object of A, extends to a functor $L\colon A \to B$, where the action of L on arrows of A is determined by universality.

```
  abbreviation L
  where L ≡ λl χ. λa. if A.arr a then
                        limit-cone.induced-arrow J B (λj. D (j, A.cod a))
                          (l (A.cod a)) (χ (A.cod a))
                          (l (A.dom a)) (vertical-composite.map J B
                                           (χ (A.dom a)) (λj. D (j, a)))
                      else B.null

  abbreviation P
  where P ≡ λl χ. λa f. «f : l (A.dom a) →B l (A.cod a)» ∧
```

```
                    diagram.cones-map J B (λj. D (j, A.cod a)) f (χ (A.cod a)) =
                    vertical-composite.map J B (χ (A.dom a)) (λj. D (j, a))

  lemma L-arr:
  assumes ∀a. A.ide a ⟶ limit-cone J B (λj. D (j, a)) (l a) (χ a)
  shows ⋀a. A.arr a ⟹ (∃!f. P l χ a f) ∧ P l χ a (L l χ a)
  proof
    fix a
    assume a: A.arr a
    interpret χ-dom-a: limit-cone J B ‹λj. D (j, A.dom a)› ‹l (A.dom a)› ‹χ (A.dom a)›
      using a assms by auto
    interpret χ-cod-a: limit-cone J B ‹λj. D (j, A.cod a)› ‹l (A.cod a)› ‹χ (A.cod a)›
      using a assms by auto
    interpret Da: natural-transformation J B ‹λj. D (j, A.dom a)› ‹λj. D (j, A.cod a)›
                                             ‹λj. D (j, a)›
      using a fixing-arr-gives-natural-transformation-2 by simp
    interpret Daoχ-dom-a: vertical-composite J B
                          χ-dom-a.A.map ‹λj. D (j, A.dom a)› ‹λj. D (j, A.cod a)›
                          ‹χ (A.dom a)› ‹λj. D (j, a)› ..
    interpret Daoχ-dom-a: cone J B ‹λj. D (j, A.cod a)› ‹l (A.dom a)› Daoχ-dom-a.map ..
    show P l χ a (L l χ a)
      using a Daoχ-dom-a.cone-axioms χ-cod-a.induced-arrowI [of Daoχ-dom-a.map l (A.dom
a)]
      by auto
    show ∃!f. P l χ a f
      using χ-cod-a.is-universal Daoχ-dom-a.cone-axioms by blast
  qed

  lemma L-ide:
  assumes ∀a. A.ide a ⟶ limit-cone J B (λj. D (j, a)) (l a) (χ a)
  shows ⋀a. A.ide a ⟹ L l χ a = l a
  proof −
    let ?L = L l χ
    let ?P = P l χ
    fix a
    assume a: A.ide a
    interpret χa: limit-cone J B ‹λj. D (j, a)› ‹l a› ‹χ a› using a assms by auto
    have Pa: ?P a = (λf. f ∈ B.hom (l a) (l a) ∧
                         diagram.cones-map J B (λj. D (j, a)) f (χ a) = χ a)
      using a vcomp-ide-dom χa.natural-transformation-axioms by simp
    have ?P a (?L a) using assms a L-arr [of l χ a] by fastforce
    moreover have ?P a (l a)
    proof −
      have ?P a (l a) ⟷ l a ∈ B.hom (l a) (l a) ∧ χa.D.cones-map (l a) (χ a) = χ a
        using Pa by meson
      thus ?thesis
        using a χa.ide-apex χa.cone-axioms χa.D.cones-map-ide [of χ a l a] by force
    qed
    moreover have ∃!f. ?P a f
```

```
    using a Pa χa.is-universal χa.cone-axioms by force
  ultimately show ?L a = l a by blast
qed

lemma chosen-limits-induce-functor:
assumes ∀ a. A.ide a ⟶ limit-cone J B (λj. D (j, a)) (l a) (χ a)
shows functor A B (L l χ)
proof −
  let ?L = L l χ
  let ?P = λa. λf. «f : l (A.dom a) →B l (A.cod a)» ∧
                   diagram.cones-map J B (λj. D (j, A.cod a)) f (χ (A.cod a))
                     = vertical-composite.map J B (χ (A.dom a)) (λj. D (j, a))
  interpret L: functor A B ?L
    apply unfold-locales
    using assms L-arr [of l] L-ide
       apply auto[4]
  proof −
    fix a′ a
    assume 1: A.arr (A a′ a)
    have a: A.arr a using 1 by auto
    have a′: «a′ : A.cod a →A A.cod a′» using 1 by auto
    have a′a: A.seq a′ a using 1 by auto
    interpret χ-dom-a: limit-cone J B ‹λj. D (j, A.dom a)› ‹l (A.dom a)› ‹χ (A.dom a)›
      using a assms by auto
    interpret χ-cod-a: limit-cone J B ‹λj. D (j, A.cod a)› ‹l (A.cod a)› ‹χ (A.cod a)›
      using a′a assms by auto
    interpret χ-dom-a′a: limit-cone J B ‹λj. D (j, A.dom (a′ ·A a))› ‹l (A.dom (a′ ·A a))›
                                        ‹χ (A.dom (a′ ·A a))›
      using a′a assms by auto
    interpret χ-cod-a′a: limit-cone J B ‹λj. D (j, A.cod (a′ ·A a))› ‹l (A.cod (a′ ·A a))›
                                        ‹χ (A.cod (a′ ·A a))›
      using a′a assms by auto
    interpret Da: natural-transformation J B
                  ‹λj. D (j, A.dom a)› ‹λj. D (j, A.cod a)› ‹λj. D (j, a)›
      using a fixing-arr-gives-natural-transformation-2 by simp
    interpret Da′: natural-transformation J B
                  ‹λj. D (j, A.cod a)› ‹λj. D (j, A.cod (a′ ·A a))› ‹λj. D (j, a′)›
      using a a′a fixing-arr-gives-natural-transformation-2 by fastforce
    interpret Da′oχ-cod-a: vertical-composite J B
                           χ-cod-a.A.map ‹λj. D (j, A.cod a)› ‹λj. D (j, A.cod (a′ ·A a))›
                           ‹χ (A.cod a)› ‹λj. D (j, a′)›..
  interpret Da′oχ-cod-a: cone J B ‹λj. D (j, A.cod (a′ ·A a))› ‹l (A.cod a)› Da′oχ-cod-a.map
    ..
    interpret Da′a: natural-transformation J B
                    ‹λj. D (j, A.dom (a′ ·A a))› ‹λj. D (j, A.cod (a′ ·A a))›
                    ‹λj. D (j, a′ ·A a)›
      using a′a fixing-arr-gives-natural-transformation-2 [of a′ ·A a] by auto
    interpret Da′aoχ-dom-a′a:
       vertical-composite J B χ-dom-a′a.A.map ‹λj. D (j, A.dom (a′ ·A a))›
```

```
                    ‹λj. D (j, A.cod (a' ·A a))› ‹χ (A.dom (a' ·A a))›
                    ‹λj. D (j, a' ·A a)› ..
interpret Da'aoχ-dom-a'a: cone J B ‹λj. D (j, A.cod (a' ·A a))›
                                   ‹l (A.dom (a' ·A a))› Da'aoχ-dom-a'a.map ..
show ?L (a' ·A a) = ?L a' ·B ?L a
proof −
  have ?P (a' ·A a) (?L (a' ·A a)) using assms a'a L-arr [of l χ a' ·A a] by fastforce
  moreover have ?P (a' ·A a) (?L a' ·B ?L a)
  proof
    have La: «?L a : l (A.dom a) →B l (A.cod a)»
      using assms a L-arr by fast
    moreover have La': «?L a' : l (A.cod a) →B l (A.cod a')»
      using assms a a' L-arr [of l χ a'] by auto
    ultimately have seq: B.seq (?L a') (?L a) by (elim B.in-homE, auto)
    thus La'-La: «?L a' ·B ?L a : l (A.dom (a' ·A a)) →B l (A.cod (a' ·A a))»
      using a a' 1 La La' by (intro B.comp-in-homI, auto)
    show χ-cod-a'a.D.cones-map (?L a' ·B ?L a) (χ (A.cod (a' ·A a)))
          = Da'aoχ-dom-a'a.map
    proof −
      have χ-cod-a'a.D.cones-map (?L a' ·B ?L a) (χ (A.cod (a' ·A a)))
             = (χ-cod-a'a.D.cones-map (?L a) o χ-cod-a'a.D.cones-map (?L a'))
                (χ (A.cod a'))
      proof −
        have χ-cod-a'a.D.cones-map (?L a' ·B ?L a) (χ (A.cod (a' ·A a))) =
              restrict (χ-cod-a'a.D.cones-map (?L a) ∘ χ-cod-a'a.D.cones-map (?L a'))
                       (χ-cod-a'a.D.cones (B.cod (?L a')))
                       (χ (A.cod (a' ·A a)))
          using seq χ-cod-a'a.cone-axioms χ-cod-a'a.D.cones-map-comp [of ?L a' ?L a]
          by argo
        also have ... = (χ-cod-a'a.D.cones-map (?L a) o χ-cod-a'a.D.cones-map (?L a'))
                         (χ (A.cod a'))
        proof −
          have χ (A.cod a') ∈ χ-cod-a'a.D.cones (l (A.cod a'))
            using χ-cod-a'a.cone-axioms a'a by simp
          moreover have B.cod (?L a') = l (A.cod a')
            using assms a' L-arr [of l] by auto
          ultimately show ?thesis
            using a' a'a by simp
        qed
        finally show ?thesis by blast
      qed
      also have ... = χ-cod-a'a.D.cones-map (?L a)
                       (χ-cod-a'a.D.cones-map (?L a') (χ (A.cod a')))
        by simp
      also have ... = χ-cod-a'a.D.cones-map (?L a) Da'oχ-cod-a.map
      proof −
        have ?P a' (?L a') using assms a' L-arr [of l χ a'] by fast
        moreover have
            ?P a' = (λf. f ∈ B.hom (l (A.cod a)) (l (A.cod a')) ∧
```

```
                    χ-cod-a′a.D.cones-map f (χ (A.cod a′)) = Da′oχ-cod-a.map)
          using a′a by force
        ultimately show ?thesis using a′a by force
      qed
      also have ... = vertical-composite.map J B
                    (χ-cod-a.D.cones-map (?L a) (χ (A.cod a)))
                    (λj. D (j, a′))
        using assms χ-cod-a.D.diagram-axioms χ-cod-a′a.D.diagram-axioms
            Da′.natural-transformation-axioms χ-cod-a.cone-axioms La
            cones-map-vcomp [of J B λj. D (j, A.cod a) λj. D (j, A.cod (a′ ·A a))
                             λj. D (j, a′) l (A.cod a) χ (A.cod a)
                             ?L a l (A.dom a)]
        by blast
      also have ... = vertical-composite.map J B
                    (vertical-composite.map J B (χ (A.dom a)) (λj. D (j, a)))
                    (λj. D (j, a′))
        using assms a L-arr by presburger
      also have ... = vertical-composite.map J B (χ (A.dom a))
                    (vertical-composite.map J B (λj. D (j, a)) (λj. D (j, a′)))
        using a′a Da.natural-transformation-axioms Da′.natural-transformation-axioms
            χ-dom-a.natural-transformation-axioms vcomp-assoc
        by auto
      also have
        ... = vertical-composite.map J B (χ (A.dom (a′ ·A a))) (λj. D (j, a′ ·A a))
        using a′a preserves-comp-2 by simp
      finally show ?thesis by auto
    qed
  qed
  moreover have ∃!f. ?P (a′ ·A a) f
    using χ-cod-a′a.is-universal
        [of l (A.dom (a′ ·A a))
            vertical-composite.map J B (χ (A.dom (a′ ·A a))) (λj. D (j, a′ ·A a))]
      Da′aoχ-dom-a′a.cone-axioms
    by fast
  ultimately show ?thesis by blast
 qed
qed
show ?thesis ..
qed

end

locale diagram-in-functor-category =
  A: category A +
  B: category B +
  A-B: functor-category A B +
  diagram J A-B.comp D
for A :: 'a comp      (infixr ‹·A› 55)
and B :: 'b comp      (infixr ‹·B› 55)
```

and $J :: 'j$ *comp* (**infixr** ‹$\cdot_J$› *55*)
and $D :: 'j \Rightarrow ('a, 'b)$ *functor-category.arr*
begin

interpretation *JxA*: *product-category J A* **..**
interpretation *A-BxA*: *product-category A-B.comp A* **..**
interpretation *E*: *evaluation-functor A B* **..**
interpretation *Curry*: *currying J A B* **..**

notation *JxA.comp* (**infixr** ‹$\cdot_{J x A}$› *55*)
notation *JxA.in-hom* (‹«- : - →$_{J x A}$ -»›)

Evaluation of a functor or natural transformation from J to $[A, B]$ at an arrow a of A.

abbreviation *at*
where *at a* $\tau \equiv \lambda j$. *Curry.uncurry* τ (j, a)

lemma *at-simp*:
assumes *A.arr a* **and** *J.arr j* **and** *A-B.arr* $(\tau\ j)$
shows *at a* τ *j* = *A-B.Map* $(\tau\ j)$ *a*
using *assms Curry.uncurry-def E.map-simp* **by** *simp*

lemma *functor-at-ide-is-functor*:
assumes *functor J A-B.comp F* **and** *A.ide a*
shows *functor J B* (*at a F*)
proof −
interpret *uncurry-F*: *functor JxA.comp B* ‹*Curry.uncurry F*›
using *assms*(*1*) *Curry.uncurry-preserves-functors* **by** *simp*
interpret *uncurry-F*: *binary-functor J A B* ‹*Curry.uncurry F*› **..**
show *?thesis* **using** *assms*(*2*) *uncurry-F.fixing-ide-gives-functor-2* **by** *simp*
qed

lemma *functor-at-arr-is-transformation*:
assumes *functor J A-B.comp F* **and** *A.arr a*
shows *natural-transformation J B* (*at* (*A.dom a*) *F*) (*at* (*A.cod a*) *F*) (*at a F*)
proof −
interpret *uncurry-F*: *functor JxA.comp B* ‹*Curry.uncurry F*›
using *assms*(*1*) *Curry.uncurry-preserves-functors* **by** *simp*
interpret *uncurry-F*: *binary-functor J A B* ‹*Curry.uncurry F*› **..**
show *?thesis*
using *assms*(*2*) *uncurry-F.fixing-arr-gives-natural-transformation-2* **by** *simp*
qed

lemma *transformation-at-ide-is-transformation*:
assumes *natural-transformation J A-B.comp F G* τ **and** *A.ide a*
shows *natural-transformation J B* (*at a F*) (*at a G*) (*at a* τ)
proof −
interpret τ: *natural-transformation J A-B.comp F G* τ **using** *assms*(*1*) **by** *auto*
interpret *uncurry-F*: *functor JxA.comp B* ‹*Curry.uncurry F*›

```
    using Curry.uncurry-preserves-functors τ.F.functor-axioms by simp
  interpret uncurry-f: binary-functor J A B ‹Curry.uncurry F› ..
  interpret uncurry-G: functor JxA.comp B ‹Curry.uncurry G›
    using Curry.uncurry-preserves-functors τ.G.functor-axioms by simp
  interpret uncurry-G: binary-functor J A B ‹Curry.uncurry G› ..
  interpret uncurry-τ: natural-transformation
                       JxA.comp B ‹Curry.uncurry F› ‹Curry.uncurry G› ‹Curry.uncurry τ›
    using Curry.uncurry-preserves-transformations τ.natural-transformation-axioms
    by simp
  interpret uncurry-τ: binary-functor-transformation J A B
                     ‹Curry.uncurry F› ‹Curry.uncurry G› ‹Curry.uncurry τ› ..
  show ?thesis
    using assms(2) uncurry-τ.fixing-ide-gives-natural-transformation-2 by simp
qed

lemma constant-at-ide-is-constant:
assumes cone x χ and a: A.ide a
shows at a (constant-functor.map J A-B.comp x) =
      constant-functor.map J B (A-B.Map x a)
proof −
  interpret χ: cone J A-B.comp D x χ using assms(1) by auto
  have x: A-B.ide x using χ.ide-apex by auto
  interpret Fun-x: functor A B ‹A-B.Map x›
    using x A-B.ide-char by simp
  interpret Da: functor J B ‹at a D›
    using a functor-at-ide-is-functor functor-axioms by blast
  interpret Da: diagram J B ‹at a D› ..
  interpret Xa: constant-functor J B ‹A-B.Map x a›
    using a Fun-x.preserves-ide by unfold-locales simp
  show at a χ.A.map = Xa.map
    using a x Curry.uncurry-def E.map-def Xa.extensionality by auto
qed

lemma at-ide-is-diagram:
assumes a: A.ide a
shows diagram J B (at a D)
proof −
  interpret Da: functor J B at a D
    using a functor-at-ide-is-functor functor-axioms by simp
  show ?thesis ..
qed

lemma cone-at-ide-is-cone:
assumes cone x χ and a: A.ide a
shows diagram.cone J B (at a D) (A-B.Map x a) (at a χ)
proof −
  interpret χ: cone J A-B.comp D x χ using assms(1) by auto
  have x: A-B.ide x using χ.ide-apex by auto
  interpret Fun-x: functor A B ‹A-B.Map x›
```

```
    using x A-B.ide-char by simp
  interpret Da: diagram J B ‹at a D› using a at-ide-is-diagram by auto
  interpret Xa: constant-functor J B ‹A-B.Map x a›
    using a by (unfold-locales, simp)
  interpret χa: natural-transformation J B Xa.map ‹at a D› ‹at a χ›
  using assms(1) x a transformation-at-ide-is-transformation χ.natural-transformation-axioms
        constant-at-ide-is-constant
    by fastforce
  interpret χa: cone J B ‹at a D› ‹A-B.Map x a› ‹at a χ› ..
  show cone-χa: Da.cone (A-B.Map x a) (at a χ) ..
qed

lemma at-preserves-comp:
assumes A.seq a' a
shows at (A a' a) D = vertical-composite.map J B (at a D) (at a' D)
proof −
  interpret Da: natural-transformation J B ‹at (A.dom a) D› ‹at (A.cod a) D› ‹at a D›
    using assms functor-at-arr-is-transformation functor-axioms by blast
  interpret Da': natural-transformation J B ‹at (A.cod a) D› ‹at (A.cod a') D› ‹at a' D›
    using assms functor-at-arr-is-transformation [of D a'] functor-axioms by fastforce
  interpret Da'oDa: vertical-composite J B
                    ‹at (A.dom a) D› ‹at (A.cod a) D› ‹at (A.cod a') D›
                    ‹at a D› ‹at a' D› ..
  interpret Da'a: natural-transformation J B ‹at (A.dom a) D› ‹at (A.cod a') D›
                    ‹at (a' ·_A a) D›
    using assms functor-at-arr-is-transformation [of D a' ·_A a] functor-axioms by simp
  show at (a' ·_A a) D = Da'oDa.map
  proof (intro natural-transformation-eqI)
    show natural-transformation J B (at (A.dom a) D) (at (A.cod a') D) Da'oDa.map ..
    show natural-transformation J B (at (A.dom a) D) (at (A.cod a') D) (at (a' ·_A a) D) ..
    show ⋀j. J.ide j ⟹ at (a' ·_A a) D j = Da'oDa.map j
    proof −
      fix j
      assume j: J.ide j
      interpret Dj: functor A B ‹A-B.Map (D j)›
        using j preserves-ide A-B.ide-char by simp
      show at (a' ·_A a) D j = Da'oDa.map j
        using assms j Dj.preserves-comp at-simp Da'oDa.map-simp-ide by auto
    qed
  qed
qed

lemma cones-map-pointwise:
assumes cone x χ and cone x' χ'
and f: f ∈ A-B.hom x' x
shows cones-map f χ = χ' ⟷
      (∀a. A.ide a ⟶ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ')
proof
  interpret χ: cone J A-B.comp D x χ using assms(1) by auto
```

interpret χ': *cone J A-B.comp D x' χ'* **using** *assms(2)* **by** *auto*
have *x*: *A-B.ide x* **using** *χ.ide-apex* **by** *auto*
have *x'*: *A-B.ide x'* **using** *χ'.ide-apex* **by** *auto*
interpret *χf*: *cone J A-B.comp D x' ‹cones-map f χ›*
  using *x' f assms(1) cones-map-mapsto* **by** *blast*
interpret *Fun-x*: *functor A B ‹A-B.Map x›* **using** *x A-B.ide-char* **by** *simp*
interpret *Fun-x'*: *functor A B ‹A-B.Map x'›* **using** *x' A-B.ide-char* **by** *simp*
show *cones-map f χ = χ' ⟹*
      (∀ a. A.ide a ⟶ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ')
proof −
  assume *χ'*: *cones-map f χ = χ'*
  have *⋀a. A.ide a ⟹ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ'*
  proof −
    fix *a*
    assume *a*: *A.ide a*
    interpret *Da*: *diagram J B ‹at a D›* **using** *a at-ide-is-diagram* **by** *auto*
    interpret *χa*: *cone J B ‹at a D› ‹A-B.Map x a› ‹at a χ›*
      using *a assms(1) cone-at-ide-is-cone* **by** *simp*
    interpret *χ'a*: *cone J B ‹at a D› ‹A-B.Map x' a› ‹at a χ'›*
      using *a assms(2) cone-at-ide-is-cone* **by** *simp*
    have *1*: *«A-B.Map f a : A-B.Map x' a →B A-B.Map x a»*
      using *f a A-B.arr-char A-B.Map-cod A-B.Map-dom mem-Collect-eq*
          natural-transformation.preserves-hom A.ide-in-hom
      by (*metis* (*no-types*, *lifting*) *A-B.in-homE*)
    interpret *χfa*: *cone J B ‹at a D› ‹A-B.Map x' a›*
                  ‹Da.cones-map (A-B.Map f a) (at a χ)›
      using *1 χa.cone-axioms Da.cones-map-mapsto* **by** *force*
    show *Da.cones-map (A-B.Map f a) (at a χ) = at a χ'*
    proof
      fix *j*
      have *¬J.arr j ⟹ Da.cones-map (A-B.Map f a) (at a χ) j = at a χ' j*
        using *χ'a.extensionality χfa.extensionality* [*of j*] **by** *simp*
      moreover have *J.arr j ⟹ Da.cones-map (A-B.Map f a) (at a χ) j = at a χ' j*
        using *a f 1 χ.cone-axioms χa.cone-axioms at-simp*
        apply *simp*
        apply (*elim A-B.in-homE B.in-homE*, *auto*)
        using *χ' χ.A.map-simp A-B.Map-comp* [*of χ j f a a*] **by** *auto*
      ultimately show *Da.cones-map (A-B.Map f a) (at a χ) j = at a χ' j* **by** *blast*
    qed
  qed
  thus *∀ a. A.ide a ⟶ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ'*
    by *simp*
qed
show *∀ a. A.ide a ⟶ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ'*
      ⟹ cones-map f χ = χ'
proof −
  assume *A*:
    ∀ a. A.ide a ⟶ diagram.cones-map J B (at a D) (A-B.Map f a) (at a χ) = at a χ'
  show *cones-map f χ = χ'*

```
proof (intro natural-transformation-eqI)
  show natural-transformation J A-B.comp χ'.A.map D (cones-map f χ) ..
  show natural-transformation J A-B.comp χ'.A.map D χ' ..
  show ⋀j. J.ide j ⟹ cones-map f χ j = χ' j
  proof (intro A-B.arr-eqI)
    fix j
    assume j: J.ide j
    show 1: A-B.arr (cones-map f χ j)
      using j χf.preserves-reflects-arr by simp
    show A-B.arr (χ' j) using j by auto
    have Dom-χf-j: A-B.Dom (cones-map f χ j) = A-B.Map x'
      using x' j 1 A-B.Map-dom χ'.A.map-simp χf.preserves-dom J.ide-in-hom
      by (metis (no-types, lifting) J.ideD(2) χf.preserves-reflects-arr)
    also have Dom-χ'-j: ... = A-B.Dom (χ' j)
      using x' j A-B.Map-dom [of χ' j] χ'.preserves-hom χ'.A.map-simp by simp
    finally show A-B.Dom (cones-map f χ j) = A-B.Dom (χ' j) by auto
    have Cod-χf-j: A-B.Cod (cones-map f χ j) = A-B.Map (D (J.cod j))
      using j A-B.Map-cod A-B.cod-char J.ide-in-hom χf.preserves-hom
      by (metis (no-types, lifting) 1 J.ideD(1) χf.preserves-cod)
    also have Cod-χ'-j: ... = A-B.Cod (χ' j)
      using j A-B.Map-cod [of χ' j] χ'.preserves-hom by simp
    finally show A-B.Cod (cones-map f χ j) = A-B.Cod (χ' j) by auto
    show A-B.Map (cones-map f χ j) = A-B.Map (χ' j)
    proof (intro natural-transformation-eqI)
      interpret χfj: natural-transformation A B ‹A-B.Map x'› ‹A-B.Map (D (J.cod j))›
                                               ‹A-B.Map (cones-map f χ j)›
        using j χf.preserves-reflects-arr A-B.arr-char [of cones-map f χ j]
              Dom-χf-j Cod-χf-j
        by simp
      show natural-transformation A B (A-B.Map x') (A-B.Map (D (J.cod j)))
                                      (A-B.Map (cones-map f χ j)) ..
      interpret χ'j: natural-transformation A B ‹A-B.Map x'› ‹A-B.Map (D (J.cod j))›
                                               ‹A-B.Map (χ' j)›
        using j A-B.arr-char [of χ' j] Dom-χ'-j Cod-χ'-j by simp
      show natural-transformation A B (A-B.Map x') (A-B.Map (D (J.cod j)))
                                      (A-B.Map (χ' j)) ..
      show ⋀a. A.ide a ⟹ A-B.Map (cones-map f χ j) a = A-B.Map (χ' j) a
      proof −
        fix a
        assume a: A.ide a
        interpret Da: diagram J B ‹at a D› using a at-ide-is-diagram by auto
        have cone-χa: Da.cone (A-B.Map x a) (at a χ)
          using a assms(1) cone-at-ide-is-cone by simp
        interpret χa: cone J B ‹at a D› ‹A-B.Map x a› ‹at a χ›
          using cone-χa by auto
        interpret Fun-f: natural-transformation A B ‹A-B.Dom f› ‹A-B.Cod f›
                                                   ‹A-B.Map f›
          using f A-B.arr-char by fast
        have fa: A-B.Map f a ∈ B.hom (A-B.Map x' a) (A-B.Map x a)
```

```
          using a f Fun-f.preserves-hom A.ide-in-hom by auto
        have A-B.Map (cones-map f χ j) a = Da.cones-map (A-B.Map f a) (at a χ) j
        proof −
          have A-B.Map (cones-map f χ j) a = A-B.Map (A-B.comp (χ j) f) a
            using assms(1) f χ.extensionality by auto
          also have ... = B (A-B.Map (χ j) a) (A-B.Map f a)
            using f j a χ.preserves-hom A.ide-in-hom J.ide-in-hom A-B.Map-comp
                  χ.A.map-simp
            by (metis (no-types, lifting) A.comp-ide-self A.ideD(1) A-B.seqI'
                J.ideD(1) mem-Collect-eq)
          also have ... = Da.cones-map (A-B.Map f a) (at a χ) j
            using j a cone-χa fa Curry.uncurry-def E.map-simp by auto
          finally show ?thesis by auto
        qed
        also have ... = at a χ' j using j a A by simp
        also have ... = A-B.Map (χ' j) a
          using j Curry.uncurry-def E.map-simp χ'j.extensionality by simp
        finally show A-B.Map (cones-map f χ j) a = A-B.Map (χ' j) a by auto
      qed
    qed
  qed
 qed
qed
```

If χ is a cone with apex a over D, then χ is a limit cone if, for each object x of X, the cone obtained by evaluating χ at x is a limit cone with apex *A-B.Map a x* for the diagram in C obtained by evaluating D at x.

```
lemma cone-is-limit-if-pointwise-limit:
assumes cone-χ: cone x χ
and ∀a. A.ide a ⟶ diagram.limit-cone J B (at a D) (A-B.Map x a) (at a χ)
shows limit-cone x χ
proof −
  interpret χ: cone J A-B.comp D x χ using assms by auto
  have x: A-B.ide x using χ.ide-apex by auto
  show limit-cone x χ
  proof
    fix x' χ'
    assume cone-χ': cone x' χ'
    interpret χ': cone J A-B.comp D x' χ' using cone-χ' by auto
    have x': A-B.ide x' using χ'.ide-apex by auto
```

The universality of the limit cone *at a χ* yields, for each object a of A, a unique arrow *fa* that transforms *at a χ* to *at a χ'*.

```
    have EU: ⋀a. A.ide a ⟹
                ∃!fa. fa ∈ B.hom (A-B.Map x' a) (A-B.Map x a) ∧
                      diagram.cones-map J B (at a D) fa (at a χ) = at a χ'
    proof −
      fix a
```

```
    assume a: A.ide a
    interpret Da: diagram J B ‹at a D› using a at-ide-is-diagram by auto
    interpret χa: limit-cone J B ‹at a D› ‹A-B.Map x a› ‹at a χ›
      using assms(2) a by auto
    interpret χ'a: cone J B ‹at a D› ‹A-B.Map x' a› ‹at a χ'›
      using a cone-χ' cone-at-ide-is-cone by auto
    have Da.cone (A-B.Map x' a) (at a χ') ..
    thus ∃!fa. fa ∈ B.hom (A-B.Map x' a) (A-B.Map x a) ∧
               Da.cones-map fa (at a χ) = at a χ'
      using χa.is-universal by simp
  qed
```

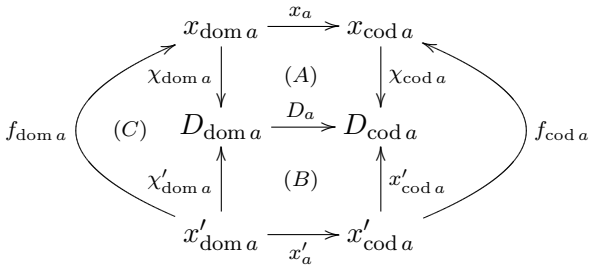
Our objective is to show the existence of a unique arrow f that transforms χ into χ'. We obtain f by bundling the arrows fa of C and proving that this yields a natural transformation from X to C, hence an arrow of $[X, C]$.

```
  show ∃!f. «f : x' →[A,B] x» ∧ cones-map f χ = χ'
  proof
    let ?P = λa fa. «fa : A-B.Map x' a →B A-B.Map x a» ∧
                    diagram.cones-map J B (at a D) fa (at a χ) = at a χ'
    have AaPa: ⋀a. A.ide a ⟹ ?P a (THE fa. ?P a fa)
    proof −
      fix a
      assume a: A.ide a
      have ∃!fa. ?P a fa using a EU by simp
      thus ?P a (THE fa. ?P a fa) using a theI' [of ?P a] by fastforce
    qed
    have AaPa-in-hom:
       ⋀a. A.ide a ⟹ «THE fa. ?P a fa : A-B.Map x' a →B A-B.Map x a»
      using AaPa by blast
    have AaPa-map:
          ⋀a. A.ide a ⟹
              diagram.cones-map J B (at a D) (THE fa. ?P a fa) (at a χ) = at a χ'
      using AaPa by blast
    let ?Fun-f = λa. if A.ide a then (THE fa. ?P a fa) else B.null
    interpret Fun-x: functor A B ‹λa. A-B.Map x a›
      using x A-B.ide-char by simp
    interpret Fun-x': functor A B ‹λa. A-B.Map x' a›
      using x' A-B.ide-char by simp
```

The arrows *Fun-f a* are the components of a natural transformation. It is more work to verify the naturality than it seems like it ought to be.

```
    interpret φ: transformation-by-components A B
                   ‹λa. A-B.Map x' a› ‹λa. A-B.Map x a› ?Fun-f
    proof
      fix a
      assume a: A.ide a
      show «?Fun-f a : A-B.Map x' a →B A-B.Map x a» using a AaPa by simp
      next
      fix a
```

assume *a*: *A.arr a*



let *?x-dom-a* = *A-B.Map x* (*A.dom a*)
let *?x-cod-a* = *A-B.Map x* (*A.cod a*)
let *?x-a* = *A-B.Map x a*
have *x-a*: «*?x-a* : *?x-dom-a* →$_B$ *?x-cod-a*»
  using *a x A-B.ide-char* **by** *auto*
let *?x′-dom-a* = *A-B.Map x′* (*A.dom a*)
let *?x′-cod-a* = *A-B.Map x′* (*A.cod a*)
let *?x′-a* = *A-B.Map x′ a*
have *x′-a*: «*?x′-a* : *?x′-dom-a* →$_B$ *?x′-cod-a*»
  using *a x′ A-B.ide-char* **by** *auto*
let *?f-dom-a* = *?Fun-f* (*A.dom a*)
let *?f-cod-a* = *?Fun-f* (*A.cod a*)
have *f-dom-a*: «*?f-dom-a* : *?x′-dom-a* →$_B$ *?x-dom-a*» **using** *a AaPa* **by** *simp*
have *f-cod-a*: «*?f-cod-a* : *?x′-cod-a* →$_B$ *?x-cod-a*» **using** *a AaPa* **by** *simp*
interpret *D-dom-a*: *diagram J B* ‹*at* (*A.dom a*) *D*› **using** *a at-ide-is-diagram* **by** *simp*
interpret *D-cod-a*: *diagram J B* ‹*at* (*A.cod a*) *D*› **using** *a at-ide-is-diagram* **by** *simp*
interpret *Da*: *natural-transformation J B* ‹*at* (*A.dom a*) *D*› ‹*at* (*A.cod a*) *D*› ‹*at a D*›
  using *a functor-axioms functor-at-arr-is-transformation* **by** *simp*
interpret *χ-dom-a*: *limit-cone J B* ‹*at* (*A.dom a*) *D*› ‹*A-B.Map x* (*A.dom a*)›
    ‹*at* (*A.dom a*) *χ*›
  using *assms*(*2*) *a* **by** *auto*
interpret *χ-cod-a*: *limit-cone J B* ‹*at* (*A.cod a*) *D*› ‹*A-B.Map x* (*A.cod a*)›
    ‹*at* (*A.cod a*) *χ*›
  using *assms*(*2*) *a* **by** *auto*
interpret *χ′-dom-a*: *cone J B* ‹*at* (*A.dom a*) *D*› ‹*A-B.Map x′* (*A.dom a*)›
    ‹*at* (*A.dom a*) *χ′*›
  using *a cone-χ′ cone-at-ide-is-cone* **by** *auto*
interpret *χ′-cod-a*: *cone J B* ‹*at* (*A.cod a*) *D*› ‹*A-B.Map x′* (*A.cod a*)›
    ‹*at* (*A.cod a*) *χ′*›
  using *a cone-χ′ cone-at-ide-is-cone* **by** *auto*

Now construct cones with apexes *x-dom-a* and *x′-dom-a* over *at* (*A.cod a*) *D* by forming the vertical composites of *at* (*A.dom a*) *χ* and *at* (*A.cod a*) *χ′* with the natural transformation *at a D*.

interpret *Daoχ-dom-a*: *vertical-composite J B*
    χ-dom-a.A.map ‹*at* (*A.dom a*) *D*› ‹*at* (*A.cod a*) *D*›
    ‹*at* (*A.dom a*) *χ*› ‹*at a D*› **..**

```
  interpret Daoχ-dom-a: cone J B ‹at (A.cod a) D› ?x-dom-a Daoχ-dom-a.map
using χ-dom-a.cone-axioms Da.natural-transformation-axioms vcomp-transformation-cone
  by metis
  interpret Daoχ'-dom-a: vertical-composite J B
                      χ'-dom-a.A.map ‹at (A.dom a) D› ‹at (A.cod a) D›
                      ‹at (A.dom a) χ'› ‹at a D› ..
  interpret Daoχ'-dom-a: cone J B ‹at (A.cod a) D› ?x'-dom-a Daoχ'-dom-a.map
using χ'-dom-a.cone-axioms Da.natural-transformation-axioms vcomp-transformation-cone
  by metis
  have Daoχ-dom-a: D-cod-a.cone ?x-dom-a Daoχ-dom-a.map ..
  have Daoχ'-dom-a: D-cod-a.cone ?x'-dom-a Daoχ'-dom-a.map ..
```

These cones are also obtained by transforming the cones $at\ (A.cod\ a)\ \chi$ and $at\ (A.cod\ a)\ \chi'$ by x-a and x'-a, respectively.

```
  have A: Daoχ-dom-a.map = D-cod-a.cones-map ?x-a (at (A.cod a) χ)
  proof
    fix j
    have ¬J.arr j ⟹ Daoχ-dom-a.map j = D-cod-a.cones-map ?x-a (at (A.cod a) χ) j
      using Daoχ-dom-a.extensionality χ-cod-a.cone-axioms x-a by force
    moreover have
        J.arr j ⟹ Daoχ-dom-a.map j = D-cod-a.cones-map ?x-a (at (A.cod a) χ) j
    proof −
      assume j: J.arr j
      have Daoχ-dom-a.map j = at a D j ·B at (A.dom a) χ (J.dom j)
        using j Daoχ-dom-a.map-simp-2 by simp
      also have ... = A-B.Map (D j) a ·B A-B.Map (χ (J.dom j)) (A.dom a)
        using a j at-simp by simp
      also have ... = A-B.Map (A-B.comp (D j) (χ (J.dom j))) a
        using a j A-B.Map-comp
        by (metis (no-types, lifting) A.comp-arr-dom χ.naturality1
            χ.preserves-reflects-arr)
      also have ... = A-B.Map (A-B.comp (χ (J.cod j)) (χ.A.map j)) a
        using a j χ.naturality by simp
      also have ... = A-B.Map (χ (J.cod j)) (A.cod a) ·B A-B.Map x a
        using a j x A-B.Map-comp
        by (metis (no-types, lifting) A.comp-cod-arr χ.A.map-simp χ.naturality2
              χ.preserves-reflects-arr)
      also have ... = at (A.cod a) χ (J.cod j) ·B A-B.Map x a
        using a j at-simp by simp
      also have ... = at (A.cod a) χ j ·B A-B.Map x a
        using a j χ-cod-a.naturality2 χ-cod-a.A.map-simp
        by (metis J.arr-cod-iff-arr J.cod-cod)
      also have ... = D-cod-a.cones-map ?x-a (at (A.cod a) χ) j
        using a j x χ-cod-a.cone-axioms preserves-cod by simp
      finally show ?thesis by blast
    qed
    ultimately show Daoχ-dom-a.map j = D-cod-a.cones-map ?x-a (at (A.cod a) χ) j
      by blast
  qed
```

```
have B: Daoχ'-dom-a.map = D-cod-a.cones-map ?x'-a (at (A.cod a) χ')
proof
  fix j
  have ¬J.arr j ⟹
         Daoχ'-dom-a.map j = D-cod-a.cones-map ?x'-a (at (A.cod a) χ') j
    using Daoχ'-dom-a.extensionality χ'-cod-a.cone-axioms x'-a by force
  moreover have
     J.arr j ⟹ Daoχ'-dom-a.map j = D-cod-a.cones-map ?x'-a (at (A.cod a) χ') j
  proof −
    assume j: J.arr j
    have Daoχ'-dom-a.map j = at a D j ·B at (A.dom a) χ' (J.dom j)
      using j Daoχ'-dom-a.map-simp-2 by simp
    also have ... = A-B.Map (D j) a ·B A-B.Map (χ' (J.dom j)) (A.dom a)
      using a j at-simp by simp
    also have ... = A-B.Map (A-B.comp (D j) (χ' (J.dom j))) a
      using a j A-B.Map-comp
      by (metis (no-types, lifting) A.comp-arr-dom χ'.naturality1
          χ'.preserves-reflects-arr)
    also have ... = A-B.Map (A-B.comp (χ' (J.cod j)) (χ'.A.map j)) a
      using a j χ'.naturality by simp
    also have ... = A-B.Map (χ' (J.cod j)) (A.cod a) ·B A-B.Map x' a
      using a j x' A-B.Map-comp
      by (metis (no-types, lifting) A.comp-cod-arr χ'.A.map-simp χ'.naturality2
            χ'.preserves-reflects-arr)
    also have ... = at (A.cod a) χ' (J.cod j) ·B A-B.Map x' a
      using a j at-simp by simp
    also have ... = at (A.cod a) χ' j ·B A-B.Map x' a
      using a j χ'-cod-a.naturality2 χ'-cod-a.A.map-simp
      by (metis J.arr-cod-iff-arr J.cod-cod)
    also have ... = D-cod-a.cones-map ?x'-a (at (A.cod a) χ') j
      using a j x' χ'-cod-a.cone-axioms preserves-cod by simp
    finally show ?thesis by blast
  qed
  ultimately show
     Daoχ'-dom-a.map j = D-cod-a.cones-map ?x'-a (at (A.cod a) χ') j
    by blast
qed
```

Next, we show that *f-dom-a*, which is the unique arrow that transforms χ*-dom-a* into χ'*-dom-a*, is also the unique arrow that transforms *Daoχ-dom-a* into *Daoχ'-dom-a*.

```
have C: D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map = Daoχ'-dom-a.map
proof (intro natural-transformation-eqI)
  show natural-transformation
         J B χ'-dom-a.A.map (at (A.cod a) D) Daoχ'-dom-a.map ..
  show natural-transformation J B χ'-dom-a.A.map (at (A.cod a) D)
         (D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map)
  proof −
    interpret κ: cone J B ‹at (A.cod a) D› ?x'-dom-a
                  ‹D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map›
```

```
    proof −
      have ⋀b b′ f. ⟦ f ∈ B.hom b′ b; D-cod-a.cone b Daoχ-dom-a.map ⟧
                      ⟹ D-cod-a.cone b′ (D-cod-a.cones-map f Daoχ-dom-a.map)
        using D-cod-a.cones-map-mapsto by blast
      moreover have D-cod-a.cone ?x-dom-a Daoχ-dom-a.map ..
      ultimately show D-cod-a.cone ?x′-dom-a
                        (D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map)
        using f-dom-a by simp
    qed
    show ?thesis ..
  qed
  show ⋀j. J.ide j ⟹
           D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map j = Daoχ′-dom-a.map j
  proof −
    fix j
    assume j: J.ide j
    have D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map j =
         Daoχ-dom-a.map j ·B ?f-dom-a
      using j f-dom-a Daoχ-dom-a.cone-axioms
      by (elim B.in-homE, auto)
    also have ... = (at a D j ·B at (A.dom a) χ j) ·B ?f-dom-a
      using j Daoχ-dom-a.map-simp-ide by simp
    also have ... = at a D j ·B at (A.dom a) χ j ·B ?f-dom-a
      using B.comp-assoc by simp
    also have ... = at a D j ·B D-dom-a.cones-map ?f-dom-a (at (A.dom a) χ) j
      using j χ-dom-a.cone-axioms f-dom-a
      by (elim B.in-homE, auto)
    also have ... = at a D j ·B at (A.dom a) χ′ j
      using a AaPa A.ide-dom by presburger
    also have ... = Daoχ′-dom-a.map j
      using j Daoχ′-dom-a.map-simp-ide by simp
    finally show
       D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map j = Daoχ′-dom-a.map j
      by auto
  qed
qed
```

Naturality amounts to showing that *C f-cod-a x′-a = C x-a f-dom-a*. To do this, we show that both arrows transform *at (A.cod a) χ* into *Daoχ′-cod-a*, thus they are equal by the universality of *at (A.cod a) χ*.

```
  have ∃!fa. «fa : ?x′-dom-a →B ?x-cod-a» ∧
           D-cod-a.cones-map fa (at (A.cod a) χ) = Daoχ′-dom-a.map
using Daoχ′-dom-a.cone-axioms a χ-cod-a.is-universal [of ?x′-dom-a Daoχ′-dom-a.map]
  by fast
  moreover have
     ?f-cod-a ·B ?x′-a ∈ B.hom ?x′-dom-a ?x-cod-a ∧
     D-cod-a.cones-map (?f-cod-a ·B ?x′-a) (at (A.cod a) χ) = Daoχ′-dom-a.map
  proof
    show ?f-cod-a ·B ?x′-a ∈ B.hom ?x′-dom-a ?x-cod-a
```

```
    using f-cod-a x'-a by blast
  show D-cod-a.cones-map (?f-cod-a ·B ?x'-a) (at (A.cod a) χ) = Daoχ'-dom-a.map
  proof −
    have D-cod-a.cones-map (?f-cod-a ·B ?x'-a) (at (A.cod a) χ)
          = restrict (D-cod-a.cones-map ?x'-a o D-cod-a.cones-map ?f-cod-a)
                     (D-cod-a.cones (?x-cod-a))
                     (at (A.cod a) χ)
      using x'-a D-cod-a.cones-map-comp [of ?f-cod-a ?x'-a] f-cod-a
      by (elim B.in-homE, auto)
    also have ... = D-cod-a.cones-map ?x'-a
                    (D-cod-a.cones-map ?f-cod-a (at (A.cod a) χ))
      using χ-cod-a.cone-axioms by simp
    also have ... = Daoχ'-dom-a.map
      using a B AaPa-map A.ide-cod by presburger
    finally show ?thesis by auto
  qed
qed
moreover have
    ?x-a ·B ?f-dom-a ∈ B.hom ?x'-dom-a ?x-cod-a ∧
    D-cod-a.cones-map (?x-a ·B ?f-dom-a) (at (A.cod a) χ) = Daoχ'-dom-a.map
proof
  show ?x-a ·B ?f-dom-a ∈ B.hom ?x'-dom-a ?x-cod-a
    using f-dom-a x-a by blast
  show D-cod-a.cones-map (?x-a ·B ?f-dom-a) (at (A.cod a) χ) = Daoχ'-dom-a.map
  proof −
    have
      D-cod-a.cones (B.cod (A-B.Map x a)) = D-cod-a.cones (A-B.Map x (A.cod a))
      using a x by simp
    moreover have B.seq ?x-a ?f-dom-a
      using f-dom-a x-a by (elim B.in-homE, auto)
    ultimately have
      D-cod-a.cones-map (?x-a ·B ?f-dom-a) (at (A.cod a) χ)
        = restrict (D-cod-a.cones-map ?f-dom-a o D-cod-a.cones-map ?x-a)
                   (D-cod-a.cones (?x-cod-a))
                   (at (A.cod a) χ)
      using D-cod-a.cones-map-comp [of ?x-a ?f-dom-a] x-a by argo
    also have ... = D-cod-a.cones-map ?f-dom-a
                    (D-cod-a.cones-map ?x-a (at (A.cod a) χ))
      using χ-cod-a.cone-axioms by simp
    also have ... = Daoχ'-dom-a.map
      using A C a AaPa by argo
    finally show ?thesis by blast
  qed
qed
ultimately show ?f-cod-a ·B ?x'-a = ?x-a ·B ?f-dom-a
  using a χ-cod-a.is-universal by blast
qed
```

The arrow from x' to x in $[A, B]$ determined by the natural transformation φ transforms χ into χ'. Moreover, it is the unique such arrow, since the components of φ are

each determined by universality.

```
let ?f = A-B.MkArr (λa. A-B.Map x' a) (λa. A-B.Map x a) φ.map
have f-in-hom: ?f ∈ A-B.hom x' x
proof −
  have arr-f: A-B.arr ?f
    using x' x A-B.arr-MkArr φ.natural-transformation-axioms by simp
  moreover have A-B.MkIde (λa. A-B.Map x a) = x
    using x A-B.ide-char A-B.MkArr-Map A-B.in-homE A-B.ide-in-hom by metis
  moreover have A-B.MkIde (λa. A-B.Map x' a) = x'
    using x' A-B.ide-char A-B.MkArr-Map A-B.in-homE A-B.ide-in-hom by metis
  ultimately show ?thesis
    using A-B.dom-char A-B.cod-char by auto
qed
have Fun-f: ⋀a. A.ide a ⟹ A-B.Map ?f a = (THE fa. ?P a fa)
  using f-in-hom φ.map-simp-ide by fastforce
have cones-map-f: cones-map ?f χ = χ'
  using AaPa Fun-f at-ide-is-diagram assms(2) x x' cone-χ cone-χ' f-in-hom Fun-f
        cones-map-pointwise
  by presburger
show «?f : x' →[A,B] x» ∧ cones-map ?f χ = χ' using f-in-hom cones-map-f by auto
show ⋀f'. «f' : x' →[A,B] x» ∧ cones-map f' χ = χ' ⟹ f' = ?f
proof −
  fix f'
  assume f': «f' : x' →[A,B] x» ∧ cones-map f' χ = χ'
  have 0: ⋀a. A.ide a ⟹
              diagram.cones-map J B (at a D) (A-B.Map f' a) (at a χ) = at a χ'
    using f' cone-χ cone-χ' cones-map-pointwise by blast
  have f' = A-B.MkArr (A-B.Dom f') (A-B.Cod f') (A-B.Map f')
    using f' A-B.MkArr-Map by auto
  also have ... = ?f
  proof (intro A-B.MkArr-eqI)
    show 1: A-B.Dom f' = A-B.Map x' using f' A-B.Map-dom by auto
    show 2: A-B.Cod f' = A-B.Map x using f' A-B.Map-cod by auto
    show A-B.Map f' = φ.map
    proof (intro natural-transformation-eqI)
      show natural-transformation A B (A-B.Map x') (A-B.Map x) φ.map ..
      show natural-transformation A B (A-B.Map x') (A-B.Map x) (A-B.Map f')
        using f' 1 2 A-B.arr-char [of f'] by auto
      show ⋀a. A.ide a ⟹ A-B.Map f' a = φ.map a
      proof −
        fix a
        assume a: A.ide a
        interpret Da: diagram J B ‹at a D› using a at-ide-is-diagram by auto
        interpret Fun-f': natural-transformation A B ‹A-B.Dom f'› ‹A-B.Cod f'›
                                                     ‹A-B.Map f'›
          using f' A-B.arr-char by fast
        have A-B.Map f' a ∈ B.hom (A-B.Map x' a) (A-B.Map x a)
          using a f' Fun-f'.preserves-hom A.ide-in-hom by auto
        hence ?P a (A-B.Map f' a) using a 0 [of a] by simp
```

```
            moreover have ?P a (φ.map a)
              using a φ.map-simp-ide Fun-f AaPa by presburger
            ultimately show A-B.Map f' a = φ.map a using a EU by blast
          qed
        qed
      qed
      finally show f' = ?f by auto
    qed
  qed
 qed

end

context functor-category
begin
```

A functor category $[A, B]$ has limits of shape J whenever $(\cdot_B)$ has limits of shape J.

```
 lemma has-limits-of-shape-if-target-does:
 assumes category (J :: 'j comp)
 and B.has-limits-of-shape J
 shows has-limits-of-shape J
 proof (unfold has-limits-of-shape-def)
  have ⋀D. diagram J comp D ⟹ (∃x χ. limit-cone J comp D x χ)
  proof −
   fix D
   assume D: diagram J comp D
   interpret J: category J using assms(1) by auto
   interpret JxA: product-category J A ..
   interpret D: diagram J comp D using D by auto
   interpret D: diagram-in-functor-category A B J D ..
   interpret Curry: currying J A B ..
```

Given diagram D in $[A, B]$, choose for each object a of A a limit cone $(la, \chi a)$ for $at\ a\ D$ in B.

```
   let ?l = λa. diagram.some-limit J B (D.at a D)
   let ?χ = λa. diagram.some-limit-cone J B (D.at a D)
   have lχ: ⋀a. A.ide a ⟹ diagram.limit-cone J B (D.at a D) (?l a) (?χ a)
    using B.has-limits-of-shape-def D.at-ide-is-diagram assms(2)
          diagram.limit-cone-some-limit-cone
    by blast
```

The choice of limit cones induces a limit functor from A to B.

```
   interpret uncurry-D: diagram JxA.comp B Curry.uncurry D
   proof −
    interpret functor JxA.comp B ‹Curry.uncurry D›
     using D.functor-axioms Curry.uncurry-preserves-functors by simp
    interpret binary-functor J A B ‹Curry.uncurry D› ..
    show diagram JxA.comp B (Curry.uncurry D) ..
```

qed
interpret *uncurry-D*: *parametrized-diagram J A B* ‹*Curry.uncurry D*› **..**
let *?L* = *uncurry-D.L ?l ?χ*
let *?P* = *uncurry-D.P ?l ?χ*
interpret *L*: *functor A B ?L*
using *lχ uncurry-D.chosen-limits-induce-functor* [*of ?l ?χ*] **by** *simp*
have *L-ide*: ⋀*a*. *A.ide a* ⟹ *?L a* = *?l a*
using *uncurry-D.L-ide* [*of ?l ?χ*] *lχ* **by** *blast*
have *L-arr*: ⋀*a*. *A.arr a* ⟹ (∃!*f*. *?P a f*) ∧ *?P a* (*?L a*)
using *uncurry-D.L-arr* [*of ?l ?χ*] *lχ* **by** *blast*

The functor L extends to a functor L' from JxA to B that is constant on J.

let *?L′* = *λja*. *if JxA.arr ja then ?L* (*snd ja*) *else B.null*
let *?P′* = *λja*. *?P* (*snd ja*)
interpret *L′*: *functor JxA.comp B ?L′*
apply *unfold-locales*
using *L.preserves-arr L.preserves-dom L.preserves-cod*
apply *auto*[4]
using *L.preserves-comp JxA.comp-char* **by** (*elim JxA.seqE*, *auto*)
have ⋀*ja*. *JxA.arr ja* ⟹ (∃!*f*. *?P′ ja f*) ∧ *?P′ ja* (*?L′ ja*)
proof −
fix *ja*
assume *ja*: *JxA.arr ja*
have *A.arr* (*snd ja*) **using** *ja* **by** *blast*
thus (∃!*f*. *?P′ ja f*) ∧ *?P′ ja* (*?L′ ja*)
using *ja L-arr* **by** *presburger*
qed
hence *L′-arr*: ⋀*ja*. *JxA.arr ja* ⟹ *?P′ ja* (*?L′ ja*) **by** *blast*
have *L′-ide*: ⋀*ja*. ⟦ *J.arr* (*fst ja*); *A.ide* (*snd ja*) ⟧ ⟹ *?L′ ja* = *?l* (*snd ja*)
using *L-ide lχ* **by** *force*
have *L′-arr-map*:
⋀*ja*. *JxA.arr ja* ⟹ *uncurry-D.P ?l ?χ* (*snd ja*) (*uncurry-D.L ?l ?χ* (*snd ja*))
using *L′-arr* **by** *presburger*

The map that takes an object $(j,\ a)$ of JxA to the component $\chi\ a\ j$ of the limit cone $\chi\ a$ is a natural transformation from L to uncurry D.

let *?χ′* = *λja*. *?χ* (*snd ja*) (*fst ja*)
interpret *χ′*: *transformation-by-components JxA.comp B ?L′* ‹*Curry.uncurry D*› *?χ′*
proof
fix *ja*
assume *ja*: *JxA.ide ja*
let *?j* = *fst ja*
let *?a* = *snd ja*
interpret *χa*: *limit-cone J B* ‹*D.at ?a D*› ‹*?l ?a*› ‹*?χ ?a*›
using *ja lχ* **by** *blast*
show «*?χ′ ja* : *?L′ ja* →*B* *Curry.uncurry D ja*»
using *ja L′-ide* [*of ja*] **by** *force*
next
fix *ja*

```
    assume ja: JxA.arr ja
    let ?j = fst ja
    let ?a = snd ja
    have j: J.arr ?j using ja by simp
    have a: A.arr ?a using ja by simp
    interpret D-dom-a: diagram J B ‹D.at (A.dom ?a) D›
      using a D.at-ide-is-diagram by auto
    interpret D-cod-a: diagram J B ‹D.at (A.cod ?a) D›
      using a D.at-ide-is-diagram by auto
    interpret Da: natural-transformation J B
                  ‹D.at (A.dom ?a) D› ‹D.at (A.cod ?a) D› ‹D.at ?a D›
      using a D.functor-axioms D.functor-at-arr-is-transformation by simp
    interpret χ-dom-a: limit-cone J B ‹D.at (A.dom ?a) D› ‹?l (A.dom ?a)›
                       ‹?χ (A.dom ?a)›
      using a lχ by simp
    interpret χ-cod-a: limit-cone J B ‹D.at (A.cod ?a) D› ‹?l (A.cod ?a)›
                       ‹?χ (A.cod ?a)›
      using a lχ by simp
    interpret Daoχ-dom-a: vertical-composite J B
                          χ-dom-a.A.map ‹D.at (A.dom ?a) D› ‹D.at (A.cod ?a) D›
                          ‹?χ (A.dom ?a)› ‹D.at ?a D›
      ..
  interpret Daoχ-dom-a: cone J B ‹D.at (A.cod ?a) D› ‹?l (A.dom ?a)› Daoχ-dom-a.map
..
    show ?χ' (JxA.cod ja) ·B ?L' ja = B (Curry.uncurry D ja) (?χ' (JxA.dom ja))
    proof −
      have ?χ' (JxA.cod ja) ·B ?L' ja = ?χ (A.cod ?a) (J.cod ?j) ·B ?L' ja
        using ja by fastforce
      also have ... = D-cod-a.cones-map (?L' ja) (?χ (A.cod ?a)) (J.cod ?j)
        using ja L'-arr-map [of ja] χ-cod-a.cone-axioms by auto
      also have ... = Daoχ-dom-a.map (J.cod ?j)
        using ja χ-cod-a.induced-arrowI Daoχ-dom-a.cone-axioms L'-arr by presburger
      also have ... = D.at ?a D (J.cod ?j) ·B D-dom-a.some-limit-cone (J.cod ?j)
        using ja Daoχ-dom-a.map-simp-ide by fastforce
      also have ... = D.at ?a D (J.cod ?j) ·B D.at (A.dom ?a) D ?j ·B ?χ' (JxA.dom ja)
        using ja χ-dom-a.naturality χ-dom-a.ide-apex apply simp
        by (metis B.comp-arr-ide χ-dom-a.preserves-reflects-arr)
      also have ... = (D.at ?a D (J.cod ?j) ·B D.at (A.dom ?a) D ?j) ·B ?χ' (JxA.dom ja)
        using j ja B.comp-assoc by presburger
      also have ... = B (D.at ?a D ?j) (?χ' (JxA.dom ja))
        using a j ja Map-comp A.comp-arr-dom D.as-nat-trans.naturality2 by simp
      also have ... = Curry.uncurry D ja ·B ?χ' (JxA.dom ja)
        using Curry.uncurry-def by simp
      finally show ?thesis by auto
    qed
  qed
```

Since χ' is constant on J, *curry* χ' is a cone over D.

```
  interpret constL: constant-functor J comp ‹MkIde ?L›
```

```
    using L.as-nat-trans.natural-transformation-axioms MkArr-in-hom ide-in-hom
        L.functor-axioms
    by unfold-locales blast

  have curry-L': constL.map = Curry.curry ?L' ?L' ?L'
  proof
    fix j
    have ¬J.arr j ⟹ constL.map j = Curry.curry ?L' ?L' ?L' j
      using Curry.curry-def constL.extensionality by simp
    moreover have J.arr j ⟹ constL.map j = Curry.curry ?L' ?L' ?L' j
      using Curry.curry-def constL.value-is-ide in-homE ide-in-hom by auto
    ultimately show constL.map j = Curry.curry ?L' ?L' ?L' j by blast
  qed
  hence uncurry-constL: Curry.uncurry constL.map = ?L'
    using L'.as-nat-trans.natural-transformation-axioms Curry.uncurry-curry by simp
  interpret curry-χ': natural-transformation J comp constL.map D
                      ‹Curry.curry ?L' (Curry.uncurry D) χ'.map›
  proof −
    have Curry.curry (Curry.uncurry D) (Curry.uncurry D) (Curry.uncurry D) = D
   using Curry.curry-uncurry D.functor-axioms D.as-nat-trans.natural-transformation-axioms
      by blast
    thus natural-transformation J comp constL.map D
          (Curry.curry ?L' (Curry.uncurry D) χ'.map)
     using Curry.curry-preserves-transformations curry-L' χ'.natural-transformation-axioms
      by force
  qed
    interpret curry-χ': cone J comp D ‹MkIde ?L› ‹Curry.curry ?L' (Curry.uncurry D)
χ'.map›
    ..
```

The value of *curry-χ'* at each object a of A is the limit cone $\chi\ a$, hence *curry-χ'* is a limit cone.

```
  have 1: ⋀a. A.ide a ⟹ D.at a (Curry.curry ?L' (Curry.uncurry D) χ'.map) = ?χ a
  proof −
    fix a
    assume a: A.ide a
    have D.at a (Curry.curry ?L' (Curry.uncurry D) χ'.map) =
          (λj. Curry.uncurry (Curry.curry ?L' (Curry.uncurry D) χ'.map) (j, a))
      using a by simp
    moreover have ... = (λj. χ'.map (j, a))
      using a Curry.uncurry-curry χ'.natural-transformation-axioms by simp
    moreover have ... = ?χ a
    proof (intro natural-transformation-eqI)
      interpret χa: limit-cone J B ‹D.at a D› ‹?l a› ‹?χ a› using a lχ by simp
      interpret χ': binary-functor-transformation J A B ?L' ‹Curry.uncurry D› χ'.map ..
      show natural-transformation J B χa.A.map (D.at a D) (?χ a) ..
      show natural-transformation J B χa.A.map (D.at a D) (λj. χ'.map (j, a))
      proof −
        have χa.A.map = (λj. ?L' (j, a))
```

```
        using a χa.A.map-def L'-ide by auto
      thus ?thesis
        using a χ'.fixing-ide-gives-natural-transformation-2 by simp
    qed
    fix j
    assume j: J.ide j
    show χ'.map (j, a) = ?χ a j
      using a j χ'.map-simp-ide by simp
  qed
  ultimately show D.at a (Curry.curry ?L' (Curry.uncurry D) χ'.map) = ?χ a by simp
qed
hence 2: ⋀a. A.ide a ⟹ diagram.limit-cone J B (D.at a D) (?l a)
                         (D.at a (Curry.curry ?L' (Curry.uncurry D) χ'.map))
  using lχ by simp
hence limit-cone J comp D (MkIde ?L) (Curry.curry ?L' (Curry.uncurry D) χ'.map)
  using 1 2 L.functor-axioms L-ide curry-χ'.cone-axioms curry-L'
        D.cone-is-limit-if-pointwise-limit
  by simp
thus ∃x χ. limit-cone J comp D x χ by blast
qed
thus ∀D. diagram J comp D ⟶ (∃x χ. limit-cone J comp D x χ) by blast
qed

lemma has-limits-if-target-does:
assumes B.has-limits (undefined :: 'j)
shows has-limits (undefined :: 'j)
  using assms B.has-limits-def has-limits-def has-limits-of-shape-if-target-does by fast

end
```

20.10 The Yoneda Functor Preserves Limits

In this section, we show that the Yoneda functor from C to $[Cop, S]$ preserves limits.

```
context yoneda-functor
begin

lemma preserves-limits:
fixes J :: 'j comp
assumes diagram J C D and diagram.has-as-limit J C D a
shows diagram.has-as-limit J Cop-S.comp (map o D) (map a)
proof −
```

The basic idea of the proof is as follows: If χ is a limit cone in C, then for every object a' of Cop the evaluation of $Y\ o\ \chi$ at a' is a limit cone in S. By the results on limits in functor categories, this implies that $Y\ o\ \chi$ is a limit cone in $[Cop, S]$.

```
  interpret J: category J using assms(1) diagram-def by auto
  interpret D: diagram J C D using assms(1) by auto
  from assms(2) obtain χ where χ: D.limit-cone a χ by blast
```

interpret χ: *limit-cone J C D a* χ **using** χ **by** *auto*
have *a*: *C.ide a* **using** χ*.ide-apex* **by** *auto*
interpret *YoD*: *diagram J Cop-S.comp ‹map o D›*
using *D.diagram-axioms functor-axioms preserves-diagrams* [*of J D*] **by** *simp*
interpret *YoD*: *diagram-in-functor-category Cop.comp S J ‹map o D›* **..**
interpret *Yo*χ: *cone J Cop-S.comp ‹map o D› ‹map a› ‹map o* χ*›*
using χ*.cone-axioms preserves-cones* **by** *blast*
have $\bigwedge a'$. *C.ide a'* $\Longrightarrow$
limit-cone J S (*YoD.at a'* (*map o D*))
(*Cop-S.Map* (*map a*) *a'*) (*YoD.at a'* (*map o* χ))
proof −
fix *a'*
assume *a'*: *C.ide a'*
interpret *A'*: *constant-functor J C a'*
using *a'* **by** (*unfold-locales*, *auto*)
interpret *YoD-a'*: *diagram J S ‹YoD.at a' (map o D)›*
using *a' YoD.at-ide-is-diagram* **by** *simp*
interpret *Yo*χ*-a'*: *cone J S ‹YoD.at a' (map o D)›*
‹Cop-S.Map (map a) a'› ‹YoD.at a' (map o χ*)›*
using *a' YoD.cone-at-ide-is-cone Yo*χ*.cone-axioms* **by** *fastforce*
have *eval-at-ide*: $\bigwedge j$. *J.ide j* $\Longrightarrow$ *YoD.at a'* (*map* $\circ$ *D*) *j* = *Hom.map* (*a'*, *D j*)
proof −
fix *j*
assume *j*: *J.ide j*
have *YoD.at a'* (*map* $\circ$ *D*) *j* = *Cop-S.Map* (*map* (*D j*)) *a'*
using *a' j YoD.at-simp YoD.preserves-arr* [*of j*] **by** *auto*
also have ... = *Y* (*D j*) *a'* **using** *Y-def* **by** *simp*
also have ... = *Hom.map* (*a'*, *D j*) **using** *a' j D.preserves-arr* **by** *simp*
finally show *YoD.at a'* (*map* $\circ$ *D*) *j* = *Hom.map* (*a'*, *D j*) **by** *auto*
qed
have *eval-at-arr*: $\bigwedge j$. *J.arr j* $\Longrightarrow$ *YoD.at a'* (*map* $\circ$ χ) *j* = *Hom.map* (*a'*, χ *j*)
proof −
fix *j*
assume *j*: *J.arr j*
have *YoD.at a'* (*map* $\circ$ χ) *j* = *Cop-S.Map* ((*map o* χ) *j*) *a'*
using *a' j YoD.at-simp* [*of a' j map o* χ] *preserves-arr* **by** *fastforce*
also have ... = *Y* (χ *j*) *a'* **using** *Y-def* **by** *simp*
also have ... = *Hom.map* (*a'*, χ *j*) **using** *a' j* **by** *simp*
finally show *YoD.at a'* (*map* $\circ$ χ) *j* = *Hom.map* (*a'*, χ *j*) **by** *auto*
qed
have *Fun-map-a-a'*: *Cop-S.Map* (*map a*) *a'* = *Hom.map* (*a'*, *a*)
using *a a' map-simp preserves-arr* [*of a*] **by** *simp*
show *limit-cone J S* (*YoD.at a'* (*map o D*))
(*Cop-S.Map* (*map a*) *a'*) (*YoD.at a'* (*map o* χ))
proof
fix *x* σ
assume σ: *YoD-a'.cone x* σ
interpret σ: *cone J S ‹YoD.at a' (map o D)› x* σ **using** σ **by** *auto*
have *x*: *S.ide x* **using** σ*.ide-apex* **by** *simp*

For each object j of J, the component $\sigma\ j$ is an arrow in $S.hom\ x\ (Hom.map\ (a',\ D\ j))$. Each element $e \in S.set\ x$ therefore determines an arrow $\psi\ (a',\ D\ j)\ (S.Fun\ (\sigma\ j)\ e) \in C.hom\ a'\ (D\ j)$. These arrows are the components of a cone $\kappa\ e$ over D with apex a'.

```
have σj: ⋀j. J.ide j ⟹ «σ j : x →S Hom.map (a', D j)»
  using eval-at-ide σ.preserves-hom J.ide-in-hom by force
have κ: ⋀e. e ∈ S.set x ⟹
            transformation-by-components
              J C A'.map D (λj. ψ (a', D j) (S.Fun (σ j) e))
proof −
  fix e
  assume e: e ∈ S.set x
  show transformation-by-components J C A'.map D (λj. ψ (a', D j) (S.Fun (σ j) e))
  proof
    fix j
    assume j: J.ide j
    show «ψ (a', D j) (S.Fun (σ j) e) : A'.map j → D j»
      using e j S.Fun-mapsto [of σ j] A'.preserves-ide Hom.set-map eval-at-ide
            Hom.ψ-mapsto [of A'.map j D j]
      by force
    next
    fix j
    assume j: J.arr j
    show ψ (a', D (J.cod j)) (S.Fun (σ (J.cod j)) e) · A'.map j =
         D j · ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e)
    proof −
      have ψ (a', D (J.cod j)) (S.Fun (σ (J.cod j)) e) · A'.map j =
           ψ (a', D (J.cod j)) (S.Fun (σ (J.cod j)) e) · a'
        using A'.map-simp j by simp
      also have ... = ψ (a', D (J.cod j)) (S.Fun (σ (J.cod j)) e)
      proof −
        have ψ (a', D (J.cod j)) (S.Fun (σ (J.cod j)) e) ∈ C.hom a' (D (J.cod j))
          using a' e j Hom.ψ-mapsto [of A'.map j D (J.cod j)] A'.map-simp
                S.Fun-mapsto [of σ (J.cod j)] Hom.set-map eval-at-ide
          by auto
        thus ?thesis
          using C.comp-arr-dom by fastforce
      qed
      also have ... = ψ (a', D (J.cod j)) (S.Fun (Y (D j) a') (S.Fun (σ (J.dom j)) e))
      proof −
        have S.Fun (Y (D j) a') (S.Fun (σ (J.dom j)) e) =
             (S.Fun (Y (D j) a') o S.Fun (σ (J.dom j))) e
          by simp
        also have ... = S.Fun (Y (D j) a' ·S σ (J.dom j)) e
          using a' e j Y-arr-ide(1) S.in-homE σj eval-at-ide S.Fun-comp by force
        also have ... = S.Fun (σ (J.cod j)) e
          using a' j x σ.naturality2 σ.A.map-simp S.comp-arr-dom J.arr-cod-iff-arr
                J.cod-cod YoD.preserves-arr σ.naturality1 YoD.at-simp
          by auto
        finally have
```

```
        S.Fun (Y (D j) a') (S.Fun (σ (J.dom j)) e) = S.Fun (σ (J.cod j)) e
       by auto
     thus ?thesis by simp
   qed
   also have ... = D j · ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e)
   proof −
     have S.Fun (Y (D j) a') (S.Fun (σ (J.dom j)) e) =
           φ (a', D (J.cod j)) (D j · ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e))
     proof −
       have S.Fun (σ (J.dom j)) e ∈ Hom.set (a', D (J.dom j))
         using a' e j σj S.Fun-mapsto [of σ (J.dom j)] Hom.set-map
               YoD.at-simp eval-at-ide
         by auto
       moreover have C.arr (ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e)) ∧
                     C.dom (ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e)) = a'
         using a' e j σj S.Fun-mapsto [of σ (J.dom j)] Hom.set-map eval-at-ide
               Hom.ψ-mapsto [of a' D (J.dom j)]
         by auto
       ultimately show ?thesis
         using a' e j Hom.Fun-map C.comp-arr-dom by force
     qed
     moreover have D j · ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e)
                     ∈ C.hom a' (D (J.cod j))
     proof −
       have ψ (a', D (J.dom j)) (S.Fun (σ (J.dom j)) e) ∈ C.hom a' (D (J.dom j))
         using a' e j Hom.ψ-mapsto [of a' D (J.dom j)] eval-at-ide
               S.Fun-mapsto [of σ (J.dom j)] Hom.set-map
         by auto
       thus ?thesis using j D.preserves-hom by blast
     qed
     ultimately show ?thesis using a' j Hom.ψ-φ by simp
   qed
   finally show ?thesis by auto
  qed
 qed
 let ?κ = λe. transformation-by-components.map J C A'.map
             (λj. ψ (a', D j) (S.Fun (σ j) e))
 have cone-κe: ⋀e. e ∈ S.set x ⟹ D.cone a' (?κ e)
 proof −
   fix e
   assume e: e ∈ S.set x
   interpret κe: transformation-by-components J C A'.map D
               ‹λj. ψ (a', D j) (S.Fun (σ j) e)›
     using e κ by blast
   show D.cone a' (?κ e) ..
 qed
```

Since $\kappa\ e$ is a cone for each element e of $S.set\ x$, by the universal property of the limit cone χ there is a unique arrow $fe \in C.hom\ a'\ a$ that transforms χ to $\kappa\ e$.

have *ex-fe*: $\bigwedge e.\ e \in S.set\ x \Longrightarrow \exists!fe.\ «fe : a' \to a» \wedge D.cones\text{-}map\ fe\ \chi = ?\kappa\ e$
using *cone-κe χ.is-universal* **by** *simp*

The map taking $e \in S.set\ x$ to $fe \in C.hom\ a'\ a$ determines an arrow $f \in S.hom\ x\ (Hom\ (a',\ a))$ that transforms the cone obtained by evaluating $Y\ o\ \chi$ at a' to the cone σ.

let *?f* = *S.mkArr* (*S.set x*) (*Hom.set* (*a'*, *a*))
(λ*e*. *φ* (*a'*, *a*) (*χ.induced-arrow a'* (*?κ e*)))
have *0*: (λ*e*. *φ* (*a'*, *a*) (*χ.induced-arrow a'* (*?κ e*))) ∈ *S.set x* → *Hom.set* (*a'*, *a*)
proof
fix *e*
assume *e*: *e* ∈ *S.set x*
interpret *κe*: *cone J C D a'* ‹*?κ e*› **using** *e cone-κe* **by** *simp*
have *χ.induced-arrow a'* (*?κ e*) ∈ *C.hom a' a*
using *a a' e ex-fe χ.induced-arrowI κe.cone-axioms* **by** *simp*
thus *φ* (*a'*, *a*) (*χ.induced-arrow a'* (*?κ e*)) ∈ *Hom.set* (*a'*, *a*)
using *a a' Hom.φ-mapsto* **by** *auto*
qed
have *f*: «*?f* : *x* →$_S$ *Hom.map* (*a'*, *a*)»
proof −
have (λ*e*. *φ* (*a'*, *a*) (*χ.induced-arrow a'* (*?κ e*))) ∈ *S.set x* → *Hom.set* (*a'*, *a*)
proof
fix *e*
assume *e*: *e* ∈ *S.set x*
interpret *κe*: *cone J C D a'* ‹*?κ e*› **using** *e cone-κe* **by** *simp*
have *χ.induced-arrow a'* (*?κ e*) ∈ *C.hom a' a*
using *a a' e ex-fe χ.induced-arrowI κe.cone-axioms* **by** *simp*
thus *φ* (*a'*, *a*) (*χ.induced-arrow a'* (*?κ e*)) ∈ *Hom.set* (*a'*, *a*)
using *a a' Hom.φ-mapsto* **by** *auto*
qed
moreover have *setp* (*Hom.set* (*a'*, *a*))
using *a a' Hom.small-homs*
by (*metis Fun-map-a-a' Hom.map-ide S.arr-mkIde S.ideD*(*1*) *Yoχ-a'.ide-apex*)
ultimately show *?thesis*
using *a a' x σ.ide-apex S.mkArr-in-hom* [*of S.set x Hom.set* (*a'*, *a*)]
Hom.set-subset-Univ S.mkIde-set
by *simp*
qed
have *YoD-a'.cones-map ?f* (*YoD.at a'* (*map o χ*)) = *σ*
proof (*intro natural-transformation-eqI*)
show *natural-transformation J S σ.A.map* (*YoD.at a'* (*map o D*)) *σ*
using *σ.natural-transformation-axioms* **by** *auto*
have *1*: *S.cod ?f* = *Cop-S.Map* (*map a*) *a'*
using *f Fun-map-a-a'* **by** *force*
interpret *YoD-a'of*: *cone J S* ‹*YoD.at a'* (*map o D*)› *x*
‹*YoD-a'.cones-map ?f* (*YoD.at a'* (*map o χ*))›
proof −
have *YoD-a'.cone* (*S.cod ?f*) (*YoD.at a'* (*map o χ*))
using *a a' f Yoχ-a'.cone-axioms preserves-arr* [*of a*] **by** *auto*

```
  hence YoD-a'.cone (S.dom ?f) (YoD-a'.cones-map ?f (YoD.at a' (map o χ)))
    using f YoD-a'.cones-map-mapsto S.arrI by blast
  thus cone J S (YoD.at a' (map o D)) x
                        (YoD-a'.cones-map ?f (YoD.at a' (map o χ)))
    using f by auto
qed
show natural-transformation J S σ.A.map (YoD.at a' (map o D))
                          (YoD-a'.cones-map ?f (YoD.at a' (map o χ))) ..
fix j
assume j: J.ide j
have YoD-a'.cones-map ?f (YoD.at a' (map o χ)) j = YoD.at a' (map o χ) j ·S ?f
  using f j Fun-map-a-a' Yoχ-a'.cone-axioms by fastforce
also have ... = σ j
proof (intro S.arr-eqI_SC)
  show S.par (YoD.at a' (map o χ) j ·S ?f) (σ j)
    using 1 f j x YoD-a'.preserves-hom by fastforce
  show S.Fun (YoD.at a' (map o χ) j ·S ?f) = S.Fun (σ j)
  proof
    fix e
    have e ∉ S.set x ⟹ S.Fun (YoD.at a' (map o χ) j ·S ?f) e = S.Fun (σ j) e
      using 1 f j x S.Fun-mapsto [of σ j] σ.A.map-simp
            extensional-arb [of S.Fun (σ j)]
      by auto
    moreover have e ∈ S.set x ⟹
                    S.Fun (YoD.at a' (map o χ) j ·S ?f) e = S.Fun (σ j) e
    proof −
      assume e: e ∈ S.set x
      interpret κe: transformation-by-components J C A'.map D
                    ‹λj. ψ (a', D j) (S.Fun (σ j) e)›
        using e κ by blast
      interpret κe: cone J C D a' ‹?κ e› using e cone-κe by simp
      have induced-arrow: χ.induced-arrow a' (?κ e) ∈ C.hom a' a
        using a a' e ex-fe χ.induced-arrowI κe.cone-axioms by simp
      have S.Fun (YoD.at a' (map o χ) j ·S ?f) e =
            restrict (S.Fun (YoD.at a' (map o χ) j) o S.Fun ?f) (S.set x) e
        using 1 e f j S.Fun-comp YoD-a'.preserves-hom by force
      also have ... = (φ (a', D j) o C (χ j) o ψ (a', a)) (S.Fun ?f e)
        using j a' f e Hom.map-simp-2 S.Fun-mkArr Hom.preserves-arr [of (a', χ j)]
              eval-at-arr
        by (elim S.in-homE, auto)
      also have ... = (φ (a', D j) o C (χ j) o ψ (a', a))
                      (φ (a', a) (χ.induced-arrow a' (?κ e)))
        using e f S.Fun-mkArr by fastforce
      also have ... = φ (a', D j) (D.cones-map (χ.induced-arrow a' (?κ e)) χ j)
          using a a' e j 0 Hom.ψ-φ induced-arrow χ.cone-axioms by auto
      also have ... = φ (a', D j) (?κ e j)
        using χ.induced-arrowI κe.cone-axioms by fastforce
      also have ... = φ (a', D j) (ψ (a', D j) (S.Fun (σ j) e))
        using j κe.map-def [of j] by simp
```

```
      also have ... = S.Fun (σ j) e
      proof −
        have S.Fun (σ j) e ∈ Hom.set (a′, D j)
          using a′ e j S.Fun-mapsto [of σ j] eval-at-ide Hom.set-map by auto
        thus ?thesis
          using a′ j Hom.φ-ψ C.ide-in-hom J.ide-in-hom by blast
      qed
      finally show S.Fun (YoD.at a′ (map o χ) j ·S ?f) e = S.Fun (σ j) e
        by auto
    qed
    ultimately show S.Fun (YoD.at a′ (map o χ) j ·S ?f) e = S.Fun (σ j) e
      by auto
  qed
  finally show YoD-a′.cones-map ?f (YoD.at a′ (map o χ)) j = σ j by auto
qed
hence ff: ?f ∈ S.hom x (Hom.map (a′, a)) ∧
          YoD-a′.cones-map ?f (YoD.at a′ (map o χ)) = σ
  using f by auto
```

Any other arrow $f' \in S.hom\ x\ (Hom.map\ (a',\ a))$ that transforms the cone obtained by evaluating $Y\ o\ \chi$ at a' to the cone σ, must equal f, showing that f is unique.

```
moreover have ⋀f′. «f′ : x →S Hom.map (a′, a)» ∧
                   YoD-a′.cones-map f′ (YoD.at a′ (map o χ)) = σ
                     ⟹ f′ = ?f
proof −
  fix f′
  assume f′: «f′ : x →S Hom.map (a′, a)» ∧
             YoD-a′.cones-map f′ (YoD.at a′ (map o χ)) = σ
  show f′ = ?f
  proof (intro S.arr-eqI_SC)
    show par: S.par f′ ?f using f f′ by (elim S.in-homE, auto)
    show S.Fun f′ = S.Fun ?f
    proof
      fix e
      have e ∉ S.set x ⟹ S.Fun f′ e = S.Fun ?f e
        using f f′ S.Fun-in-terms-of-comp by fastforce
      moreover have e ∈ S.set x ⟹ S.Fun f′ e = S.Fun ?f e
      proof −
        assume e: e ∈ S.set x
        have fe: S.Fun ?f e ∈ Hom.set (a′, a)
          using e f par by auto
        have f′e: S.Fun f′ e ∈ Hom.set (a′, a)
          using a a′ e f′ S.Fun-mapsto Hom.set-map by fastforce
        have 1: «ψ (a′, a) (S.Fun f′ e) : a′ → a»
          using a a′ e f′ f′e S.Fun-mapsto Hom.ψ-mapsto Hom.set-map by blast
        have 2: «ψ (a′, a) (S.Fun ?f e) : a′ → a»
          using a a′ e f′ fe S.Fun-mapsto Hom.ψ-mapsto Hom.set-map by blast
        interpret χofe: cone J C D a′ ‹D.cones-map (ψ (a′, a) (S.Fun ?f e)) χ›
```

```
proof −
  have D.cones-map (ψ (a′, a) (S.Fun ?f e)) ∈ D.cones a → D.cones a′
    using 2 D.cones-map-mapsto [of ψ (a′, a) (S.Fun ?f e)]
    by (elim C.in-homE, auto)
  thus cone J C D a′ (D.cones-map (ψ (a′, a) (S.Fun ?f e)) χ)
    using χ.cone-axioms by blast
qed
have A: ⋀h j. h ∈ C.hom a′ a ⟹ J.arr j ⟹
              S.Fun (YoD.at a′ (map o χ) j) (φ (a′, a) h)
                = φ (a′, D (J.cod j)) (χ j · h)
proof −
  fix h j
  assume j: J.arr j
  assume h: h ∈ C.hom a′ a
  have S.Fun (YoD.at a′ (map o χ) j) (φ (a′, a) h)
         = (φ (a′, D (J.cod j)) ∘ C (χ j) ∘ ψ (a′, a)) (φ (a′, a) h)
  proof −
    have S.Fun (YoD.at a′ (map o χ) j)
         = restrict (φ (a′, D (J.cod j)) ∘ C (χ j) ∘ ψ (a′, a))
                    (Hom.set (a′, a))
    proof −
      have S.Fun (YoD.at a′ (map o χ) j) = S.Fun (Y (χ j) a′)
        using a′ j YoD.at-simp Y-def Yoχ.preserves-reflects-arr [of j]
        by simp
      also have ... = restrict (φ (a′, D (J.cod j)) ∘ C (χ j) ∘ ψ (a′, a))
                               (Hom.set (a′, a))
        using a′ j χ.preserves-hom [of j J.dom j J.cod j]
              Y-arr-ide [of a′ χ j a D (J.cod j)] χ.A.map-simp S.Fun-mkArr
        by fastforce
      finally show ?thesis by blast
    qed
    thus ?thesis
      using a a′ h Hom.φ-mapsto by auto
  qed
  also have ... = φ (a′, D (J.cod j)) (χ j · h)
    using a a′ h Hom.ψ-φ by simp
  finally show S.Fun (YoD.at a′ (map o χ) j) (φ (a′, a) h)
                 = φ (a′, D (J.cod j)) (χ j · h)
    by auto
qed
have D.cones-map (ψ (a′, a) (S.Fun f′ e)) χ =
     D.cones-map (ψ (a′, a) (S.Fun ?f e)) χ
proof
  fix j
  have ¬J.arr j ⟹ D.cones-map (ψ (a′, a) (S.Fun f′ e)) χ j =
                   D.cones-map (ψ (a′, a) (S.Fun ?f e)) χ j
    using 1 2 χ.cone-axioms by (elim C.in-homE, auto)
  moreover have J.arr j ⟹ D.cones-map (ψ (a′, a) (S.Fun f′ e)) χ j =
                           D.cones-map (ψ (a′, a) (S.Fun ?f e)) χ j
```

```
proof −
  assume j: J.arr j
  have D.cones-map (ψ (a', a) (S.Fun f' e)) χ j =
         χ j · ψ (a', a) (S.Fun f' e)
    using j 1 χ.cone-axioms by auto
  also have ... = ψ (a', D (J.cod j)) (S.Fun (σ j) e)
  proof −
    have ψ (a', D (J.cod j)) (S.Fun (YoD.at a' (map o χ) j) (S.Fun f' e)) =
          ψ (a', D (J.cod j))
            (φ (a', D (J.cod j)) (χ j · ψ (a', a) (S.Fun f' e)))
      using j a a' f'e A Hom.φ-ψ Hom.ψ-mapsto by force
    moreover have χ j · ψ (a', a) (S.Fun f' e) ∈ C.hom a' (D (J.cod j))
      using a a' j f'e Hom.ψ-mapsto χ.preserves-hom [of j J.dom j J.cod j]
          χ.A.map-simp
      by auto
    moreover have S.Fun (YoD.at a' (map o χ) j) (S.Fun f' e) =
                  S.Fun (σ j) e
      using Fun-map-a-a' a a' j f' e x Yoχ-a'.A.map-simp eval-at-ide
          Yoχ-a'.cone-axioms
      by auto
    ultimately show ?thesis
      using a a' Hom.ψ-φ by auto
  qed
  also have ... = χ j · ψ (a', a) (S.Fun ?f e)
  proof −
    have S.Fun (YoD.at a' (map o χ) j) (S.Fun ?f e) =
          φ (a', D (J.cod j)) (χ j · ψ (a', a) (S.Fun ?f e))
      using j a a' fe A [of ψ (a', a) (S.Fun ?f e) j] Hom.φ-ψ Hom.ψ-mapsto
      by auto
    hence ψ (a', D (J.cod j)) (S.Fun (YoD.at a' (map o χ) j) (S.Fun ?f e)) =
          ψ (a', D (J.cod j))
            (φ (a', D (J.cod j)) (χ j · ψ (a', a) (S.Fun ?f e)))
      by simp
    moreover have χ j · ψ (a', a) (S.Fun ?f e) ∈ C.hom a' (D (J.cod j))
      using a a' j fe Hom.ψ-mapsto χ.preserves-hom [of j J.dom j J.cod j]
          χ.A.map-simp
      by auto
    moreover have S.Fun (YoD.at a' (map o χ) j) (S.Fun ?f e) =
                  S.Fun (σ j) e
    proof −
      have S.Fun (YoD.at a' (map o χ) j) (S.Fun ?f e)
            = (S.Fun (YoD.at a' (map o χ) j) o S.Fun ?f) e
        by simp
      also have ... = S.Fun (YoD.at a' (map o χ) j ·S ?f) e
        using Fun-map-a-a' a a' j f e x Yoχ-a'.A.map-simp eval-at-ide
        by auto
      also have ... = S.Fun (σ j) e
      proof −
        have YoD.at a' (map o χ) j ·S ?f =
```

```
                    YoD-a'.cones-map ?f (YoD.at a' (map o χ)) j
                  using j f Yoχ-a'.cone-axioms Fun-map-a-a' by auto
                thus ?thesis using j ff by argo
              qed
              finally show ?thesis by auto
            qed
            ultimately show ?thesis
              using a a' Hom.ψ-φ by auto
          qed
          also have ... = D.cones-map (ψ (a', a) (S.Fun ?f e)) χ j
            using j 2 χ.cone-axioms by force
          finally show D.cones-map (ψ (a', a) (S.Fun f' e)) χ j =
                       D.cones-map (ψ (a', a) (S.Fun ?f e)) χ j
            by auto
        qed
        ultimately show D.cones-map (ψ (a', a) (S.Fun f' e)) χ j =
                        D.cones-map (ψ (a', a) (S.Fun ?f e)) χ j
          by auto
      qed
      hence ψ (a', a) (S.Fun f' e) = ψ (a', a) (S.Fun ?f e)
        using 1 2 χofe.cone-axioms χ.cone-axioms χ.is-universal by blast
      hence φ (a', a) (ψ (a', a) (S.Fun f' e)) = φ (a', a) (ψ (a', a) (S.Fun ?f e))
        by simp
      thus S.Fun f' e = S.Fun ?f e
        using a a' fe f'e Hom.φ-ψ by force
    qed
    ultimately show S.Fun f' e = S.Fun ?f e by auto
  qed
 qed
 ultimately have ∃!f. «f : x →S Hom.map (a', a)» ∧
                      YoD-a'.cones-map f (YoD.at a' (map o χ)) = σ
  using ex1I [of λf. S.in-hom x (Hom.map (a', a)) f ∧
                      YoD-a'.cones-map f (YoD.at a' (map o χ)) = σ]
  by blast
 thus ∃!f. «f : x →S Cop-S.Map (map a) a'» ∧
           YoD-a'.cones-map f (YoD.at a' (map o χ)) = σ
  using a a' Y-def by simp
qed
qed
thus YoD.has-as-limit (map a)
 using YoD.cone-is-limit-if-pointwise-limit Yoχ.cone-axioms by auto
qed

end

end
```

Chapter 21

Category with Pullbacks

theory *CategoryWithPullbacks*
imports *Limit*
begin

In this chapter, we give a traditional definition of pullbacks in a category as limits of cospan diagrams and we define a locale *category-with-pullbacks* that is satisfied by categories in which every cospan diagram has a limit. These definitions build on the general definition of limit that we gave in *Category3.Limit*. We then define a locale *elementary-category-with-pullbacks* that axiomatizes categories equipped with chosen functions that assign to each cospan a corresponding span of "projections", which enjoy the familiar universal property of a pullback. After developing consequences of the axioms, we prove that the two locales are in agreement, in the sense that every interpretation of *category-with-pullbacks* extends to an interpretation of *elementary-category-with-pullbacks*, and conversely, the underlying category of an interpretation of *elementary-category-with-pullbacks* always yields an interpretation of *category-with-pullbacks*.

21.1 Commutative Squares

context *category*
begin

The following provides some useful technology for working with commutative squares.

definition *commutative-square*
where *commutative-square f g h k* ≡ *cospan f g* ∧ *span h k* ∧ *dom f* = *cod h* ∧ *f* ⋅ *h* = *g* ⋅ *k*

lemma *commutative-squareI* [*intro*, *simp*]:
assumes *cospan f g* **and** *span h k* **and** *dom f* = *cod h* **and** *f* ⋅ *h* = *g* ⋅ *k*
shows *commutative-square f g h k*
using *assms commutative-square-def* **by** *auto*

lemma *commutative-squareE* [*elim*]:
assumes *commutative-square f g h k*

and ⟦ *arr f*; *arr g*; *arr h*; *arr k*; *cod f* = *cod g*; *dom h* = *dom k*; *dom f* = *cod h*;
dom g = *cod k*; *f* · *h* = *g* · *k* ⟧ ⟹ *T*
shows *T*
using *assms commutative-square-def*
by (*metis* (*mono-tags*, *lifting*) *seqE seqI*)

lemma *commutative-square-comp-arr*:
assumes *commutative-square f g h k* **and** *seq h l*
shows *commutative-square f g* (*h* · *l*) (*k* · *l*)
using *assms*
apply (*elim commutative-squareE*, *intro commutative-squareI*, *auto*)
using *comp-assoc* **by** *metis*

lemma *arr-comp-commutative-square*:
assumes *commutative-square f g h k* **and** *seq l f*
shows *commutative-square* (*l* · *f*) (*l* · *g*) *h k*
using *assms comp-assoc*
by (*elim commutative-squareE*, *intro commutative-squareI*, *auto*)

end

21.2 Cospan Diagrams

The "shape" of a cospan diagram is a category having two non-identity arrows with distinct domains and a common codomain.

locale *cospan-shape*
begin

datatype *Arr* = *Null* | *AA* | *BB* | *TT* | *AT* | *BT*

fun *comp*
where *comp AA AA* = *AA*
| *comp AT AA* = *AT*
| *comp TT AT* = *AT*
| *comp BB BB* = *BB*
| *comp BT BB* = *BT*
| *comp TT BT* = *BT*
| *comp TT TT* = *TT*
| *comp - -* = *Null*

interpretation *partial-composition comp*
proof
show ∃!*n*. ∀*f*. *comp n f* = *n* ∧ *comp f n* = *n*
proof
show ∀*f*. *comp Null f* = *Null* ∧ *comp f Null* = *Null* **by** *simp*
show ⋀*n*. ∀*f*. *comp n f* = *n* ∧ *comp f n* = *n* ⟹ *n* = *Null*
by (*metis comp.simps*(*8*))
qed

qed

lemma *null-char*:
shows $null = Null$
proof −
  have $\forall f.\ comp\ Null\ f = Null \wedge comp\ f\ Null = Null$ **by** *simp*
  thus *?thesis*
    using *null-def ex-un-null theI* [*of* $\lambda n.\ \forall f.\ comp\ n\ f = n \wedge comp\ f\ n = n$]
    by (*metis partial-magma.null-is-zero*(*2*) *partial-magma-axioms*)
qed

lemma *ide-char*:
shows $ide\ f \longleftrightarrow f = AA \vee f = BB \vee f = TT$
proof
  show $ide\ f \Longrightarrow f = AA \vee f = BB \vee f = TT$
    using *ide-def null-char* **by** (*cases f*, *simp-all*)
  show $f = AA \vee f = BB \vee f = TT \Longrightarrow ide\ f$
  proof −
    have *1*: $\bigwedge f\ g.\ f = AA \vee f = BB \vee f = TT \Longrightarrow$
                  $comp\ f\ f \neq Null \wedge$
                  $(comp\ g\ f \neq Null \longrightarrow comp\ g\ f = g) \wedge$
                  $(comp\ f\ g \neq Null \longrightarrow comp\ f\ g = g)$
    proof −
      fix *f g*
      show $f = AA \vee f = BB \vee f = TT \Longrightarrow$
            $comp\ f\ f \neq Null \wedge$
            $(comp\ g\ f \neq Null \longrightarrow comp\ g\ f = g) \wedge$
            $(comp\ f\ g \neq Null \longrightarrow comp\ f\ g = g)$
        by (*cases f*; *cases g*, *auto*)
    qed
    assume *f*: $f = AA \vee f = BB \vee f = TT$
    show $ide\ f$
      using *f 1 ide-def null-char* **by** *simp*
  qed
qed

fun *Dom*
where $Dom\ AA = AA$
  | $Dom\ BB = BB$
  | $Dom\ TT = TT$
  | $Dom\ AT = AA$
  | $Dom\ BT = BB$
  | $Dom\ \text{-} = Null$

fun *Cod*
where $Cod\ AA = AA$
  | $Cod\ BB = BB$
  | $Cod\ TT = TT$
  | $Cod\ AT = TT$

```
  | Cod BT = TT
  | Cod - = Null

lemma domains-char':
shows domains f = (if f = Null then {} else {Dom f})
  using domains-def ide-char null-char
  by (cases f, auto)

lemma codomains-char':
shows codomains f = (if f = Null then {} else {Cod f})
  using codomains-def ide-char null-char
  by (cases f, auto)

lemma arr-char:
shows arr f ⟷ f ≠ Null
  using arr-def domains-char' codomains-char' by simp

lemma seq-char:
shows seq g f ⟷ (f = AA ∧ (g = AA ∨ g = AT)) ∨
                (f = BB ∧ (g = BB ∨ g = BT)) ∨
                (f = AT ∧ g = TT) ∨
                (f = BT ∧ g = TT) ∨
                (f = TT ∧ g = TT)
  using arr-char null-char
  by (cases f; cases g, simp-all)

interpretation category comp
proof
  fix f g h
  show comp g f ≠ null ⟹ seq g f
    using null-char arr-char seq-char by simp
  show domains f ≠ {} ⟷ codomains f ≠ {}
    using domains-char' codomains-char' by auto
  show seq h g ⟹ seq (comp h g) f ⟹ seq g f
    using seq-char arr-char
    by (cases g; cases h; simp-all)
  show seq h (comp g f) ⟹ seq g f ⟹ seq h g
    using seq-char arr-char
    by (cases f; cases g; simp-all)
  show seq g f ⟹ seq h g ⟹ seq (comp h g) f
    using seq-char arr-char
    by (cases f; simp-all; cases g; simp-all; cases h; auto)
  show seq g f ⟹ seq h g ⟹ comp (comp h g) f = comp h (comp g f)
   using seq-char
   by (cases f; simp-all; cases g; simp-all; cases h; auto)
qed

lemma is-category:
shows category comp
```

```
  ..

  lemma dom-char:
  shows dom = Dom
    using dom-def domains-char domains-char' null-char by fastforce

  lemma cod-char:
  shows cod = Cod
    using cod-def codomains-char codomains-char' null-char by fastforce

end

sublocale cospan-shape ⊆ category comp
  using is-category by auto

locale cospan-diagram =
  J: cospan-shape +
  C: category C
for C :: 'c comp      (infixr ‹·› 55)
and f0 :: 'c
and f1 :: 'c +
assumes is-cospan: C.cospan f0 f1
begin

  no-notation J.comp   (infixr ‹·› 55)
  notation J.comp      (infixr ‹·J› 55)

  fun map
  where map J.AA = C.dom f0
      | map J.BB = C.dom f1
      | map J.TT = C.cod f0
      | map J.AT = f0
      | map J.BT = f1
      | map - = C.null

end

sublocale cospan-diagram ⊆ diagram J.comp C map
proof
  show ⋀f. ¬ J.arr f ⟹ map f = C.null
    using J.arr-char by simp
  fix f
  assume f: J.arr f
  show C.arr (map f)
    using f J.arr-char is-cospan by (cases f, simp-all)
  show C.dom (map f) = map (J.dom f)
    using f J.arr-char J.dom-char is-cospan by (cases f, simp-all)
  show C.cod (map f) = map (J.cod f)
    using f J.arr-char J.cod-char is-cospan by (cases f, simp-all)
```

next
fix *f g*
assume *fg*: *J.seq g f*
show *map* (*g* ·$_J$ *f*) = *map g* · *map f*
using *fg J.seq-char J.null-char J.not-arr-null is-cospan*
apply (*cases f*; *cases g*, *simp-all*)
using *C.comp-arr-dom C.comp-cod-arr* **by** *auto*
qed

21.3 Category with Pullbacks

A *pullback* in a category C is a limit of a cospan diagram in C.

context *cospan-diagram*
begin

definition *mkCone*
where *mkCone p0 p1* ≡ λ*j*. *if j* = *J.AA then p0*
else if j = *J.BB then p1*
else if j = *J.AT then f0* · *p0*
else if j = *J.BT then f1* · *p1*
else if j = *J.TT then f0* · *p0*
else C.null

abbreviation *is-rendered-commutative-by*
where *is-rendered-commutative-by p0 p1* ≡ *C.seq f0 p0* ∧ *f0* · *p0* = *f1* · *p1*

abbreviation *has-as-pullback*
where *has-as-pullback p0 p1* ≡ *limit-cone* (*C.dom p0*) (*mkCone p0 p1*)

lemma *cone-mkCone*:
assumes *is-rendered-commutative-by p0 p1*
shows *cone* (*C.dom p0*) (*mkCone p0 p1*)
proof −
interpret *E*: *constant-functor J.comp C* ‹*C.dom p0*›
apply *unfold-locales* **using** *assms* **by** *auto*
show *cone* (*C.dom p0*) (*mkCone p0 p1*)
proof
fix *f*
show ¬ *J.arr f* ⟹ *mkCone p0 p1 f* = *C.null*
using *mkCone-def J.arr-char* **by** *simp*
assume *f*: *J.arr f*
show *C.arr* (*mkCone p0 p1 f*)
using *assms f mkCone-def J.arr-char*
by (*cases f*, *simp-all*) *blast*+
show *map f* · *mkCone p0 p1* (*J.dom f*) = *mkCone p0 p1 f*
using *assms f mkCone-def J.arr-char J.dom-char C.comp-ide-arr is-cospan*
by (*cases f*, *auto*)
show *mkCone p0 p1* (*J.cod f*) · *E.map f* = *mkCone p0 p1 f*

```
    using assms f mkCone-def J.arr-char J.cod-char C.comp-arr-dom
    apply (cases f, auto)
      apply (metis C.dom-comp C.seqE)
    by (metis C.dom-comp)+
  qed
qed

lemma is-rendered-commutative-by-cone:
assumes cone a χ
shows is-rendered-commutative-by (χ J.AA) (χ J.BB)
proof −
  interpret χ: cone J.comp C map a χ
    using assms by auto
  show ?thesis
  proof
    show C.seq f0 (χ J.AA)
      by (metis C.seqI J.cod-char J.seq-char χ.preserves-cod χ.preserves-reflects-arr
          J.seqE is-cospan J.Cod.simps(1) map.simps(1))
    show f0 · χ J.AA = f1 · χ J.BB
      by (metis J.cod-char J.dom-char χ.A.map-simp χ.naturality
      J.Cod.simps(4−5) J.Dom.simps(4−5) J.comp.simps(2,5) J.seq-char map.simps(4−5))
  qed
qed

lemma mkCone-cone:
assumes cone a χ
shows mkCone (χ J.AA) (χ J.BB) = χ
proof −
  interpret χ: cone J.comp C map a χ
    using assms by auto
  have 1: is-rendered-commutative-by (χ J.AA) (χ J.BB)
    using assms is-rendered-commutative-by-cone by blast
 interpret mkCone-χ: cone J.comp C map ‹C.dom (χ J.AA)› ‹mkCone (χ J.AA) (χ J.BB)›
    using assms cone-mkCone 1 by auto
  show ?thesis
  proof −
    have ⋀j. j = J.AA ⟹ mkCone (χ J.AA) (χ J.BB) j = χ j
      using mkCone-def χ.extensionality by simp
    moreover have ⋀j. j = J.BB ⟹ mkCone (χ J.AA) (χ J.BB) j = χ j
      using mkCone-def χ.extensionality by simp
    moreover have ⋀j. j = J.TT ⟹ mkCone (χ J.AA) (χ J.BB) j = χ j
      using 1 mkCone-def χ.extensionality χ.A.map-simp χ.preserves-comp-1
            cospan-shape.seq-char χ.naturality2
      apply simp
      by (metis J.seqE J.comp.simps(5) map.simps(5))
    ultimately have ⋀j. J.ide j ⟹ mkCone (χ J.AA) (χ J.BB) j = χ j
      using J.ide-char by auto
    thus mkCone (χ J.AA) (χ J.BB) = χ
      using mkCone-def natural-transformation-eqI [of J.comp C]
```

```
        χ.natural-transformation-axioms mkCone-χ.natural-transformation-axioms
        J.ide-char
    by simp
  qed
qed

lemma cone-iff-commutative-square:
shows cone (C.dom h) (mkCone h k) ⟷ C.commutative-square f0 f1 h k
  using cone-mkCone mkCone-def J.arr-char J.ide-char is-rendered-commutative-by-cone
      is-cospan C.commutative-square-def cospan-shape.Arr.simps(11)
      C.dom-comp C.seqE C.seqI
  apply (intro iffI)
  by (intro C.commutative-squareI) metis+

lemma cones-map-mkCone-eq-iff:
assumes is-rendered-commutative-by p0 p1 and is-rendered-commutative-by p0' p1'
and «h : C.dom p0' → C.dom p0»
shows cones-map h (mkCone p0 p1) = mkCone p0' p1' ⟷ p0 · h = p0' ∧ p1 · h = p1'
proof −
  interpret χ: cone J.comp C map ‹C.dom p0› ‹mkCone p0 p1›
    using assms(1) cone-mkCone [of p0 p1] by blast
  interpret χ': cone J.comp C map ‹C.dom p0'› ‹mkCone p0' p1'›
    using assms(2) cone-mkCone [of p0' p1'] by blast
  show ?thesis
  proof
    assume 3: cones-map h (mkCone p0 p1) = mkCone p0' p1'
    show p0 · h = p0' ∧ p1 · h = p1'
    proof
      show p0 · h = p0'
      proof −
        have p0' = mkCone p0' p1' J.AA
          using mkCone-def J.arr-char by simp
        also have ... = cones-map h (mkCone p0 p1) J.AA
          using 3 by simp
        also have ... = p0 · h
          using assms mkCone-def J.arr-char χ.cone-axioms by auto
        finally show ?thesis by auto
      qed
      show p1 · h = p1'
      proof −
        have p1' = mkCone p0' p1' J.BB
          using mkCone-def J.arr-char by simp
        also have ... = cones-map h (mkCone p0 p1) J.BB
          using 3 by simp
        also have ... = p1 · h
          using assms mkCone-def J.arr-char χ.cone-axioms by auto
        finally show ?thesis by auto
      qed
    qed
```

```
    next
    assume 4: p0 · h = p0' ∧ p1 · h = p1'
    show cones-map h (mkCone p0 p1) = mkCone p0' p1'
    proof
      fix j
      have ¬ J.arr j ⟹ cones-map h (mkCone p0 p1) j = mkCone p0' p1' j
        using assms χ.cone-axioms mkCone-def J.arr-char by auto
      moreover have J.arr j ⟹ cones-map h (mkCone p0 p1) j = mkCone p0' p1' j
        using assms 4 χ.cone-axioms mkCone-def J.arr-char C.comp-assoc
        by fastforce
      ultimately show cones-map h (mkCone p0 p1) j = mkCone p0' p1' j
        using J.arr-char J.Dom.cases by blast
    qed
  qed
qed

end

locale pullback-cone =
  J: cospan-shape +
  C: category C +
  D: cospan-diagram C f0 f1 +
  limit-cone J.comp C D.map ‹C.dom p0› ‹D.mkCone p0 p1›
for C :: 'c comp      (infixr ‹·› 55)
and f0 :: 'c
and f1 :: 'c
and p0 :: 'c
and p1 :: 'c
begin


  lemma renders-commutative:
  shows D.is-rendered-commutative-by p0 p1
    using D.mkCone-def D.cospan-diagram-axioms cone-axioms
          cospan-diagram.is-rendered-commutative-by-cone
    by fastforce

  lemma is-universal':
  assumes D.is-rendered-commutative-by p0' p1'
  shows ∃!h. «h : C.dom p0' → C.dom p0» ∧ p0 · h = p0' ∧ p1 · h = p1'
  proof −
    have D.cone (C.dom p0') (D.mkCone p0' p1')
      using assms D.cone-mkCone by blast
    hence 1: ∃!h. «h : C.dom p0' → C.dom p0» ∧
                  D.cones-map h (D.mkCone p0 p1) = D.mkCone p0' p1'
      using is-universal by simp
    have 2: ⋀h. «h : C.dom p0' → C.dom p0» ⟹
                D.cones-map h (D.mkCone p0 p1) = D.mkCone p0' p1' ⟷
                p0 · h = p0' ∧ p1 · h = p1'
```

```
    using assms D.cones-map-mkCone-eq-iff [of p0 p1 p0' p1'] renders-commutative
    by fastforce
  thus ?thesis using 1 by blast
qed

lemma induced-arrowI':
assumes D.is-rendered-commutative-by p0' p1'
shows «induced-arrow (C.dom p0') (D.mkCone p0' p1') : C.dom p0' → C.dom p0»
and p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') = p0'
and p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') = p1'
proof −
  interpret A': constant-functor J.comp C ‹C.dom p0'›
    using assms by (unfold-locales, auto)
  have cone: D.cone (C.dom p0') (D.mkCone p0' p1')
    using assms D.cone-mkCone [of p0' p1'] by blast
  show 1: p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') = p0'
  proof −
    have p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') =
          D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1'))
                      (D.mkCone p0 p1) J.AA
      using cone induced-arrowI(1) D.mkCone-def J.arr-char is-cone by force
    also have ... = p0'
    proof −
      have D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1'))
                       (D.mkCone p0 p1) =
           D.mkCone p0' p1'
        using cone induced-arrowI by blast
      thus ?thesis
        using J.arr-char D.mkCone-def by simp
    qed
    finally show ?thesis by auto
  qed
  show 2: p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') = p1'
  proof −
    have p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') =
         D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1'))
                     (D.mkCone p0 p1) J.BB
    proof −
      have C.dom p0' = C.dom p1'
        using assms by (metis C.dom-comp)
      thus ?thesis
        using cone induced-arrowI(1) D.mkCone-def J.arr-char is-cone by force
    qed
    also have ... = p1'
    proof −
      have D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1'))
                       (D.mkCone p0 p1) =
           D.mkCone p0' p1'
        using cone induced-arrowI by blast
```

```
      thus ?thesis
        using J.arr-char D.mkCone-def by simp
    qed
    finally show ?thesis by auto
  qed
  show «induced-arrow (C.dom p0′) (D.mkCone p0′ p1′) : C.dom p0′ → C.dom p0»
    using 1 cone induced-arrowI by simp
qed

end

context category
begin

  definition has-as-pullback
  where has-as-pullback f0 f1 p0 p1 ≡
        cospan f0 f1 ∧ cospan-diagram.has-as-pullback C f0 f1 p0 p1

  definition has-pullbacks
  where has-pullbacks = (∀ f0 f1. cospan f0 f1 ⟶ (∃ p0 p1. has-as-pullback f0 f1 p0 p1))

  lemma has-as-pullbackI [intro]:
  assumes cospan f g and commutative-square f g p q
  and ⋀h k. commutative-square f g h k ⟹ ∃!l. p · l = h ∧ q · l = k
  shows has-as-pullback f g p q
  proof (unfold has-as-pullback-def, intro conjI)
    show arr f and arr g and cod f = cod g
      using assms(1) by auto
    interpret J: cospan-shape .
    interpret D: cospan-diagram C f g
      using assms(1-2) by unfold-locales auto
    show D.has-as-pullback p q
    proof -
      have 1: D.is-rendered-commutative-by p q
        using assms ide-in-hom by blast
      let ?χ = D.mkCone p q
      let ?a = dom p
      interpret χ: cone J.comp C D.map ?a ?χ
        using assms(2) D.cone-mkCone 1 by auto
      interpret χ: limit-cone J.comp C D.map ?a ?χ
      proof
        fix x χ′
        assume χ′: D.cone x χ′
        interpret χ′: cone J.comp C D.map x χ′
          using χ′ by simp
        have 2: D.is-rendered-commutative-by (χ′ J.AA) (χ′ J.BB)
          using χ′ D.is-rendered-commutative-by-cone [of x χ′] by blast
        have 3: ∃!l. p · l = χ′ J.AA ∧ q · l = χ′ J.BB
          using assms(1,3) 2 χ′.preserves-hom J.arr-char J.ide-char by simp
```

```
      obtain l where l: p · l = χ' J.AA ∧ q · l = χ' J.BB
        using 3 by blast
      have «l : x → ?a»
        using l 2 χ'.preserves-hom J.arr-char J.ide-char χ'.component-in-hom
              χ'.extensionality χ'.preserves-reflects-arr comp-in-homE null-is-zero(2) in-homE
        by metis
      moreover have D.cones-map l (D.mkCone p q) = χ'
        using l D.cones-map-mkCone-eq-iff [of p q χ' J.AA χ' J.BB l]
        by (metis (no-types, lifting) 1 2 D.mkCone-cone χ' calculation dom-comp in-homE
seqE)
      ultimately have ∃ l. «l : x → ?a» ∧ D.cones-map l (D.mkCone p q) = χ'
        by blast
      moreover have ⋀l'. ⟦«l' : x → ?a»; D.cones-map l' (D.mkCone p q) = χ'⟧ ⟹ l' = l
      proof −
        fix l'
        assume l': «l' : x → ?a»
        assume eq: D.cones-map l' (D.mkCone p q) = χ'
        have p · l' = χ' J.AA ∧ q · l' = χ' J.BB
          using l' eq J.arr-char χ.cone-axioms D.mkCone-def by auto
        thus l' = l
          using 3 l by blast
      qed
      ultimately show ∃!l. «l : x → ?a» ∧ D.cones-map l (D.mkCone p q) = χ'
        by blast
    qed
    show D.has-as-pullback p q
      using assms χ.limit-cone-axioms by blast
  qed
qed

lemma pullbacks-are-isomorphic:
assumes has-as-pullback f g h k and has-as-pullback f g h' k'
shows isomorphic (dom h) (dom h')
  using assms limits-are-isomorphic(1)
  unfolding has-as-pullback-def by blast

lemma has-as-pullbackE [elim]:
assumes has-as-pullback f g p q
and ⟦cospan f g; commutative-square f g p q;
     ⋀h k. commutative-square f g h k ⟹ ∃!l. p · l = h ∧ q · l = k⟧ ⟹ T
shows T
proof −
  interpret J: cospan-shape .
  interpret D: cospan-diagram C f g
    using assms(1) has-as-pullback-def
    by (meson category-axioms cospan-diagram.intro cospan-diagram-axioms.intro)
  have 1: ⋀h k. commutative-square f g h k ⟷ D.cone (dom h) (D.mkCone h k)
    using D.cone-iff-commutative-square by presburger
  let ?χ = D.mkCone p q
```

interpret χ: *limit-cone J.comp C D.map* ‹*dom p*› *?χ*
using *assms*(*1*) *has-as-pullback-def D.cone-mkCone* **by** *blast*
have *cospan f g*
using *D.is-cospan* **by** *blast*
moreover have *csq*: *commutative-square f g p q*
using *1* χ*.cone-axioms* **by** *blast*
moreover have $\bigwedge h\ k.$ *commutative-square f g h k* $\Longrightarrow \exists!l.\ p \cdot l = h \wedge q \cdot l = k$
proof −
fix *h k*
assume *2*: *commutative-square f g h k*
let *?*χ' = *D.mkCone h k*
interpret χ': *cone J.comp C D.map* ‹*dom h*› *?*χ'
using *1 2* **by** *blast*
have *3*: $\exists!l.$ «l : *dom h* → *dom p*» ∧ *D.cones-map l ?*χ = *?*χ'
using *1 2* χ*.is-universal* [*of dom h D.mkCone h k*] **by** *blast*
obtain *l* **where** *l*: «l : *dom h* → *dom p*» ∧ *D.cones-map l ?*χ = *?*χ'
using *3* **by** *blast*
have $p \cdot l = h \wedge q \cdot l = k$
proof
have $p \cdot l$ = *D.cones-map l ?*χ *J.AA*
using χ*.cone-axioms D.mkCone-def J.seq-char in-homE*
apply *simp*
by (*metis J.seqE l*)
also have ... = *h*
using *l* χ'*.cone-axioms D.mkCone-def J.seq-char in-homE* **by** *simp*
finally show $p \cdot l = h$ **by** *blast*
have $q \cdot l$ = *D.cones-map l ?*χ *J.BB*
using χ*.cone-axioms D.mkCone-def J.seq-char in-homE*
apply *simp*
by (*metis J.seqE l*)
also have ... = *k*
using *l* χ'*.cone-axioms D.mkCone-def J.seq-char in-homE* **by** *simp*
finally show $q \cdot l = k$ **by** *blast*
qed
moreover have $\bigwedge l'.\ p \cdot l' = h \wedge q \cdot l' = k \Longrightarrow l' = l$
proof −
fix l'
assume *1*: $p \cdot l' = h \wedge q \cdot l' = k$
have *2*: «l' : *dom h* → *dom p*»
using *1*
by (*metis* χ'*.ide-apex arr-dom-iff-arr arr-iff-in-hom ideD*(*1*) *seqE dom-comp*)
moreover have *D.cones-map l'* *?*χ = *?*χ'
using *D.cones-map-mkCone-eq-iff*
by (*meson 1 2 csq D.cone-iff-commutative-square* χ'*.cone-axioms*
commutative-squareE seqI)
ultimately show $l' = l$
using *l* χ*.is-universal* χ'*.cone-axioms* **by** *blast*
qed
ultimately show $\exists!l.\ p \cdot l = h \wedge q \cdot l = k$ **by** *blast*

```
    qed
    ultimately show T
      using assms(2) by blast
  qed

end

locale category-with-pullbacks =
  category +
assumes has-pullbacks: has-pullbacks
```

21.4 Elementary Category with Pullbacks

An *elementary category with pullbacks* is a category equipped with a specific way of mapping each cospan to a span such that the resulting square commutes and such that the span is universal for that property. It is useful to assume that the functions, mapping a cospan to the two projections of the pullback, are extensional; that is, they yield *null* when applied to arguments that do not form a cospan.

```
locale elementary-category-with-pullbacks =
  category C
for C :: 'a comp                                (infixr ‹⋅› 55)
and prj0 :: 'a ⇒ 'a ⇒ 'a                        (‹p₀[-, -]›)
and prj1 :: 'a ⇒ 'a ⇒ 'a                        (‹p₁[-, -]›) +
assumes prj0-ext: ¬ cospan f g ⟹ p₀[f, g] = null
and prj1-ext: ¬ cospan f g ⟹ p₁[f, g] = null
and pullback-commutes [intro]: cospan f g ⟹ commutative-square f g p₁[f, g] p₀[f, g]
and universal: commutative-square f g h k ⟹ ∃!l. p₁[f, g] ⋅ l = h ∧ p₀[f, g] ⋅ l = k
begin

  lemma pullback-commutes':
  assumes cospan f g
  shows f ⋅ p₁[f, g] = g ⋅ p₀[f, g]
    using assms commutative-square-def by blast

  lemma prj0-in-hom':
  assumes cospan f g
  shows «p₀[f, g] : dom p₀[f, g] → dom g»
    using assms pullback-commutes
    by (metis category.commutative-squareE category-axioms in-homI)

  lemma prj1-in-hom':
  assumes cospan f g
  shows «p₁[f, g] : dom p₀[f, g] → dom f»
    using assms pullback-commutes
    by (metis category.commutative-squareE category-axioms in-homI)
```

The following gives us a notation for the common domain of the two projections of a pullback.

definition *pbdom* (**infix** ‹↓↓› *51*)
where $f \downarrow\downarrow g \equiv dom\ \mathrm{p}_0[f, g]$

lemma *pbdom-in-hom* [*intro*]:
assumes *cospan f g*
shows «$f \downarrow\downarrow g : f \downarrow\downarrow g \to f \downarrow\downarrow g$»
  unfolding *pbdom-def*
  using *assms prj0-in-hom′*
  by (*metis arr-dom-iff-arr arr-iff-in-hom cod-dom dom-dom in-homE*)

lemma *ide-pbdom* [*simp*]:
assumes *cospan f g*
shows *ide* ($f \downarrow\downarrow g$)
  using *assms ide-in-hom* **by** *auto*[*1*]

lemma *prj0-in-hom* [*intro*, *simp*]:
assumes *cospan f g* **and** $a = f \downarrow\downarrow g$ **and** $b = dom\ g$
shows «$\mathrm{p}_0[f, g] : a \to b$»
  unfolding *pbdom-def*
  using *assms prj0-in-hom′* **by** (*simp add*: *pbdom-def*)

lemma *prj1-in-hom* [*intro*, *simp*]:
assumes *cospan f g* **and** $a = f \downarrow\downarrow g$ **and** $b = dom\ f$
shows «$\mathrm{p}_1[f, g] : a \to b$»
  unfolding *pbdom-def*
  using *assms prj1-in-hom′* **by** (*simp add*: *pbdom-def*)

lemma *prj0-simps* [*simp*]:
assumes *cospan f g*
shows *arr* $\mathrm{p}_0[f, g]$ **and** *dom* $\mathrm{p}_0[f, g] = f \downarrow\downarrow g$ **and** *cod* $\mathrm{p}_0[f, g] = dom\ g$
  using *assms prj0-in-hom* **by** (*blast*, *blast*, *blast*)

lemma *prj0-simps-arr* [*iff*]:
shows *arr* $\mathrm{p}_0[f, g] \longleftrightarrow$ *cospan f g*
proof
  show *cospan f g* $\Longrightarrow$ *arr* $\mathrm{p}_0[f, g]$
    using *prj0-in-hom* **by** *auto*
  show *arr* $\mathrm{p}_0[f, g] \Longrightarrow$ *cospan f g*
    using *prj0-ext not-arr-null* **by** *metis*
qed

lemma *prj1-simps* [*simp*]:
assumes *cospan f g*
shows *arr* $\mathrm{p}_1[f, g]$ **and** *dom* $\mathrm{p}_1[f, g] = f \downarrow\downarrow g$ **and** *cod* $\mathrm{p}_1[f, g] = dom\ f$
  using *assms prj1-in-hom* **by** (*blast*, *blast*, *blast*)

lemma *prj1-simps-arr* [*iff*]:
shows *arr* $\mathrm{p}_1[f, g] \longleftrightarrow$ *cospan f g*
proof

```
  show cospan f g ⟹ arr p₁[f, g]
    using prj1-in-hom by auto
  show arr p₁[f, g] ⟹ cospan f g
    using prj1-ext not-arr-null by metis
qed

lemma span-prj:
assumes cospan f g
shows span p₀[f, g] p₁[f, g]
  using assms by simp
```

We introduce a notation for tupling, which produces the induced arrow into a pullback. In our notation, the "0-side", which we regard as the input, occurs on the right, and the "1-side", which we regard as the output, occurs on the left.

```
definition tuple          (‹⟨- ⟦-, -⟧ -⟩›)
where ⟨h ⟦f, g⟧ k⟩ ≡ if commutative-square f g h k then
                       THE l. p₀[f, g] · l = k ∧ p₁[f, g] · l = h
                     else null

lemma tuple-in-hom [intro]:
assumes commutative-square f g h k
shows «⟨h ⟦f, g⟧ k⟩ : dom h → f ↓↓ g»
proof
  have 1: p₀[f, g] · ⟨h ⟦f, g⟧ k⟩ = k ∧ p₁[f, g] · ⟨h ⟦f, g⟧ k⟩ = h
    unfolding tuple-def
    using assms universal theI [of λl. p₀[f, g] · l = k ∧ p₁[f, g] · l = h]
    apply simp
    by meson
  show arr ⟨h ⟦f, g⟧ k⟩
    using assms 1
    apply (elim commutative-squareE)
    by (metis (no-types, lifting) seqE)
  show dom ⟨h ⟦f, g⟧ k⟩ = dom h
    using assms 1
    apply (elim commutative-squareE)
    by (metis (no-types, lifting) dom-comp)
  show cod ⟨h ⟦f, g⟧ k⟩ = f ↓↓ g
    unfolding pbdom-def
    using assms 1
    apply (elim commutative-squareE)
    by (metis seqE)
qed

lemma tuple-extensionality:
assumes ¬ commutative-square f g h k
shows ⟨h ⟦f, g⟧ k⟩ = null
  unfolding tuple-def
  using assms by simp
```

lemma *tuple-simps* [*simp*]:
assumes *commutative-square f g h k*
shows *arr* ⟨*h* ⟦*f*, *g*⟧ *k*⟩ **and** *dom* ⟨*h* ⟦*f*, *g*⟧ *k*⟩ = *dom h* **and** *cod* ⟨*h* ⟦*f*, *g*⟧ *k*⟩ = *f* ↓↓ *g*
using *assms tuple-in-hom* **apply** *blast*
using *assms tuple-in-hom* **apply** *blast*
using *assms tuple-in-hom* **by** *blast*

lemma *prj-tuple* [*simp*]:
assumes *commutative-square f g h k*
shows $\mathrm{p}_0[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = k$ **and** $\mathrm{p}_1[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = h$
proof −
have *1*: $\mathrm{p}_0[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = k \wedge \mathrm{p}_1[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = h$
unfolding *tuple-def*
using *assms universal theI* [*of* λ*l*. $\mathrm{p}_0[f, g] \cdot l = k \wedge \mathrm{p}_1[f, g] \cdot l = h$]
apply *simp*
by *meson*
show $\mathrm{p}_0[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = k$ **using** *1* **by** *simp*
show $\mathrm{p}_1[f, g] \cdot \langle h \llbracket f, g \rrbracket k \rangle = h$ **using** *1* **by** *simp*
qed

lemma *tuple-prj*:
assumes *cospan f g* **and** *seq* $\mathrm{p}_1[f, g]$ *h*
shows $\langle \mathrm{p}_1[f, g] \cdot h \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \cdot h \rangle = h$
proof −
have *1*: *commutative-square f g* ($\mathrm{p}_1[f, g] \cdot h$) ($\mathrm{p}_0[f, g] \cdot h$)
using *assms pullback-commutes*
by (*simp add*: *commutative-square-comp-arr*)
have $\mathrm{p}_0[f, g] \cdot \langle \mathrm{p}_1[f, g] \cdot h \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \cdot h \rangle = \mathrm{p}_0[f, g] \cdot h$
using *assms 1* **by** *simp*
moreover have $\mathrm{p}_1[f, g] \cdot \langle \mathrm{p}_1[f, g] \cdot h \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \cdot h \rangle = \mathrm{p}_1[f, g] \cdot h$
using *assms 1* **by** *simp*
ultimately show *?thesis*
unfolding *tuple-def*
using *assms 1 universal* [*of f g* $\mathrm{p}_1[f, g] \cdot h$ $\mathrm{p}_0[f, g] \cdot h$]
theI-unique [*of* λ*l*. $\mathrm{p}_0[f, g] \cdot l = \mathrm{p}_0[f, g] \cdot h \wedge \mathrm{p}_1[f, g] \cdot l = \mathrm{p}_1[f, g] \cdot h$ *h*]
by *auto*
qed

lemma *tuple-prj-spc* [*simp*]:
assumes *cospan f g*
shows $\langle \mathrm{p}_1[f, g] \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \rangle = f \downarrow\downarrow g$
proof −
have $\langle \mathrm{p}_1[f, g] \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \rangle = \langle \mathrm{p}_1[f, g] \cdot (f \downarrow\downarrow g) \llbracket f, g \rrbracket \mathrm{p}_0[f, g] \cdot (f \downarrow\downarrow g) \rangle$
using *assms comp-arr-dom* **by** *simp*
thus *?thesis*
using *assms tuple-prj* **by** *simp*
qed

lemma *prj-joint-monic*:

assumes *cospan f g* **and** *seq* $\mathrm{p}_1[f,\ g]\ h$ **and** *seq* $\mathrm{p}_1[f,\ g]\ h'$
and $\mathrm{p}_0[f,\ g] \cdot h = \mathrm{p}_0[f,\ g] \cdot h'$ **and** $\mathrm{p}_1[f,\ g] \cdot h = \mathrm{p}_1[f,\ g] \cdot h'$
shows $h = h'$
proof −
  have $h = \langle \mathrm{p}_1[f,\ g] \cdot h \ [\![f,\ g]\!]\ \mathrm{p}_0[f,\ g] \cdot h \rangle$
    using *assms tuple-prj* [*of f g h*] **by** *simp*
  also have ... $= \langle \mathrm{p}_1[f,\ g] \cdot h' \ [\![f,\ g]\!]\ \mathrm{p}_0[f,\ g] \cdot h' \rangle$
    using *assms* **by** *simp*
  also have ... $= h'$
    using *assms tuple-prj* [*of f g h'*] **by** *simp*
  finally show *?thesis* **by** *blast*
qed

The pullback of an identity along an arbitrary arrow is an isomorphism.

lemma *iso-pullback-ide*:
assumes *cospan* $\mu\ \nu$ **and** *ide* μ
shows *iso* $\mathrm{p}_0[\mu,\ \nu]$
proof −
  have *inverse-arrows* $\mathrm{p}_0[\mu,\ \nu]\ \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle$
  proof
    show *1*: *ide* $(\mathrm{p}_0[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle)$
      using *assms comp-arr-dom comp-cod-arr prj-tuple*(*1*) **by** *simp*
    show *ide* $(\langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu])$
    proof −
      have $\langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu] = (\mu \downarrow\downarrow \nu)$
      proof −
        have $\mathrm{p}_0[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu] = \mathrm{p}_0[\mu,\ \nu] \cdot (\mu \downarrow\downarrow \nu)$
        proof −
          have $\mathrm{p}_0[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu] = (\mathrm{p}_0[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle) \cdot \mathrm{p}_0[\mu,\ \nu]$
            using *assms 1 comp-reduce* **by** *blast*
          also have ... $= \mathrm{p}_0[\mu,\ \nu] \cdot (\mu \downarrow\downarrow \nu)$
            using *assms prj-tuple*(*1*) *pullback-commutes comp-arr-dom comp-cod-arr* **by** *simp*
          finally show *?thesis* **by** *blast*
        qed
        moreover have $\mathrm{p}_1[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu] = \mathrm{p}_1[\mu,\ \nu] \cdot (\mu \downarrow\downarrow \nu)$
        proof −
          have $\mathrm{p}_1[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle \cdot \mathrm{p}_0[\mu,\ \nu] = (\mathrm{p}_1[\mu,\ \nu] \cdot \langle \nu \ [\![\mu,\ \nu]\!]\ dom\ \nu \rangle) \cdot \mathrm{p}_0[\mu,\ \nu]$
            using *assms*(*2*) *comp-assoc* **by** *simp*
          also have ... $= \nu \cdot \mathrm{p}_0[\mu,\ \nu]$
            using *assms comp-arr-dom comp-cod-arr prj-tuple*(*2*) **by** *fastforce*
          also have ... $= \mu \cdot \mathrm{p}_1[\mu,\ \nu]$
            using *assms pullback-commutes commutative-square-def* **by** *simp*
          also have ... $= \mathrm{p}_1[\mu,\ \nu] \cdot (\mu \downarrow\downarrow \nu)$
          using *assms comp-arr-dom comp-cod-arr pullback-commutes commutative-square-def*
            by *simp*
          finally show *?thesis* **by** *simp*
        qed
        ultimately show *?thesis*
        using *assms prj0-in-hom prj1-in-hom comp-arr-dom prj1-simps*(*1*−*2*) *prj-joint-monic*

```
        by metis
    qed
    thus ?thesis
      using assms by auto
  qed
 qed
 thus ?thesis by auto
qed

lemma comp-tuple-arr:
assumes commutative-square f g h k and seq h l
shows ⟨h ⟦f, g⟧ k⟩ · l = ⟨h · l ⟦f, g⟧ k · l⟩
proof −
 have p0[f, g] · ⟨h ⟦f, g⟧ k⟩ · l = p0[f, g] · ⟨h · l ⟦f, g⟧ k · l⟩
 proof −
  have p0[f, g] · ⟨h ⟦f, g⟧ k⟩ · l = (p0[f, g] · ⟨h ⟦f, g⟧ k⟩) · l
   using comp-assoc by simp
  also have ... = p0[f, g] · ⟨h · l ⟦f, g⟧ k · l⟩
   using assms commutative-square-comp-arr by auto
  finally show ?thesis by blast
 qed
 moreover have p1[f, g] · ⟨h ⟦f, g⟧ k⟩ · l = p1[f, g] · ⟨h · l ⟦f, g⟧ k · l⟩
 proof −
  have p1[f, g] · ⟨h ⟦f, g⟧ k⟩ · l = (p1[f, g] · ⟨h ⟦f, g⟧ k⟩) · l
   using comp-assoc by simp
  also have ... = p1[f, g] · ⟨h · l ⟦f, g⟧ k · l⟩
   using assms commutative-square-comp-arr by auto
  finally show ?thesis by blast
 qed
 moreover have seq p1[f, g] (⟨h ⟦f, g⟧ k⟩ · l)
  using assms tuple-in-hom prj1-in-hom by fastforce
 ultimately show ?thesis
  using assms prj-joint-monic [of f g ⟨h ⟦f, g⟧ k⟩ · l ⟨h · l ⟦f, g⟧ k · l⟩]
  by auto
qed

lemma pullback-arr-cod:
assumes arr f
shows inverse-arrows p1[f, cod f] ⟨dom f ⟦f, cod f⟧ f⟩
and inverse-arrows p0[cod f, f] ⟨f ⟦cod f, f⟧ dom f⟩
proof −
 show inverse-arrows p1[f, cod f] ⟨dom f ⟦f, cod f⟧ f⟩
 proof
  show ide (⟨dom f ⟦f, cod f⟧ f⟩ · p1[f, cod f])
  proof −
   have ⟨dom f ⟦f, cod f⟧ f⟩ · p1[f, cod f] = f ↓↓ cod f
   proof −
    have p0[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩ · p1[f, cod f] = p0[f, cod f] · (f ↓↓ cod f)
    proof −
```

```
      have p₀[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩ · p₁[f, cod f] =
           (p₀[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩) · p₁[f, cod f]
        using comp-assoc by simp
      also have ... = p₀[f, cod f] · (f ↓↓ cod f)
        using assms pullback-commutes [of f cod f] comp-arr-dom comp-cod-arr
        by auto
      finally show ?thesis by blast
    qed
    moreover
    have p₁[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩ · p₁[f, cod f] = p₁[f, cod f] · (f ↓↓ cod f)
    proof −
      have p₁[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩ · p₁[f, cod f] =
           (p₁[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩) · p₁[f, cod f]
        using assms comp-assoc by presburger
      also have ... = p₁[f, cod f] · (f ↓↓ cod f)
        using assms comp-arr-dom comp-cod-arr by simp
      finally show ?thesis by blast
    qed
    ultimately show ?thesis
      using assms
            prj-joint-monic
              [of f cod f ⟨dom f ⟦f, cod f⟧ f⟩ · p₁[f, cod f] f ↓↓ cod f]
      by simp
  qed
  thus ?thesis
    using assms arr-cod cod-cod prj1-simps-arr by simp
qed
show ide (p₁[f, cod f] · ⟨dom f ⟦f, cod f⟧ f⟩)
  using assms comp-arr-dom comp-cod-arr by fastforce
qed
show inverse-arrows p₀[cod f, f] ⟨f ⟦cod f, f⟧ dom f⟩
proof
  show ide (p₀[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩)
    using assms comp-arr-dom comp-cod-arr by simp
  show ide (⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f])
  proof −
    have ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f] = cod f ↓↓ f
    proof −
      have p₀[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f] = p₀[cod f, f] · (cod f ↓↓ f)
      proof −
        have p₀[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f] =
             (p₀[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩) · p₀[cod f, f]
          using comp-assoc by simp
        also have ... = dom f · p₀[cod f, f]
          using assms comp-arr-dom comp-cod-arr by simp
        also have ... = p₀[cod f, f] · (cod f ↓↓ f)
          using assms comp-arr-dom comp-cod-arr by simp
        finally show ?thesis
          using prj0-in-hom by blast
```

```
        qed
        moreover
        have p₁[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f] = p₁[cod f, f] · (cod f ↓↓ f)
        proof −
          have p₁[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f] =
                (p₁[cod f, f] · ⟨f ⟦cod f, f⟧ dom f⟩) · p₀[cod f, f]
            using comp-assoc by simp
          also have ... = p₁[cod f, f] · (cod f ↓↓ f)
            using assms pullback-commutes [of cod f f] comp-arr-dom comp-cod-arr
            by auto
          finally show ?thesis by blast
        qed
        ultimately show ?thesis
          using assms prj-joint-monic [of cod f f ⟨f ⟦cod f, f⟧ dom f⟩ · p₀[cod f, f]]
          by simp
      qed
      thus ?thesis using assms by simp
    qed
  qed
qed
```

The pullback of a monomorphism along itself is automatically symmetric: the left and right projections are equal.

```
  lemma pullback-mono-self:
  assumes mono f
  shows p₀[f, f] = p₁[f, f]
  proof −
    have f · p₀[f, f] = f · p₁[f, f]
      using assms pullback-commutes [of f f]
      by (metis commutative-squareE mono-implies-arr)
    thus ?thesis
      using assms mono-cancel [of f p₁[f, f] p₀[f, f]]
      by (metis mono-implies-arr prj0-simps(1,3) seqI)
  qed

  lemma pullback-iso-self:
  assumes iso f
  shows p₀[f, f] = p₁[f, f]
    using assms pullback-mono-self iso-is-section section-is-mono by simp

  lemma pullback-ide-self [simp]:
  assumes ide a
  shows p₀[a, a] = p₁[a, a]
    using assms pullback-iso-self ide-is-iso by blast

end
```

21.5 Agreement between the Definitions

It is very easy to write locale assumptions that have unintended consequences or that are even inconsistent. So, to keep ourselves honest, we don't just accept the definition of "elementary category with pullbacks", but in fact we formally establish the sense in which it agrees with our standard definition of "category with pullbacks", which is given in terms of limit cones. This is extra work, but it ensures that we didn't make a mistake.

context *category-with-pullbacks*
begin

definition *some-prj1* (‹$p_1{}^?[-, -]$›)
where $p_1{}^?[f, g] \equiv$ *if cospan f g then*
fst (*SOME x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*))
else null

definition *some-prj0* (‹$p_0{}^?[-, -]$›)
where $p_0{}^?[f, g] \equiv$ *if cospan f g then*
snd (*SOME x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*))
else null

lemma *prj-yields-pullback*:
assumes *cospan f g*
shows *cospan-diagram.has-as-pullback C f g* $p_1{}^?[f, g]$ $p_0{}^?[f, g]$
proof −
have ∃*x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*)
using *assms has-pullbacks has-pullbacks-def has-as-pullback-def* **by** *simp*
thus *?thesis*
using *assms has-pullbacks has-pullbacks-def some-prj0-def some-prj1-def*
someI-ex [*of* λ*x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*)]
by *simp*
qed

interpretation *elementary-category-with-pullbacks C some-prj0 some-prj1*
proof
show ⋀*f g*. ¬ *cospan f g* ⟹ $p_0{}^?[f, g] =$ *null*
using *some-prj0-def* **by** *auto*
show ⋀*f g*. ¬ *cospan f g* ⟹ $p_1{}^?[f, g] =$ *null*
using *some-prj1-def* **by** *auto*
show ⋀*f g*. *cospan f g* ⟹ *commutative-square f g* $p_1{}^?[f, g]$ $p_0{}^?[f, g]$
proof
fix *f g*
assume *fg*: *cospan f g*
show *cospan f g* **by** *fact*
interpret *J*: *cospan-shape* **.**
interpret *D*: *cospan-diagram C f g*
using *fg* **by** (*unfold-locales*, *auto*)
let *?χ* = *D.mkCone* $p_1{}^?[f, g]$ $p_0{}^?[f, g]$
interpret *χ*: *limit-cone J.comp C D.map* ‹*dom* $p_1{}^?[f, g]$› *?χ*

```
    using fg prj-yields-pullback by auto
  have 1: p₁?[f, g] = ?χ J.AA ∧ p₀?[f, g] = ?χ J.BB
    using D.mkCone-def by simp
  show span p₁?[f, g] p₀?[f, g]
  proof −
    have arr p₁?[f, g] ∧ arr p₀?[f, g]
      using 1 J.arr-char J.seq-char
      by (metis J.seqE χ.preserves-reflects-arr)
    moreover have dom p₁?[f, g] = dom p₀?[f, g]
      using 1 D.is-rendered-commutative-by-cone χ.cone-axioms J.seq-char
      by (metis J.cod-eqI J.seqE χ.A.map-simp χ.preserves-dom J.ide-char)
    ultimately show ?thesis by simp
  qed
  show dom f = cod p₁?[f, g]
    using 1 fg χ.preserves-cod [of J.BB] J.cod-char D.mkCone-def
    by (metis D.map.simps(1) D.preserves-cod J.seqE χ.preserves-cod cod-dom J.seq-char)
  show f · p₁?[f, g] = g · p₀?[f, g]
    using 1 fg D.is-rendered-commutative-by-cone χ.cone-axioms by force
qed
show ⋀f g h k. commutative-square f g h k ⟹ ∃!l. p₁?[f, g] · l = h ∧ p₀?[f, g] · l = k
proof −
  fix f g h k
  assume fghk: commutative-square f g h k
  interpret J: cospan-shape .
  interpret D: cospan-diagram C f g
    using fghk by (unfold-locales, auto)
  let ?χ = D.mkCone p₁?[f, g] p₀?[f, g]
  interpret χ: limit-cone J.comp C D.map ‹dom p₁?[f, g]› ?χ
    using fghk prj-yields-pullback by auto
  interpret χ: pullback-cone C f g ‹p₁?[f, g]› ‹p₀?[f, g]› ..
  have 1: p₁?[f, g] = ?χ J.AA ∧ p₀?[f, g] = ?χ J.BB
    using D.mkCone-def by simp
  show ∃!l. p₁?[f, g] · l = h ∧ p₀?[f, g] · l = k
  proof
    let ?l = SOME l. p₁?[f, g] · l = h ∧ p₀?[f, g] · l = k
    show p₁?[f, g] · ?l = h ∧ p₀?[f, g] · ?l = k
      using fghk χ.is-universal′ χ.renders-commutative
            someI-ex [of λl. p₁?[f, g] · l = h ∧ p₀?[f, g] · l = k]
      by blast
    thus ⋀l. p₁?[f, g] · l = h ∧ p₀?[f, g] · l = k ⟹ l = ?l
      using fghk χ.is-universal′ χ.renders-commutative limit-cone-def
      by (metis (no-types, lifting) in-homI seqE commutative-squareE dom-comp seqI)
  qed
qed
qed

proposition extends-to-elementary-category-with-pullbacks:
shows elementary-category-with-pullbacks C some-prj0 some-prj1
  ..
```

end

context *elementary-category-with-pullbacks*
begin

interpretation *category-with-pullbacks C*
proof
show *has-pullbacks*
proof (*unfold has-pullbacks-def*)
have $\bigwedge f\ g.\ cospan\ f\ g \Longrightarrow \exists\, p0\ p1.\ has\text{-}as\text{-}pullback\ f\ g\ p0\ p1$
proof −
fix *f g*
assume *fg*: *cospan f g*
have *has-as-pullback f g* $\mathrm{p}_1[f,\ g]$ $\mathrm{p}_0[f,\ g]$
using *fg has-as-pullbackI pullback-commutes universal* **by** *presburger*
thus $\exists\, p0\ p1.\ has\text{-}as\text{-}pullback\ f\ g\ p0\ p1$ **by** *blast*
qed
thus $\forall f\ g.\ cospan\ f\ g \longrightarrow (\exists\, p0\ p1.\ has\text{-}as\text{-}pullback\ f\ g\ p0\ p1)$
by *simp*
qed
qed

proposition *is-category-with-pullbacks*:
shows *category-with-pullbacks C*
..

end

sublocale *elementary-category-with-pullbacks* $\subseteq$ *category-with-pullbacks*
using *is-category-with-pullbacks* **by** *auto*

end

Chapter 22

Cartesian Category

In this chapter, we explore the notion of a "cartesian category", which we define to be a category having binary products and a terminal object. We show that every cartesian category extends to an "elementary cartesian category", whose definition assumes that specific choices have been made for projections and terminal object. Conversely, the underlying category of an elementary cartesian category is a cartesian category. We also show that cartesian categories are the same thing as categories with finite products.

theory *CartesianCategory*
imports *Limit SetCat CategoryWithPullbacks*
begin

22.1 Category with Binary Products

22.1.1 Binary Product Diagrams

The "shape" of a binary product diagram is a category having two distinct identity arrows and no non-identity arrows.

locale *binary-product-shape*
begin

sublocale *concrete-category* ‹*UNIV* :: *bool set*› ‹λ*a b*. *if a* = *b then* {()} *else* {}›
‹λ-. ()› ‹λ- - - - -. ()›
apply (*unfold-locales*, *auto*)
apply (*meson empty-iff*)
by (*meson empty-iff*)

abbreviation *comp*
where *comp* ≡ *COMP*

abbreviation *FF*
where *FF* ≡ *MkIde False*

abbreviation *TT*

```
  where TT ≡ MkIde True

  lemma arr-char:
  shows arr f ⟷ f = FF ∨ f = TT
    using arr-char by (cases f, simp-all)

  lemma ide-char:
  shows ide f ⟷ f = FF ∨ f = TT
    using ide-char_CC ide-MkIde by (cases f, auto)

  lemma is-discrete:
  shows ide f ⟷ arr f
    using arr-char ide-char by simp

  lemma dom-simp [simp]:
  assumes arr f
  shows dom f = f
    using assms is-discrete by simp

  lemma cod-simp [simp]:
  assumes arr f
  shows cod f = f
    using assms is-discrete by simp

  lemma seq-char:
  shows seq f g ⟷ arr f ∧ f = g
    by auto

  lemma comp-simp [simp]:
  assumes seq f g
  shows comp f g = f
    using assms seq-char by fastforce

end

locale binary-product-diagram =
  J: binary-product-shape +
  C: category C
for C :: 'c comp      (infixr ‹·› 55)
and a0 :: 'c
and a1 :: 'c +
assumes is-discrete: C.ide a0 ∧ C.ide a1
begin

  notation J.comp      (infixr ‹·_J› 55)

  fun map
  where map J.FF = a0
      | map J.TT = a1
```

```
    | map - = C.null

  sublocale diagram J.comp C map
  proof
    show ⋀f. ¬ J.arr f ⟹ map f = C.null
      using J.arr-char map.elims by auto
    fix f
    assume f: J.arr f
    show C.arr (map f)
      using f J.arr-char is-discrete C.ideD(1) map.simps(1−2) by metis
    show C.dom (map f) = map (J.dom f)
      using f J.arr-char J.dom-char is-discrete by force
    show C.cod (map f) = map (J.cod f)
      using f J.arr-char J.cod-char is-discrete by force
    next
    fix f g
    assume fg: J.seq g f
    show map (g ·J f) = map g · map f
      using fg J.arr-char J.seq-char J.null-char J.not-arr-null is-discrete
      by (metis (no-types, lifting) C.comp-ide-self J.comp-simp map.simps(1−2))
  qed

end
```

22.1.2 Category with Binary Products

A *binary product* in a category C is a limit of a binary product diagram in C.

```
  context binary-product-diagram
  begin

    definition mkCone
    where mkCone p0 p1 ≡ λj. if j = J.FF then p0 else if j = J.TT then p1 else C.null

    abbreviation is-rendered-commutative-by
    where is-rendered-commutative-by p0 p1 ≡
          C.seq a0 p0 ∧ C.seq a1 p1 ∧ C.dom p0 = C.dom p1

    abbreviation has-as-binary-product
    where has-as-binary-product p0 p1 ≡ limit-cone (C.dom p0) (mkCone p0 p1)

    lemma cone-mkCone:
    assumes is-rendered-commutative-by p0 p1
    shows cone (C.dom p0) (mkCone p0 p1)
    proof −
      interpret E: constant-functor J.comp C ‹C.dom p0›
        using assms by unfold-locales auto
      show cone (C.dom p0) (mkCone p0 p1)
      using assms mkCone-def J.arr-char E.map-simp is-discrete C.comp-ide-arr C.comp-arr-dom
        by unfold-locales auto
```

qed

lemma *is-rendered-commutative-by-cone*:
assumes *cone a* χ
shows *is-rendered-commutative-by* (χ *J.FF*) (χ *J.TT*)
proof −
  interpret χ: *cone J.comp C map a* χ
    using *assms* **by** *auto*
  show *?thesis*
    using *is-discrete* **by** *simp*
qed

lemma *mkCone-cone*:
assumes *cone a* χ
shows *mkCone* (χ *J.FF*) (χ *J.TT*) = χ
proof −
  interpret χ: *cone J.comp C map a* χ
    using *assms* **by** *auto*
 interpret *mkCone-*χ: *cone J.comp C map* ‹*C.dom* (χ *J.FF*)› ‹*mkCone* (χ *J.FF*) (χ *J.TT*)›
    using *assms is-rendered-commutative-by-cone cone-mkCone* **by** *blast*
  show *?thesis*
    using *mkCone-def* χ*.extensionality J.ide-char mkCone-def*
        natural-transformation-eqI [*of J.comp C*]
        χ*.natural-transformation-axioms mkCone-*χ*.natural-transformation-axioms*
    by *fastforce*
qed

lemma *cone-iff-span*:
shows *cone* (*C.dom h*) (*mkCone h k*) ⟷ *C.span h k* ∧ *C.cod h* = *a0* ∧ *C.cod k* = *a1*
  by (*metis* (*no-types*, *lifting*) *C.cod-eqI C.comp-cod-arr C.comp-ide-arr J.arr.simps*(*1*)
      cone-mkCone is-discrete is-rendered-commutative-by-cone mkCone-def)

lemma *cones-map-mkCone-eq-iff*:
assumes *is-rendered-commutative-by p0 p1* **and** *is-rendered-commutative-by p0′ p1′*
and «*h* : *C.dom p0′* → *C.dom p0*»
shows *cones-map h* (*mkCone p0 p1*) = *mkCone p0′ p1′* ⟷ *p0* ⋅ *h* = *p0′* ∧ *p1* ⋅ *h* = *p1′*
proof −
  interpret χ: *cone J.comp C map* ‹*C.dom p0*› ‹*mkCone p0 p1*›
    using *assms*(*1*) *cone-mkCone* [*of p0 p1*] **by** *blast*
  interpret χ': *cone J.comp C map* ‹*C.dom p0′*› ‹*mkCone p0′ p1′*›
    using *assms*(*2*) *cone-mkCone* [*of p0′ p1′*] **by** *blast*
  show *?thesis*
  proof
    assume *1*: *cones-map h* (*mkCone p0 p1*) = *mkCone p0′ p1′*
    show *p0* ⋅ *h* = *p0′* ∧ *p1* ⋅ *h* = *p1′*
    proof
      show *p0* ⋅ *h* = *p0′*
      proof −
        have *p0′* = *cones-map h* (*mkCone p0 p1*) *J.FF*

```
        using 1 mkCone-def J.arr-char by simp
      also have ... = p0 · h
        using assms mkCone-def J.arr-char χ.cone-axioms by auto
      finally show ?thesis by auto
    qed
    show p1 · h = p1'
    proof −
      have p1' = cones-map h (mkCone p0 p1) J.TT
        using 1 mkCone-def J.arr-char by simp
      also have ... = p1 · h
        using assms mkCone-def J.arr-char χ.cone-axioms by auto
      finally show ?thesis by auto
    qed
  qed
  next
  assume p0 · h = p0' ∧ p1 · h = p1'
  thus cones-map h (mkCone p0 p1) = mkCone p0' p1'
    using assms χ.cone-axioms mkCone-def J.arr-char by auto
 qed
qed

end

locale binary-product-cone =
  J: binary-product-shape +
  C: category C +
  D: binary-product-diagram C f0 f1 +
  limit-cone J.comp C D.map ‹C.dom p0› ‹D.mkCone p0 p1›
for C :: 'c comp      (infixr ‹·› 55)
and f0 :: 'c
and f1 :: 'c
and p0 :: 'c
and p1 :: 'c
begin

  lemma renders-commutative:
  shows D.is-rendered-commutative-by p0 p1
    using cone-axioms D.is-rendered-commutative-by-cone D.mkCone-def D.cone-iff-span
    by force

  lemma is-universal':
  assumes D.is-rendered-commutative-by p0' p1'
  shows ∃!h. «h : C.dom p0' → C.dom p0» ∧ p0 · h = p0' ∧ p1 · h = p1'
  proof −
    have D.cone (C.dom p0') (D.mkCone p0' p1')
      using assms D.cone-mkCone by blast
    hence ∃!h. «h : C.dom p0' → C.dom p0» ∧
              D.cones-map h (D.mkCone p0 p1) = D.mkCone p0' p1'
      using is-universal by simp
```

```
    moreover have ⋀h. «h : C.dom p0' → C.dom p0» ⟹
                      D.cones-map h (D.mkCone p0 p1) = D.mkCone p0' p1' ⟷
                      p0 · h = p0' ∧ p1 · h = p1'
      using assms D.cones-map-mkCone-eq-iff [of p0 p1 p0' p1'] renders-commutative
      by blast
    ultimately show ?thesis by blast
  qed

  lemma induced-arrowI':
  assumes D.is-rendered-commutative-by p0' p1'
  shows «induced-arrow (C.dom p0') (D.mkCone p0' p1') : C.dom p0' → C.dom p0»
  and p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') = p0'
  and p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') = p1'
  proof −
    interpret A': constant-functor J.comp C ‹C.dom p0'›
      using assms by (unfold-locales, auto)
    have cone: D.cone (C.dom p0') (D.mkCone p0' p1')
      using assms D.cone-mkCone [of p0' p1'] by blast
    show 0: p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') = p0'
    proof −
      have p0 · induced-arrow (C.dom p0') (D.mkCone p0' p1') =
            D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1')) (D.mkCone p0 p1)
J.FF
        using cone induced-arrowI(1) D.mkCone-def J.arr-char is-cone by force
      also have ... = p0'
        by (metis (no-types, lifting) D.mkCone-def cone induced-arrowI(2)
            mem-Collect-eq restrict-apply)
      finally show ?thesis by auto
    qed
    show p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') = p1'
    proof −
      have p1 · induced-arrow (C.dom p1') (D.mkCone p0' p1') =
            D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1')) (D.mkCone p0 p1)
J.TT
        using assms cone induced-arrowI(1) D.mkCone-def J.arr-char is-cone by fastforce
      also have ... = p1'
      proof −
        have D.cones-map (induced-arrow (C.dom p0') (D.mkCone p0' p1')) (D.mkCone p0
p1) =
             D.mkCone p0' p1'
          using cone induced-arrowI by blast
        thus ?thesis
          using J.arr-char D.mkCone-def by simp
      qed
      finally show ?thesis by auto
    qed
    show «induced-arrow (C.dom p0') (D.mkCone p0' p1') : C.dom p0' → C.dom p0»
      using 0 cone induced-arrowI by simp
  qed
```

end

context *category*
begin

definition *has-as-binary-product*
where *has-as-binary-product a0 a1 p0 p1* ≡
ide a0 ∧ *ide a1* ∧ *binary-product-diagram.has-as-binary-product C a0 a1 p0 p1*

definition *has-binary-products*
where *has-binary-products* =
(∀ *a0 a1*. *ide a0* ∧ *ide a1* ⟶ (∃ *p0 p1*. *has-as-binary-product a0 a1 p0 p1*))

lemma *has-as-binary-productI* [*intro*]:
assumes *span p q* **and** *cod p* = *a* **and** *cod q* = *b*
and ⋀*x f g*. ⟦«*f* : *x* → *a*»; «*g* : *x* → *b*»⟧ ⟹ ∃!*h*. «*h* : *x* → *dom p*» ∧ *p* · *h* = *f* ∧ *q* · *h* = *g*
shows *has-as-binary-product a b p q*
proof (*unfold has-as-binary-product-def*, *intro conjI*)
show *a*: *ide a*
using *assms* **by** *auto*
show *b*: *ide b*
using *assms* **by** *auto*
let *?c* = *dom p*
interpret *J*: *binary-product-shape* **.**
interpret *D*: *binary-product-diagram C a b*
using *a b* **by** *unfold-locales auto*
show *D.has-as-binary-product p q*
proof −
have *1*: *D.is-rendered-commutative-by p q*
using *assms a b ide-in-hom* **by** *blast*
let *?χ* = *D.mkCone p q*
interpret *χ*: *cone J.comp C D.map ?c ?χ*
using *assms*(*4*) *D.cone-mkCone 1* **by** *auto*
interpret *χ*: *limit-cone J.comp C D.map ?c ?χ*
proof
fix *x χ′*
assume *χ′*: *D.cone x χ′*
interpret *χ′*: *cone J.comp C D.map x χ′*
using *χ′* **by** *simp*
have *2*: ∃!*h*. «*h* : *x* → *?c*» ∧ *p* · *h* = *χ′ J.FF* ∧ *q* · *h* = *χ′ J.TT*
proof −
have «*χ′ J.FF* : *x* → *a*» ∧ «*χ′ J.TT* : *x* → *b*»
by *auto*
thus *?thesis*
using *assms*(*4*) [*of χ′ J.FF x χ′ J.TT*] **by** *simp*
qed
have *3*: *D.is-rendered-commutative-by* (*χ′ J.FF*) (*χ′ J.TT*)
using *a b* **by** *force*

```
      obtain h where h: «h : x → ?c» ∧ p · h = χ' J.FF ∧ q · h = χ' J.TT
        using 2 by blast
      have 4: «h : dom (χ' (J.MkIde False)) → dom p»
        using assms(3) h by auto
      have «h : x → ?c» ∧ D.cones-map h (D.mkCone p q) = χ'
      proof (intro conjI)
        show «h : x → ?c»
          using h by blast
        show D.cones-map h (D.mkCone p q) = χ'
        proof
          fix j
          show D.cones-map h (D.mkCone p q) j = χ' j
            using h 1 3 4 D.cones-map-mkCone-eq-iff [of p q χ' J.FF χ' J.TT]
                  χ.cone-axioms J.is-discrete χ'.extensionality
                  D.mkCone-def binary-product-shape.ide-char
            apply (cases J.ide j)
            by (metis (no-types, lifting))+
        qed
      qed
     moreover have ⋀h'. «h' : x → ?c» ∧ D.cones-map h' (D.mkCone p q) = χ' ⟹ h' = h
      proof −
        fix h'
        assume 5: «h' : x → ?c» ∧ D.cones-map h' (D.mkCone p q) = χ'
        have ∃!h. «h : x → ?c» ∧ p · h = χ' J.FF ∧ q · h = χ' J.TT
          by (simp add: assms(4) in-homI)
        moreover have «h : x → ?c» ∧ χ' J.FF = p · h ∧ q · h = χ' J.TT
          using h by simp
        moreover have «h' : x → ?c» ∧ χ' J.FF = p · h' ∧ q · h' = χ' J.TT
          using 5 χ.cone-axioms D.mkCone-def [of p q] by auto
        ultimately show h' = h by auto
      qed
      ultimately show ∃!h. «h : x → ?c» ∧ D.cones-map h (D.mkCone p q) = χ'
        by blast
    qed
    show D.has-as-binary-product p q
      using assms χ.limit-cone-axioms by blast
  qed
qed

lemma has-as-binary-productE [elim]:
assumes has-as-binary-product a b p q
and ⟦«p : dom p → a»; «q : dom p → b»;
     ⋀x f g. ⟦«f : x → a»; «g : x → b»⟧ ⟹ ∃!h. p · h = f ∧ q · h = g⟧ ⟹ T
shows T
proof −
  interpret J: binary-product-shape .
  interpret D: binary-product-diagram C a b
    using assms(1) has-as-binary-product-def
    by (simp add: binary-product-diagram.intro binary-product-diagram-axioms.intro
```

```
        category-axioms)
have 1: ⋀h k. span h k ∧ cod h = a ∧ cod k = b ⟷ D.cone (dom h) (D.mkCone h k)
  using D.cone-iff-span by presburger
let ?χ = D.mkCone p q
interpret χ: limit-cone J.comp C D.map ‹dom p› ?χ
  using assms(1) has-as-binary-product-def D.cone-mkCone by blast
have span: span p q
  using 1 χ.cone-axioms by blast
moreover have «p : dom p → a» ∧ «q : dom p → b»
  using span χ.preserves-hom χ.cone-axioms binary-product-shape.arr-char
  by (metis D.cone-iff-span arr-iff-in-hom)
moreover have ⋀x f g. ⟦«f : x → a»; «g : x → b»⟧ ⟹ ∃!l. p · l = f ∧ q · l = g
proof −
  fix x f g
  assume f: «f : x → a» and g: «g : x → b»
  let ?χ′ = D.mkCone f g
  interpret χ′: cone J.comp C D.map x ?χ′
    using 1 f g by blast
  have 2: ∃!l. «l : x → dom p» ∧ D.cones-map l ?χ = ?χ′
    using 1 f g χ.is-universal [of x D.mkCone f g] χ′.cone-axioms by fastforce
  obtain l where l: «l : x → dom p» ∧ D.cones-map l ?χ = ?χ′
    using 2 by blast
  have p · l = f ∧ q · l = g
  proof
    show p · l = f
    proof −
      have p · l = ?χ J.FF · l
        using D.mkCone-def by presburger
      also have ... = D.cones-map l ?χ J.FF
        using χ.cone-axioms
        apply simp
        using l by fastforce
      also have ... = f
        using D.mkCone-def l by presburger
      finally show ?thesis by blast
    qed
    show q · l = g
    proof −
      have q · l = ?χ J.TT · l
        using D.mkCone-def by simp
      also have ... = D.cones-map l ?χ J.TT
        using χ.cone-axioms
        apply simp
        using l by fastforce
      also have ... = g
        using D.mkCone-def l by simp
      finally show q · l = g by blast
    qed
  qed
```

```
    moreover have ⋀l'. p · l' = f ∧ q · l' = g ⟹ l' = l
    proof −
      fix l'
      assume 1: p · l' = f ∧ q · l' = g
      have 2: «l' : x → dom p»
        using 1 f by blast
      moreover have D.cones-map l' ?χ = ?χ'
        using 1 2 D.cones-map-mkCone-eq-iff [of p q f g l']
        by (metis (no-types, lifting) f g ‹«p : dom p → a» ∧ «q : dom p → b»›
            comp-cod-arr in-homE)
      ultimately show l' = l
        using l χ.is-universal χ'.cone-axioms by blast
    qed
    ultimately show ∃!l. p · l = f ∧ q · l = g by blast
  qed
  ultimately show T
    using assms(2) by simp
qed

end

locale category-with-binary-products =
  category +
assumes has-binary-products: has-binary-products
```

22.1.3 Elementary Category with Binary Products

An *elementary category with binary products* is a category equipped with a specific way of mapping each pair of objects a and b to a pair of arrows $\mathfrak{p}_1[a,\ b]$ and $\mathfrak{p}_0[a,\ b]$ that comprise a universal span.

```
locale elementary-category-with-binary-products =
  category C
for C :: 'a comp                                    (infixr ‹·› 55)
and pr0 :: 'a ⇒ 'a ⇒ 'a                              (‹𝔭₀[-, -]›)
and pr1 :: 'a ⇒ 'a ⇒ 'a                              (‹𝔭₁[-, -]›) +
assumes span-pr: ⟦ ide a; ide b ⟧ ⟹ span 𝔭₁[a, b] 𝔭₀[a, b]
and cod-pr0: ⟦ ide a; ide b ⟧ ⟹ cod 𝔭₀[a, b] = b
and cod-pr1: ⟦ ide a; ide b ⟧ ⟹ cod 𝔭₁[a, b] = a
and universal: span f g ⟹ ∃!l. 𝔭₁[cod f, cod g] · l = f ∧ 𝔭₀[cod f, cod g] · l = g
begin

  lemma pr0-in-hom':
  assumes ide a and ide b
  shows «𝔭₀[a, b] : dom 𝔭₀[a, b] → b»
    using assms span-pr cod-pr0 by auto

  lemma pr1-in-hom':
  assumes ide a and ide b
  shows «𝔭₁[a, b] : dom 𝔭₀[a, b] → a»
```

using *assms span-pr cod-pr1* **by** *auto*

We introduce a notation for tupling, which denotes the arrow into a product that is induced by a span.

definition *tuple* (‹⟨-, -⟩›)
where $\langle f,\ g\rangle \equiv$ *if span f g then*
THE l. $\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$
else null

The following defines product of arrows (not just of objects). It will take a little while before we can prove that it is functorial, but for right now it is nice to have it as a notation for the apex of a product cone. We have to go through some slightly unnatural contortions in the development here, though, to avoid having to introduce a separate preliminary notation just for the product of objects.

definition *prod* (**infixr** ‹⊗› *51*)
where $f \otimes g \equiv \langle f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g],\ g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g]\rangle$

lemma *seq-pr-tuple*:
assumes *span f g*
shows *seq* $\mathfrak{p}_0[cod\ f,\ cod\ g]$ $\langle f,\ g\rangle$
proof −
have $\mathfrak{p}_0[cod\ f,\ cod\ g] \cdot \langle f,\ g\rangle = g$
unfolding *tuple-def*
using *assms universal theI* [*of* $\lambda l.\ \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$]
by *simp meson*
thus *?thesis*
using *assms* **by** *simp*
qed

lemma *tuple-pr-arr*:
assumes *ide a* **and** *ide b* **and** *seq* $\mathfrak{p}_0[a,\ b]$ *h*
shows $\langle \mathfrak{p}_1[a,\ b] \cdot h,\ \mathfrak{p}_0[a,\ b] \cdot h\rangle = h$
unfolding *tuple-def*
using *assms span-pr cod-pr0 cod-pr1 universal* [*of* $\mathfrak{p}_1[a,\ b] \cdot h$ $\mathfrak{p}_0[a,\ b] \cdot h$]
theI-unique [*of* $\lambda l.\ \mathfrak{p}_1[a,\ b] \cdot l = \mathfrak{p}_1[a,\ b] \cdot h \wedge \mathfrak{p}_0[a,\ b] \cdot l = \mathfrak{p}_0[a,\ b] \cdot h\ h$]
by *auto*

lemma *pr-tuple* [*simp*]:
assumes *span f g* **and** *cod f* = *a* **and** *cod g* = *b*
shows $\mathfrak{p}_1[a,\ b] \cdot \langle f,\ g\rangle = f$ **and** $\mathfrak{p}_0[a,\ b] \cdot \langle f,\ g\rangle = g$
proof −
have *1*: $\mathfrak{p}_1[a,\ b] \cdot \langle f,\ g\rangle = f \wedge \mathfrak{p}_0[a,\ b] \cdot \langle f,\ g\rangle = g$
unfolding *tuple-def*
using *assms universal theI* [*of* $\lambda l.\ \mathfrak{p}_1[a,\ b] \cdot l = f \wedge \mathfrak{p}_0[a,\ b] \cdot l = g$]
by *simp meson*
show $\mathfrak{p}_1[a,\ b] \cdot \langle f,\ g\rangle = f$ **using** *1* **by** *simp*
show $\mathfrak{p}_0[a,\ b] \cdot \langle f,\ g\rangle = g$ **using** *1* **by** *simp*
qed

lemma *cod-tuple*:
assumes *span f g*
shows *cod* ⟨*f*, *g*⟩ = *cod f* ⊗ *cod g*
proof −
  have *cod f* ⊗ *cod g* = ⟨𝔭₁[*cod f*, *cod g*], 𝔭₀[*cod f*, *cod g*]⟩
    unfolding *prod-def*
    using *assms comp-cod-arr span-pr cod-pr0 cod-pr1* **by** *simp*
  also have ... = ⟨𝔭₁[*cod f*, *cod g*] · *dom* 𝔭₀[*cod f*, *cod g*],
                𝔭₀[*cod f*, *cod g*] · *dom* 𝔭₀[*cod f*, *cod g*]⟩
    using *assms span-pr comp-arr-dom* **by** *simp*
  also have ... = *dom* 𝔭₀[*cod f*, *cod g*]
    using *assms tuple-pr-arr span-pr* **by** *simp*
  also have ... = *cod* ⟨*f*, *g*⟩
    using *assms seq-pr-tuple* **by** *blast*
  finally show *?thesis* **by** *simp*
qed

lemma *tuple-in-hom* [*intro*]:
assumes «*f* : *a* → *b*» **and** «*g* : *a* → *c*»
shows «⟨*f*, *g*⟩ : *a* → *b* ⊗ *c*»
  using *assms pr-tuple dom-comp cod-tuple*
  apply (*elim in-homE*, *intro in-homI*)
    apply (*metis seqE*)
  by *metis*+

lemma *tuple-in-hom′* [*simp*]:
assumes *arr f* **and** *dom f* = *a* **and** *cod f* = *b*
and *arr g* **and** *dom g* = *a* **and** *cod g* = *c*
shows «⟨*f*, *g*⟩ : *a* → *b* ⊗ *c*»
  using *assms* **by** *auto*

lemma *tuple-ext*:
assumes ¬ *span f g*
shows ⟨*f*, *g*⟩ = *null*
  unfolding *tuple-def*
  by (*simp add*: *assms*)

lemma *tuple-simps* [*simp*]:
assumes *span f g*
shows *arr* ⟨*f*, *g*⟩ **and** *dom* ⟨*f*, *g*⟩ = *dom f* **and** *cod* ⟨*f*, *g*⟩ = *cod f* ⊗ *cod g*
proof −
  show *arr* ⟨*f*, *g*⟩
    using *assms tuple-in-hom* **by** *blast*
  show *dom* ⟨*f*, *g*⟩ = *dom f*
    using *assms tuple-in-hom*
    by (*metis dom-comp pr-tuple*(*1*))
  show *cod* ⟨*f*, *g*⟩ = *cod f* ⊗ *cod g*
    using *assms cod-tuple* **by** *auto*
qed

lemma *tuple-pr* [*simp*]:
assumes *ide a* **and** *ide b*
shows $\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle = a \otimes b$
proof −
  have *1*: *dom* $\mathfrak{p}_0[a,\ b] = a \otimes b$
    using *assms seq-pr-tuple cod-tuple* [*of* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$] *span-pr*
        pr0-in-hom′ pr1-in-hom′
    by (*metis cod-pr0 cod-pr1 seqE*)
  hence $\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle = \langle \mathfrak{p}_1[a,\ b] \cdot (a \otimes b),\ \mathfrak{p}_0[a,\ b] \cdot (a \otimes b)\rangle$
    using *assms pr0-in-hom′ pr1-in-hom′ comp-arr-dom span-pr* **by** *simp*
  thus *?thesis*
    using *assms 1 tuple-pr-arr span-pr*
    by (*metis comp-arr-dom*)
qed

lemma *pr-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b* **and** $x = a \otimes b$
shows «$\mathfrak{p}_0[a,\ b] : x \to b$» **and** «$\mathfrak{p}_1[a,\ b] : x \to a$»
proof −
  show *0*: «$\mathfrak{p}_0[a,\ b] : x \to b$»
    using *assms pr0-in-hom′ seq-pr-tuple* [*of* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$]
        cod-tuple [*of* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$] *span-pr cod-pr0 cod-pr1*
    by (*intro in-homI*, *auto*)
  show «$\mathfrak{p}_1[a,\ b] : x \to a$»
    using *assms 0 span-pr pr1-in-hom′* **by** *fastforce*
qed

lemma *pr-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* $\mathfrak{p}_0[a,\ b]$ **and** *dom* $\mathfrak{p}_0[a,\ b] = a \otimes b$ **and** *cod* $\mathfrak{p}_0[a,\ b] = b$
and *arr* $\mathfrak{p}_1[a,\ b]$ **and** *dom* $\mathfrak{p}_1[a,\ b] = a \otimes b$ **and** *cod* $\mathfrak{p}_1[a,\ b] = a$
  using *assms pr-in-hom* **by** *blast+*

lemma *pr-joint-monic*:
assumes *ide a* **and** *ide b* **and** *seq* $\mathfrak{p}_0[a,\ b]$ *h*
and $\mathfrak{p}_0[a,\ b] \cdot h = \mathfrak{p}_0[a,\ b] \cdot h'$ **and** $\mathfrak{p}_1[a,\ b] \cdot h = \mathfrak{p}_1[a,\ b] \cdot h'$
shows $h = h'$
  using *assms* **by** (*metis tuple-pr-arr*)

lemma *comp-tuple-arr* [*simp*]:
assumes *span f g* **and** *arr h* **and** *dom f* = *cod h*
shows $\langle f,\ g\rangle \cdot h = \langle f \cdot h,\ g \cdot h\rangle$
proof (*intro pr-joint-monic* [**where** $h = \langle f,\ g\rangle \cdot h$])
  show *ide* (*cod f*) **and** *ide* (*cod g*)
    using *assms ide-cod* **by** *blast+*
  show *seq* $\mathfrak{p}_0$[*cod f*, *cod g*] ($\langle f,\ g\rangle \cdot h$)
    using *assms* **by** *fastforce*
  show $\mathfrak{p}_0$[*cod f*, *cod g*] $\cdot \langle f,\ g\rangle \cdot h = \mathfrak{p}_0$[*cod f*, *cod g*] $\cdot \langle f \cdot h,\ g \cdot h\rangle$

using *assms*(*1*−*3*) *comp-reduce* **by** *auto*
show $\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot \langle f,\ g\rangle \cdot h = \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot \langle f \cdot h,\ g \cdot h\rangle$
using *assms comp-reduce* **by** *auto*
qed

lemma *ide-prod* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *ide* $(a \otimes b)$
using *assms pr-simps ide-dom* [*of* $\mathfrak{p}_0[a,\ b]$] **by** *simp*

lemma *prod-in-hom* [*intro*]:
assumes «$f : a \to c$» **and** «$g : b \to d$»
shows «$f \otimes g : a \otimes b \to c \otimes d$»
using *assms prod-def* **by** *fastforce*

lemma *prod-in-hom′* [*simp*]:
assumes *arr f* **and** $dom\ f = a$ **and** $cod\ f = c$
and *arr g* **and** $dom\ g = b$ **and** $cod\ g = d$
shows «$f \otimes g : a \otimes b \to c \otimes d$»
using *assms* **by** *blast*

lemma *prod-simps* [*simp*]:
assumes *arr f0* **and** *arr f1*
shows *arr* (*f0* $\otimes$ *f1*)
and *dom* (*f0* $\otimes$ *f1*) = *dom f0* $\otimes$ *dom f1*
and *cod* (*f0* $\otimes$ *f1*) = *cod f0* $\otimes$ *cod f1*
using *assms prod-in-hom* **by** *blast*+

lemma *has-as-binary-product*:
assumes *ide a* **and** *ide b*
shows *has-as-binary-product a b* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$
proof
show *span* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$ **and** *cod* $\mathfrak{p}_1[a,\ b] = a$ **and** *cod* $\mathfrak{p}_0[a,\ b] = b$
using *assms* **by** *auto*
fix *x f g*
assume *f*: «$f : x \to a$» **and** *g*: «$g : x \to b$»
have *span f g*
using *f g* **by** *auto*
hence $\exists!l.\ \mathfrak{p}_1[a,\ b] \cdot l = f \wedge \mathfrak{p}_0[a,\ b] \cdot l = g$
using *assms f g universal* [*of f g*]
by (*metis in-homE*)
thus $\exists!h.$ «$h : x \to dom\ \mathfrak{p}_1[a,\ b]$» $\wedge\ \mathfrak{p}_1[a,\ b] \cdot h = f \wedge \mathfrak{p}_0[a,\ b] \cdot h = g$
using *f* **by** *blast*
qed

end

lemma (**in** *category*) *elementary-category-with-binary-productsI*:
assumes $\bigwedge a\ b.\ [\![ide\ a;\ ide\ b]\!] \Longrightarrow$ *has-as-binary-product a b* (*p a b*) (*q a b*)

shows *elementary-category-with-binary-products C*
(λ*a b*. *if ide a ∧ ide b then q a b else null*)
(λ*a b*. *if ide a ∧ ide b then p a b else null*)
proof −
let *?p* = λ*a b*. *if ide a ∧ ide b then p a b else null*
let *?q* = λ*a b*. *if ide a ∧ ide b then q a b else null*
show *?thesis*
proof
fix *a b*
assume *a*: *ide a* **and** *b*: *ide b*
show *span* (*?p a b*) (*?q a b*)
using *assms a b*
by *auto force*
show *cod* (*?q a b*) = *b*
using *a b assms has-as-binary-productE* **by** *auto*
show *cod* (*?p a b*) = *a*
using *a b assms has-as-binary-productE* **by** *auto*
next
fix *f g*
assume *fg*: *span f g*
have *1*: *has-as-binary-product* (*cod f*) (*cod g*)
(*p* (*cod f*) (*cod g*)) (*q* (*cod f*) (*cod g*))
using *assms fg* **by** *simp*
obtain *l* **where** *p* (*cod f*) (*cod g*) · *l* = *f* ∧ *q* (*cod f*) (*cod g*) · *l* = *g*
using *1 fg has-as-binary-productE*
by (*metis in-homI*)
hence ∃ *l*. *?p* (*cod f*) (*cod g*) · *l* = *f* ∧ *?q* (*cod f*) (*cod g*) · *l* = *g*
using *fg* **by** *auto*
moreover have
⋀*l l′*. *?p* (*cod f*) (*cod g*) · *l* = *f* ∧ *?q* (*cod f*) (*cod g*) · *l* = *g* ∧
?p (*cod f*) (*cod g*) · *l′* = *f* ∧ *?q* (*cod f*) (*cod g*) · *l′* = *g*
⟹ *l* = *l′*
using *1 fg arr-iff-in-hom ide-cod*
by (*elim has-as-binary-productE*, *auto*) *metis*
ultimately show ∃!*l*. *?p* (*cod f*) (*cod g*) · *l* = *f* ∧ *?q* (*cod f*) (*cod g*) · *l* = *g*
by *blast*
qed
qed

22.1.4 Agreement between the Definitions

We now show that a category with binary products extends (by making a choice) to an elementary category with binary products, and that the underlying category of an elementary category with binary products is a category with binary products.

context *category-with-binary-products*
begin

definition *some-pr1* (‹𝔭₁^?[-, -]›)
where *some-pr1 a b* ≡ *if ide a ∧ ide b then*

```
                fst (SOME x. has-as-binary-product a b (fst x) (snd x))
              else null

definition some-pr0 (‹𝔭₀^?[-, -]›)
where some-pr0 a b ≡ if ide a ∧ ide b then
                snd (SOME x. has-as-binary-product a b (fst x) (snd x))
              else null

lemma pr-yields-binary-product:
assumes ide a and ide b
shows has-as-binary-product a b 𝔭₁^?[a, b] 𝔭₀^?[a, b]
proof −
  have ∃x. has-as-binary-product a b (fst x) (snd x)
    using assms has-binary-products has-binary-products-def has-as-binary-product-def
    by simp
  thus ?thesis
    using assms has-binary-products has-binary-products-def some-pr0-def some-pr1-def
        someI-ex [of λx. has-as-binary-product a b (fst x) (snd x)]
    by simp
qed

interpretation elementary-category-with-binary-products C some-pr0 some-pr1
proof
  fix a b
  assume a: ide a and b: ide b
  interpret J: binary-product-shape .
  interpret D: binary-product-diagram C a b
    using a b by unfold-locales auto
  let ?χ = D.mkCone 𝔭₁^?[a, b] 𝔭₀^?[a, b]
  interpret χ: limit-cone J.comp C D.map ‹dom 𝔭₁^?[a, b]› ?χ
    using a b pr-yields-binary-product
    by (simp add: has-as-binary-product-def)
  have 1: 𝔭₁^?[a, b] = ?χ J.FF ∧ 𝔭₀^?[a, b] = ?χ J.TT
    using D.mkCone-def by simp
  show span 𝔭₁^?[a, b] 𝔭₀^?[a, b]
    using 1 χ.preserves-reflects-arr J.seqE J.arr-char J.seq-char J.is-category
        D.is-rendered-commutative-by-cone χ.cone-axioms
    by metis
  show cod 𝔭₁^?[a, b] = a
    using 1 χ.preserves-cod [of J.FF] J.cod-char J.arr-char by auto
  show cod 𝔭₀^?[a, b] = b
    using 1 χ.preserves-cod [of J.TT] J.cod-char J.arr-char by auto
  next
  fix f g
  assume fg: span f g
  show ∃!l. 𝔭₁^?[cod f, cod g] · l = f ∧ 𝔭₀^?[cod f, cod g] · l = g
  proof −
    interpret J: binary-product-shape .
    interpret D: binary-product-diagram C ‹cod f› ‹cod g›
```

```
      using fg by unfold-locales auto
    let ?χ = D.mkCone 𝔭₁?[cod f, cod g] 𝔭₀?[cod f, cod g]
    interpret χ: limit-cone J.comp C D.map ‹dom 𝔭₁?[cod f, cod g]› ?χ
      using fg pr-yields-binary-product [of cod f cod g] has-as-binary-product-def
      by simp
    interpret χ: binary-product-cone C ‹cod f› ‹cod g› ‹𝔭₁?[cod f, cod g]› ‹𝔭₀?[cod f, cod g]› ..
    have 1: 𝔭₁?[cod f, cod g] = ?χ J.FF ∧ 𝔭₀?[cod f, cod g] = ?χ J.TT
      using D.mkCone-def by simp
    show ∃!l. 𝔭₁?[cod f, cod g] · l = f ∧ 𝔭₀?[cod f, cod g] · l = g
    proof −
      have ∃!l. «l : dom f → dom 𝔭₁?[cod f, cod g]» ∧
                𝔭₁?[cod f, cod g] · l = f ∧ 𝔭₀?[cod f, cod g] · l = g
        using fg χ.is-universal′ by simp
      moreover have ⋀l. 𝔭₁?[cod f, cod g] · l = f ⟹ «l : dom f → dom 𝔭₁?[cod f, cod g]»
        using fg dom-comp in-homI seqE seqI by metis
      ultimately show ?thesis by auto
    qed
  qed
qed

proposition extends-to-elementary-category-with-binary-products:
shows elementary-category-with-binary-products C some-pr0 some-pr1
  ..

abbreviation some-prod    (infixr ‹⊗?› 51)
where some-prod ≡ prod

end

locale binary-product =
  category C
  for C :: 'a comp
  and a :: 'a
  and b :: 'a
  and p :: 'a
  and q :: 'a +
  assumes has-as-binary-product: has-as-binary-product a b p q
begin

  definition product
  where product ≡ dom p

  lemma ide-product [intro, simp]:
  shows ide product
    unfolding product-def
    using has-as-binary-product by fastforce

  lemma prj-in-hom [intro, simp]:
  shows «p : product → a»
```

and «q : $product \to b$»
using *has-as-binary-product product-def* **by** *auto*

lemma *prj-simps* [*simp*]:
shows $dom\ p = product$ **and** $cod\ p = a$ **and** $dom\ q = product$ **and** $cod\ q = b$
using *prj-in-hom* **by** *blast+*

definition *tuple*
where $tuple\ f\ g \equiv (THE\ h.\ C\ p\ h = f \wedge C\ q\ h = g)$

lemma *tuple-props*:
assumes *span f g* **and** $cod\ f = a$ **and** $cod\ g = b$
shows [*intro*, *simp*]: «$tuple\ f\ g : dom\ f \to product$»
and [*simp*]: $dom\ (tuple\ f\ g) = dom\ f$
and [*simp*]: $cod\ (tuple\ f\ g) = product$
and [*simp*]: $C\ p\ (tuple\ f\ g) = f$
and [*simp*]: $C\ q\ (tuple\ f\ g) = g$
and $\bigwedge h.\ [\![C\ p\ h = f;\ C\ q\ h = g]\!] \Longrightarrow h = tuple\ f\ g$
proof −
have *0*: $\exists!h.\ C\ p\ h = f \wedge C\ q\ h = g$
using *assms has-as-binary-product* **by** *blast*
hence ∗: $C\ p\ (tuple\ f\ g) = f \wedge C\ q\ (tuple\ f\ g) = g$
using *tuple-def theI'* [*of* $\lambda h.\ C\ p\ h = f \wedge C\ q\ h = g$] **by** *simp*
show *1*: «$tuple\ f\ g : dom\ f \to product$»
using *assms* ∗
by (*metis dom-comp in-homI prj-simps*(*3*) *seqE*)
show $dom\ (tuple\ f\ g) = dom\ f$
using *1* **by** *auto*
show $cod\ (tuple\ f\ g) = product$
using *1* **by** *auto*
show *2*: $C\ p\ (tuple\ f\ g) = f$
using ∗ **by** *auto*
show *3*: $C\ q\ (tuple\ f\ g) = g$
using ∗ **by** *auto*
show *4*: $\bigwedge h.\ [\![C\ p\ h = f;\ C\ q\ h = g]\!] \Longrightarrow h = tuple\ f\ g$
using ∗ *0 2 3* **by** *blast*
qed

lemma *tuple-proj*:
shows [*simp*]: $tuple\ p\ q = product$
by (*metis comp-arr-dom in-homE tuple-props*(*6*) *prj-in-hom*(*1*−*2*))

lemma *pr-joint-monic*:
assumes *seq p x* **and** *seq p y* **and** $C\ p\ x = C\ p\ y$ **and** $C\ q\ x = C\ q\ y$
shows $x = y$
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*,*3*−*4*) *cod-comp dom-comp tuple-props*(*6*)
product-def arrI prj-in-hom(*2*) *prj-simps*(*2*−*4*) *seqE seqI*)

end

```
lemma (in category) binary-products-are-isomorphic:
assumes has-as-binary-product a b p q and has-as-binary-product a b p' q'
shows isomorphic (dom p) (dom p')
and inverse-arrows (binary-product.tuple C p q p' q')
                   (binary-product.tuple C p' q' p q)
proof −
  show isomorphic (dom p) (dom p')
    using assms limits-are-isomorphic(1)
    unfolding has-as-binary-product-def by blast
  interpret pq: binary-product C a b p q
    using assms(1) by unfold-locales auto
  interpret p'q': binary-product C a b p' q'
    using assms(2) by unfold-locales auto
  show inverse-arrows (pq.tuple p' q') (p'q'.tuple p q)
  proof
    show ide (p'q'.tuple p q · pq.tuple p' q')
    proof −
      have p' · p'q'.tuple p q · pq.tuple p' q' = p' · p'q'.product ∧
           q' · p'q'.tuple p q · pq.tuple p' q' = q' · p'q'.product
        using pq.tuple-props [of p' q'] p'q'.tuple-props [of p q]
              comp-assoc
        by (metis arrI comp-arr-dom p'q'.prj-in-hom(1−2) p'q'.prj-simps(2−4)
            p'q'.product-def pq.prj-in-hom(1−2) pq.prj-simps(2−4) pq.product-def)
      thus ?thesis
        by (metis comp-arr-dom p'q'.ide-product p'q'.tuple-props(6)
            p'q'.prj-in-hom(1−2) p'q'.prj-simps(1−4) arrI)
    qed
    show ide (pq.tuple p' q' · p'q'.tuple p q)
    proof −
      have p · pq.tuple p' q' · p'q'.tuple p q = p · pq.product ∧
           q · pq.tuple p' q' · p'q'.tuple p q = q · pq.product
        using pq.tuple-props [of p' q'] p'q'.tuple-props [of p q]
              comp-assoc
          by (metis arrI comp-arr-dom p'q'.prj-in-hom(1−2) p'q'.prj-simps(2−4)
              p'q'.product-def pq.prj-in-hom(1−2) pq.prj-simps(2−4)
              pq.product-def)
      thus ?thesis
        by (metis arrI comp-arr-dom ide-char' pq.tuple-props(6) pq.prj-in-hom(1−2)
            pq.prj-simps(2−4) pq.product-def seqE)
    qed
  qed
qed

context elementary-category-with-binary-products
begin

  sublocale category-with-binary-products C
  proof
```

show *has-binary-products*
by (*meson has-as-binary-product has-binary-products-def*)
qed

proposition *is-category-with-binary-products*:
shows *category-with-binary-products C*
..

end

22.1.5 Further Properties

context *elementary-category-with-binary-products*
begin

lemma *interchange*:
assumes *seq h f* **and** *seq k g*
shows $(h \otimes k) \cdot (f \otimes g) = h \cdot f \otimes k \cdot g$
using *assms prod-def comp-tuple-arr comp-assoc* **by** *fastforce*

lemma *pr-naturality* [*simp*]:
assumes *arr g* **and** *dom g* = *b* **and** *cod g* = *d*
and *arr f* **and** *dom f* = *a* **and** *cod f* = *c*
shows $\mathfrak{p}_0[c, d] \cdot (f \otimes g) = g \cdot \mathfrak{p}_0[a, b]$
and $\mathfrak{p}_1[c, d] \cdot (f \otimes g) = f \cdot \mathfrak{p}_1[a, b]$
using *assms prod-def* **by** *fastforce+*

abbreviation *dup* (‹d[-]›)
where $\mathrm{d}[f] \equiv \langle f, f \rangle$

lemma *dup-in-hom* [*intro*, *simp*]:
assumes «$f : a \to b$»
shows «$\mathrm{d}[f] : a \to b \otimes b$»
using *assms* **by** *fastforce*

lemma *dup-simps* [*simp*]:
assumes *arr f*
shows *arr* $\mathrm{d}[f]$ **and** *dom* $\mathrm{d}[f]$ = *dom f* **and** *cod* $\mathrm{d}[f]$ = *cod f* $\otimes$ *cod f*
using *assms dup-in-hom* **by** *auto*

lemma *dup-naturality*:
assumes «$f : a \to b$»
shows $\mathrm{d}[b] \cdot f = (f \otimes f) \cdot \mathrm{d}[a]$
using *assms prod-def comp-arr-dom comp-cod-arr comp-tuple-arr comp-assoc*
by *fastforce*

lemma *pr-dup* [*simp*]:
assumes *ide a*
shows $\mathfrak{p}_0[a, a] \cdot \mathrm{d}[a] = a$ **and** $\mathfrak{p}_1[a, a] \cdot \mathrm{d}[a] = a$

using *assms* **by** *simp-all*

lemma *prod-tuple*:
assumes *span f g* **and** *seq h f* **and** *seq k g*
shows $(h \otimes k) \cdot \langle f, g \rangle = \langle h \cdot f, k \cdot g \rangle$
using *assms prod-def comp-assoc comp-tuple-arr* **by** *fastforce*

lemma *tuple-eqI*:
assumes *ide b* **and** *ide c* **and** *seq* $\mathfrak{p}_0[b, c]$ *f* **and** *seq* $\mathfrak{p}_1[b, c]$ *f*
and $\mathfrak{p}_0[b, c] \cdot f = f0$ **and** $\mathfrak{p}_1[b, c] \cdot f = f1$
shows $f = \langle f1, f0 \rangle$
using *assms pr-joint-monic* [*of b c* $\langle f1, f0 \rangle$ *f*] *pr-tuple* **by** *auto*

lemma *tuple-expansion*:
assumes *span f g*
shows $(f \otimes g) \cdot \mathrm{d}[dom\ f] = \langle f, g \rangle$
using *assms prod-tuple comp-arr-dom* **by** *simp*

definition *assoc* (‹a[-, -, -]›)
where $\mathrm{a}[a, b, c] \equiv \langle \mathfrak{p}_1[a, b] \cdot \mathfrak{p}_1[a \otimes b, c], \langle \mathfrak{p}_0[a, b] \cdot \mathfrak{p}_1[a \otimes b, c], \mathfrak{p}_0[a \otimes b, c] \rangle \rangle$

definition *assoc′* (‹a^{-1}[-, -, -]›)
where $\mathrm{a}^{-1}[a, b, c] \equiv \langle \langle \mathfrak{p}_1[a, b \otimes c], \mathfrak{p}_1[b, c] \cdot \mathfrak{p}_0[a, b \otimes c] \rangle, \mathfrak{p}_0[b, c] \cdot \mathfrak{p}_0[a, b \otimes c] \rangle$

lemma *assoc-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «$\mathrm{a}[a, b, c] : (a \otimes b) \otimes c \to a \otimes (b \otimes c)$»
using *assms assoc-def* **by** *auto*

lemma *assoc-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* $\mathrm{a}[a, b, c]$
and *dom* $\mathrm{a}[a, b, c] = (a \otimes b) \otimes c$
and *cod* $\mathrm{a}[a, b, c] = a \otimes (b \otimes c)$
using *assms assoc-in-hom* **by** *auto*

lemma *assoc′-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «$\mathrm{a}^{-1}[a, b, c] : a \otimes (b \otimes c) \to (a \otimes b) \otimes c$»
using *assms assoc′-def* **by** *auto*

lemma *assoc′-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* $\mathrm{a}^{-1}[a, b, c]$
and *dom* $\mathrm{a}^{-1}[a, b, c] = a \otimes (b \otimes c)$
and *cod* $\mathrm{a}^{-1}[a, b, c] = (a \otimes b) \otimes c$
using *assms assoc′-in-hom* **by** *auto*

lemma *pr-assoc*:

assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathfrak{p}_0[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] = \mathfrak{p}_0[a \otimes b,\ c]$
and $\mathfrak{p}_1[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] = \mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$
and $\mathfrak{p}_1[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] = \mathfrak{p}_1[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$
using *assms assoc-def* **by** *force+*

lemma *pr-assoc′*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathfrak{p}_1[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c] \cdot \mathrm{a}^{-1}[a,\ b,\ c] = \mathfrak{p}_1[a,\ b \otimes c]$
and $\mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c] \cdot \mathrm{a}^{-1}[a,\ b,\ c] = \mathfrak{p}_1[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c]$
and $\mathfrak{p}_0[a \otimes b,\ c] \cdot \mathrm{a}^{-1}[a,\ b,\ c] = \mathfrak{p}_0[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c]$
using *assms assoc′-def* **by** *force+*

lemma *assoc-naturality*:
assumes «*f0* : *a0* → *b0*» **and** «*f1* : *a1* → *b1*» **and** «*f2* : *a2* → *b2*»
shows $\mathrm{a}[b0,\ b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) = (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[a0,\ a1,\ a2]$
proof −
have $\mathfrak{p}_0[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) =$
$\mathfrak{p}_0[b0,\ b1 \otimes b2] \cdot (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[a0,\ a1,\ a2]$
proof −
have $\mathfrak{p}_0[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) =$
$(\mathfrak{p}_0[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2]) \cdot ((f0 \otimes f1) \otimes f2)$
using *comp-assoc* **by** *simp*
also have ... $= \langle \mathfrak{p}_0[b0,\ b1] \cdot \mathfrak{p}_1[b0 \otimes b1,\ b2],\ \mathfrak{p}_0[b0 \otimes b1,\ b2] \rangle \cdot ((f0 \otimes f1) \otimes f2)$
using *assms assoc-def* **by** *fastforce*
also have ... $= \langle (\mathfrak{p}_0[b0,\ b1] \cdot \mathfrak{p}_1[b0 \otimes b1,\ b2]) \cdot ((f0 \otimes f1) \otimes f2),$
$\mathfrak{p}_0[b0 \otimes b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) \rangle$
using *assms comp-tuple-arr* **by** *fastforce*
also have ... $= \langle (\mathfrak{p}_0[b0,\ b1] \cdot (f0 \otimes f1)) \cdot \mathfrak{p}_1[a0 \otimes a1,\ a2],\ f2 \cdot \mathfrak{p}_0[a0 \otimes a1,\ a2] \rangle$
using *assms comp-assoc* **by** *fastforce*
also have ... $= \langle f1 \cdot \mathfrak{p}_0[a0,\ a1] \cdot \mathfrak{p}_1[a0 \otimes a1,\ a2],\ f2 \cdot \mathfrak{p}_0[a0 \otimes a1,\ a2] \rangle$
using *assms comp-assoc*
by (*metis in-homE pr-naturality*(*1*))
also have ... $= \mathfrak{p}_0[b0,\ b1 \otimes b2] \cdot (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[a0,\ a1,\ a2]$
using *assms comp-assoc assoc-def prod-tuple* **by** *fastforce*
finally show *?thesis* **by** *blast*
qed
moreover have $\mathfrak{p}_1[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) =$
$\mathfrak{p}_1[b0,\ b1 \otimes b2] \cdot (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[a0,\ a1,\ a2]$
proof −
have $\mathfrak{p}_1[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2] \cdot ((f0 \otimes f1) \otimes f2) =$
$(\mathfrak{p}_1[b0,\ b1 \otimes b2] \cdot \mathrm{a}[b0,\ b1,\ b2]) \cdot ((f0 \otimes f1) \otimes f2)$
using *comp-assoc* **by** *simp*
also have ... $= (\mathfrak{p}_1[b0,\ b1] \cdot \mathfrak{p}_1[b0 \otimes b1,\ b2]) \cdot ((f0 \otimes f1) \otimes f2)$
using *assms assoc-def* **by** *fastforce*
also have ... $= (\mathfrak{p}_1[b0,\ b1] \cdot (f0 \otimes f1)) \cdot \mathfrak{p}_1[a0 \otimes a1,\ a2]$
using *assms comp-assoc* **by** *fastforce*
also have ... $= f0 \cdot \mathfrak{p}_1[a0,\ a1] \cdot \mathfrak{p}_1[a0 \otimes a1,\ a2]$
using *assms comp-assoc*

```
    by (metis in-homE pr-naturality(2))
  also have ... = 𝔭₁[b0, b1 ⊗ b2] · (f0 ⊗ (f1 ⊗ f2)) · a[a0, a1, a2]
  proof −
    have 𝔭₁[b0, b1 ⊗ b2] · (f0 ⊗ (f1 ⊗ f2)) · a[a0, a1, a2] =
         (𝔭₁[b0, b1 ⊗ b2] · (f0 ⊗ (f1 ⊗ f2))) · a[a0, a1, a2]
      using comp-assoc by simp
    also have ... = f0 · 𝔭₁[a0, a1 ⊗ a2] · a[a0, a1, a2]
      using assms comp-assoc by fastforce
    also have ... = f0 · 𝔭₁[a0, a1] · 𝔭₁[a0 ⊗ a1, a2]
      using assms assoc-def by fastforce
    finally show ?thesis by simp
  qed
  finally show ?thesis by blast
 qed
 ultimately show ?thesis
  using assms pr-joint-monic [of b0 b1 ⊗ b2 a[b0, b1, b2] · ((f0 ⊗ f1) ⊗ f2)
                                 (f0 ⊗ (f1 ⊗ f2)) · a[a0, a1, a2]]
  by fastforce
qed

lemma pentagon:
assumes ide a and ide b and ide c and ide d
shows ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) = a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
proof (intro pr-joint-monic
            [where h = ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d)
              and h' = a[a, b, c ⊗ d] · a[a ⊗ b, c, d]])
 show ide a by fact
 show ide (b ⊗ (c ⊗ d))
  by (simp add: assms(2−4))
 show seq 𝔭₀[a, b ⊗ (c ⊗ d)] (((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d))
  using assms by simp
 show 𝔭₁[a, b ⊗ c ⊗ d] · ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) =
      𝔭₁[a, b ⊗ c ⊗ d] · a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
 proof −
  have 𝔭₁[a, b ⊗ c ⊗ d] · ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) =
       ((𝔭₁[a, b ⊗ c ⊗ d] · (a ⊗ a[b, c, d])) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d)
    using comp-assoc by simp
  also have ... = (𝔭₁[a, (b ⊗ c) ⊗ d] · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d)
    using assms pr-naturality(2) comp-cod-arr by force
  also have ... = 𝔭₁[a, b ⊗ c] · 𝔭₁[a ⊗ b ⊗ c, d] · (a[a, b, c] ⊗ d)
    using assms assoc-def comp-assoc by simp
  also have ... = (𝔭₁[a, b ⊗ c] · a[a, b, c]) · 𝔭₁[(a ⊗ b) ⊗ c, d]
    using assms pr-naturality(2) comp-assoc by fastforce
  also have ... = 𝔭₁[a, b] · 𝔭₁[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d]
    using assms assoc-def comp-assoc by simp
  finally have 𝔭₁[a, b ⊗ c ⊗ d] · ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) =
               𝔭₁[a, b] · 𝔭₁[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d]
    by blast
  also have ... = 𝔭₁[a, b ⊗ c ⊗ d] · a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
```

```
    using assms assoc-def comp-assoc by auto
  finally show ?thesis by blast
qed
show 𝔭₀[a, b ⊗ (c ⊗ d)] · ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) =
     𝔭₀[a, b ⊗ (c ⊗ d)] · a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
proof −
  have 𝔭₀[a, b ⊗ (c ⊗ d)] · ((a ⊗ a[b, c, d]) · a[a, b ⊗ c, d]) · (a[a, b, c] ⊗ d) =
       𝔭₀[a, b ⊗ c ⊗ d] ·
         ((a ⊗ ⟨𝔭₁[b, c] · 𝔭₁[b ⊗ c, d], ⟨𝔭₀[b, c] · 𝔭₁[b ⊗ c, d], 𝔭₀[b ⊗ c, d]⟩⟩) ·
          ⟨𝔭₁[a, b ⊗ c] · 𝔭₁[a ⊗ b ⊗ c, d],
           ⟨𝔭₀[a, b ⊗ c] · 𝔭₁[a ⊗ b ⊗ c, d], 𝔭₀[a ⊗ b ⊗ c, d]⟩⟩) ·
         (⟨𝔭₁[a, b] · 𝔭₁[a ⊗ b, c], ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩⟩ ⊗ d)
    using assms assoc-def by simp
  also have ... = ⟨𝔭₁[b, c] · 𝔭₁[b ⊗ c, d],
                   ⟨𝔭₀[b, c] · 𝔭₁[b ⊗ c, d], 𝔭₀[b ⊗ c, d]⟩⟩ · (𝔭₀[a, (b ⊗ c) ⊗ d] ·
                    ⟨𝔭₁[a, b ⊗ c] · 𝔭₁[a ⊗ b ⊗ c, d],
                     ⟨𝔭₀[a, b ⊗ c] · 𝔭₁[a ⊗ b ⊗ c, d], 𝔭₀[a ⊗ b ⊗ c, d]⟩⟩) ·
                    (⟨𝔭₁[a, b] · 𝔭₁[a ⊗ b, c],
                      ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩⟩ ⊗ d)
  proof −
    have 𝔭₀[a, b ⊗ c ⊗ d] ·
           (a ⊗ ⟨𝔭₁[b, c] · 𝔭₁[b ⊗ c, d], ⟨𝔭₀[b, c] · 𝔭₁[b ⊗ c, d], 𝔭₀[b ⊗ c, d]⟩⟩) =
         ⟨𝔭₁[b, c] · 𝔭₁[b ⊗ c, d], ⟨𝔭₀[b, c] · 𝔭₁[b ⊗ c, d], 𝔭₀[b ⊗ c, d]⟩⟩ ·
          𝔭₀[a, (b ⊗ c) ⊗ d]
      using assms assoc-def ide-in-hom pr-naturality(1) by auto
    thus ?thesis using comp-assoc by metis
  qed
  also have ... = ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d],
                   ⟨𝔭₀[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d], d · 𝔭₀[(a ⊗ b) ⊗ c, d]⟩⟩
    using assms comp-assoc by simp
  also have ... = ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d],
                   ⟨𝔭₀[a ⊗ b, c] · 𝔭₁[(a ⊗ b) ⊗ c, d], 𝔭₀[(a ⊗ b) ⊗ c, d]⟩⟩
    using assms comp-cod-arr by simp
  also have ... = 𝔭₀[a, b ⊗ (c ⊗ d)] · a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
    using assms assoc-def comp-assoc by simp
  finally show ?thesis by simp
qed
qed

lemma inverse-arrows-assoc:
assumes ide a and ide b and ide c
shows inverse-arrows a[a, b, c] a⁻¹[a, b, c]
  using assms assoc-def assoc'-def comp-assoc
  by (auto simp add: tuple-pr-arr)

lemma inv-prod:
assumes iso f and iso g
shows iso (prod f g)
and inv (prod f g) = prod (inv f) (inv g)
```

proof −
  have *1*: *inverse-arrows* (*prod f g*) (*prod* (*inv f*) (*inv g*))
    by (*auto simp add*: *assms comp-inv-arr′ comp-arr-inv′ iso-is-arr interchange*)
  show *iso* (*prod f g*)
    using *1* **by** *blast*
  show *inv* (*prod f g*) = *prod* (*inv f*) (*inv g*)
    using *1* **by** (*simp add*: *inverse-unique*)
qed

interpretation *CC*: *product-category C C* **..**

abbreviation *Prod*
where *Prod fg* ≡ *fst fg* ⊗ *snd fg*
abbreviation *Prod′*
where *Prod′ fg* ≡ *snd fg* ⊗ *fst fg*

interpretation Π: *binary-functor C C C Prod*
  using *tuple-ext CC.comp-char interchange*
  apply *unfold-locales*
    apply *auto*
  by (*metis prod-def seqE*)+

interpretation *Prod′*: *binary-functor C C C Prod′*
  using *tuple-ext CC.comp-char interchange*
  apply *unfold-locales*
    apply *auto*
  by (*metis prod-def seqE*)+

lemma *binary-functor-Prod*:
shows *binary-functor C C C Prod* **and** *binary-functor C C C Prod′*
  ..

interpretation *CCC*: *product-category C CC.comp* **..**
interpretation *T*: *binary-endofunctor C Prod* **..**
interpretation *ToTC*: *functor CCC.comp C T.ToTC*
  using *T.functor-ToTC* **by** *auto*
interpretation *ToCT*: *functor CCC.comp C T.ToCT*
  using *T.functor-ToCT* **by** *auto*

abbreviation α
where α *f* ≡ a[*cod* (*fst f*), *cod* (*fst* (*snd f*)), *cod* (*snd* (*snd f*))] ·
                ((*fst f* ⊗ *fst* (*snd f*)) ⊗ *snd* (*snd f*))

lemma α*-simp-ide*:
assumes *CCC.ide a*
shows α *a* = a[*fst a*, *fst* (*snd a*), *snd* (*snd a*)]
  using *assms comp-arr-dom* **by** *auto*

interpretation α: *natural-isomorphism CCC.comp C T.ToTC T.ToCT* α

proof
  fix *f*
  show ¬ *CCC.arr f* ⟹ *α f* = *null*
    by (*metis CC.arr-char CCC.arr-char ext prod-def seqE tuple-ext*)
  assume *f*: *CCC.arr f*
  show *arr* (*α f*)
    using *f* **by** *auto*
  show *T.ToCT f* ⋅ *α* (*CCC.dom f*) = *α f*
    using *f T.ToCT-def T.ToTC-def comp-assoc*
        assoc-naturality
          [*of fst f dom* (*fst f*) *cod* (*fst f*)
              fst (*snd f*) *dom* (*fst* (*snd f*)) *cod* (*fst* (*snd f*))
              snd (*snd f*) *dom* (*snd* (*snd f*)) *cod* (*snd* (*snd f*))]
    by (*simp add*: *comp-arr-dom in-homI*)
  show *α* (*CCC.cod f*) ⋅ *T.ToTC f* = *α f*
    using *T.ToTC-def comp-arr-dom f* **by** *auto*
  next
  show ⋀*a*. *CCC.ide a* ⟹ *iso* (*α a*)
    using *CCC.ide-char*$_{PC}$ *CC.ide-char*$_{PC}$ *comp-arr-dom inverse-arrows-assoc isoI*
    by (*metis assoc-simps*(*1*−*2*) *ideD*(*3*))
qed

lemma *α-naturalityisomorphism*:
shows *natural-isomorphism CCC.comp C T.ToTC T.ToCT α*
  ..

definition *sym* (‹s[-, -]›)
where s[*a1*, *a0*] ≡ *if ide a0* ∧ *ide a1 then* ⟨𝔭$_0$[*a1*, *a0*], 𝔭$_1$[*a1*, *a0*]⟩ *else null*

lemma *sym-in-hom* [*intro*]:
assumes *ide a* **and** *ide b*
shows «s[*a*, *b*] : *a* ⊗ *b* → *b* ⊗ *a*»
  using *assms sym-def* **by** *auto*

lemma *sym-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* s[*a*, *b*] **and** *dom* s[*a*, *b*] = *a* ⊗ *b* **and** *cod* s[*a*, *b*] = *b* ⊗ *a*
  using *assms sym-in-hom* **by** *auto*

lemma *comp-sym-tuple* [*simp*]:
assumes «*f0* : *a* → *b0*» **and** «*f1* : *a* → *b1*»
shows s[*b0*, *b1*] ⋅ ⟨*f0*, *f1*⟩ = ⟨*f1*, *f0*⟩
  using *assms sym-def comp-tuple-arr* **by** *fastforce*

lemma *prj-sym* [*simp*]:
assumes *ide a0* **and** *ide a1*
shows 𝔭$_0$[*a1*, *a0*] ⋅ s[*a0*, *a1*] = 𝔭$_1$[*a0*, *a1*]
and 𝔭$_1$[*a1*, *a0*] ⋅ s[*a0*, *a1*] = 𝔭$_0$[*a0*, *a1*]
  using *assms sym-def* **by** *auto*

lemma *comp-sym-sym* [*simp*]:
assumes *ide a0* **and** *ide a1*
shows s[*a1*, *a0*] · s[*a0*, *a1*] = (*a0* ⊗ *a1*)
using *assms sym-def comp-tuple-arr* **by** *auto*

lemma *sym-inverse-arrows*:
assumes *ide a0* **and** *ide a1*
shows *inverse-arrows* s[*a0*, *a1*] s[*a1*, *a0*]
using *assms sym-in-hom comp-sym-sym* **by** *auto*

lemma *sym-assoc-coherence*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows a[*b*, *c*, *a*] · s[*a*, *b* ⊗ *c*] · a[*a*, *b*, *c*] = (*b* ⊗ s[*a*, *c*]) · a[*b*, *a*, *c*] · (s[*a*, *b*] ⊗ *c*)
using *assms sym-def assoc-def comp-assoc prod-tuple comp-cod-arr* **by** *simp*

lemma *sym-naturality*:
assumes «*f0* : *a0* → *b0*» **and** «*f1* : *a1* → *b1*»
shows s[*b0*, *b1*] · (*f0* ⊗ *f1*) = (*f1* ⊗ *f0*) · s[*a0*, *a1*]
using *assms sym-def comp-assoc prod-tuple* **by** *fastforce*

abbreviation σ
where σ *fg* ≡ s[*cod* (*fst fg*), *cod* (*snd fg*)] · (*fst fg* ⊗ *snd fg*)

interpretation σ: *natural-transformation CC.comp C Prod Prod'* σ
using *sym-def CC.arr-char CC.null-char comp-arr-dom comp-cod-arr*
apply *unfold-locales*
apply *auto*
using *arr-cod-iff-arr ideD*(*1*)
apply *metis*
using *arr-cod-iff-arr ideD*(*1*)
apply *metis*
using *prod-tuple* **by** *simp*

lemma σ*-naturalitytransformation*:
shows *natural-transformation CC.comp C Prod Prod'* σ
..

abbreviation *Diag*
where *Diag f* ≡ *if arr f then* (*f*, *f*) *else CC.null*

interpretation Δ: *functor C CC.comp Diag*
by (*unfold-locales*, *auto*)

lemma *functor-Diag*:
shows *functor C CC.comp Diag*
..

interpretation Δ*o*Π: *composite-functor CC.comp C CC.comp Prod Diag* **..**

interpretation $\Pi o \Delta$: *composite-functor C CC.comp C Diag Prod* **..**

abbreviation π
where $\pi \equiv \lambda(f,\ g).\ (\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot (f \otimes g),\ \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot (f \otimes g))$

interpretation π: *transformation-by-components CC.comp CC.comp* $\Delta o \Pi$*.map CC.map* π
  using *pr-naturality comp-arr-dom comp-cod-arr*
  by *unfold-locales auto*

lemma π*-naturalitytransformation*:
shows *natural-transformation CC.comp CC.comp* $\Delta o \Pi$*.map CC.map* π
proof −
  have π*.map* = π
    using π*.map-def ext* Π*.extensionality comp-arr-dom comp-cod-arr* **by** *auto*
  thus *natural-transformation CC.comp CC.comp* $\Delta o \Pi$*.map CC.map* π
    using π*.natural-transformation-axioms* **by** *simp*
qed

interpretation δ: *natural-transformation C C map* $\Pi o \Delta$*.map dup*
  using *dup-naturality comp-arr-dom comp-cod-arr prod-tuple tuple-ext*
  by *unfold-locales auto*

lemma *dup-naturalitytransformation*:
shows *natural-transformation C C map* $\Pi o \Delta$*.map dup*
  ..

interpretation $\Delta o \Pi o \Delta$: *composite-functor C CC.comp CC.comp Diag* $\Delta o \Pi$*.map* **..**
interpretation $\Pi o \Delta o \Pi$: *composite-functor CC.comp C C Prod* $\Pi o \Delta$*.map* **..**

interpretation $\Delta o \delta$: *natural-transformation C CC.comp Diag* $\Delta o \Pi o \Delta$*.map* ‹*Diag* ∘ *dup*›
proof −
  have *Diag* ∘ *map* = *Diag*
    by *auto*
  thus *natural-transformation C CC.comp Diag* $\Delta o \Pi o \Delta$*.map* (*Diag* ∘ *dup*)
    using Δ*.as-nat-trans.natural-transformation-axioms* δ*.natural-transformation-axioms*
        o-assoc horizontal-composite [*of C C map* $\Pi o \Delta$*.map dup CC.comp Diag Diag Diag*]
    by *metis*
qed

interpretation $\delta o \Pi$: *natural-transformation CC.comp C Prod* $\Pi o \Delta o \Pi$*.map* ‹*dup* ∘ *Prod*›
  using δ*.natural-transformation-axioms* Π*.as-nat-trans.natural-transformation-axioms*
      o-assoc horizontal-composite [*of CC.comp C Prod Prod Prod C map* $\Pi o \Delta$*.map dup*]
  by *simp*

interpretation $\pi o \Delta$: *natural-transformation C CC.comp* $\Delta o \Pi o \Delta$*.map Diag* ‹π*.map* ∘ *Diag*›
  using π*.natural-transformation-axioms* Δ*.as-nat-trans.natural-transformation-axioms*
      horizontal-composite
        [*of C CC.comp Diag Diag Diag CC.comp* $\Delta o \Pi$*.map CC.map* π*.map*]
  by *simp*

```
interpretation Πoπ: natural-transformation CC.comp C ΠoΔoΠ.map Prod ‹Prod ∘ π.map›
proof −
  have Prod ∘ ΔoΠ.map = ΠoΔoΠ.map
    by auto
  thus natural-transformation CC.comp C ΠoΔoΠ.map Prod (Prod ∘ π.map)
    using π.natural-transformation-axioms Π.as-nat-trans.natural-transformation-axioms
          o-assoc
          horizontal-composite
            [of CC.comp CC.comp ΔoΠ.map CC.map π.map C Prod Prod Prod]
    by simp
qed

interpretation Δoδ-πoΔ: vertical-composite C CC.comp Diag ΔoΠoΔ.map Diag
                        ‹Diag ∘ dup› ‹π.map ∘ Diag›
  ..
interpretation Πoπ-δoΠ: vertical-composite CC.comp C Prod ΠoΔoΠ.map Prod
                        ‹dup ∘ Prod› ‹Prod ∘ π.map›
  ..

interpretation ΔΠ: unit-counit-adjunction CC.comp C Diag Prod dup π.map
proof
  show Δoδ-πoΔ.map = Diag
  proof
    fix f
    have ¬ arr f ⟹ Δoδ-πoΔ.map f = Diag f
      by (simp add: Δoδ-πoΔ.extensionality )
    moreover have arr f ⟹ Δoδ-πoΔ.map f = Diag f
      using comp-cod-arr comp-assoc Δoδ-πoΔ.map-def by auto
    ultimately show Δoδ-πoΔ.map f = Diag f by blast
  qed
  show Πoπ-δoΠ.map = Prod
  proof
    fix fg
    show Πoπ-δoΠ.map fg = Prod fg
    proof −
      have ¬ CC.arr fg ⟹ ?thesis
        by (simp add: Π.extensionality  Πoπ-δoΠ.extensionality )
      moreover have CC.arr fg ⟹ ?thesis
      proof −
        assume fg: CC.arr fg
        have 1: dup (Prod fg) = ⟨cod (fst fg) ⊗ cod (snd fg), cod (fst fg) ⊗ cod (snd fg)⟩ ·
                                (fst fg ⊗ snd fg)
          using fg δ.naturality2
          apply simp
          by (metis (no-types, lifting) prod-simps(1) prod-simps(3))
        have Πoπ-δoΠ.map fg =
              (𝔭₁[cod (fst fg), cod (snd fg)] ⊗ 𝔭₀[cod (fst fg), cod (snd fg)]) ·
                ⟨cod (fst fg) ⊗ cod (snd fg), cod (fst fg) ⊗ cod (snd fg)⟩ ·
```

```
            (fst fg ⊗ snd fg)
        using fg 1 Πoπ-δoΠ.map-def comp-cod-arr by simp
      also have ... = ((𝔭₁[cod (fst fg), cod (snd fg)] ⊗ 𝔭₀[cod (fst fg), cod (snd fg)]) ·
                       ⟨cod (fst fg) ⊗ cod (snd fg), cod (fst fg) ⊗ cod (snd fg)⟩) ·
                      (fst fg ⊗ snd fg)
        using comp-assoc by simp
      also have ... = ⟨𝔭₁[cod (fst fg), cod (snd fg)] · (cod (fst fg) ⊗ cod (snd fg)),
                       𝔭₀[cod (fst fg), cod (snd fg)] · (cod (fst fg) ⊗ cod (snd fg))⟩ ·
                      (fst fg ⊗ snd fg)
        using fg prod-tuple by simp
      also have ... = Prod fg
        using fg comp-arr-dom Π.as-nat-trans.naturality2 by auto
      finally show ?thesis by simp
    qed
    ultimately show ?thesis by blast
  qed
 qed
 qed

 proposition induces-unit-counit-adjunction:
 shows unit-counit-adjunction CC.comp C Diag Prod dup π.map
  using ΔΠ.unit-counit-adjunction-axioms by simp

end
```

22.2 Category with Terminal Object

```
locale category-with-terminal-object =
  category +
assumes has-terminal: ∃ t. terminal t

locale elementary-category-with-terminal-object =
  category C
for C :: 'a comp                                (infixr ‹·› 55)
and one :: 'a                                   (‹1›)
and trm :: 'a ⇒ 'a                              (‹t[-]›) +
assumes ide-one: ide 1
and trm-in-hom [intro, simp]: ide a ⟹ «t[a] : a → 1»
and trm-eqI: ⟦ ide a; «f : a → 1» ⟧ ⟹ f = t[a]
begin

  lemma trm-simps [simp]:
  assumes ide a
  shows arr t[a] and dom t[a] = a and cod t[a] = 1
    using assms trm-in-hom by blast+

  lemma trm-one:
  shows t[1] = 1
  using ide-one trm-eqI ide-in-hom by auto
```

lemma *terminal-one*:
shows *terminal* **1**
using *ide-one trm-in-hom trm-eqI terminal-def* **by** *metis*

lemma *trm-naturality*:
assumes *arr f*
shows $\mathrm{t}[cod\ f] \cdot f = \mathrm{t}[dom\ f]$
using *assms trm-eqI*
by (*metis comp-in-homI' ide-cod ide-dom in-homE trm-in-hom*)

sublocale *category-with-terminal-object C*
apply *unfold-locales*
using *terminal-one* **by** *auto*

proposition *is-category-with-terminal-object*:
shows *category-with-terminal-object C*
..

definition τ
where $\tau = (\lambda f.\ if\ arr\ f\ then\ trm\ (dom\ f)\ else\ null)$

lemma τ*-in-hom* [*intro*, *simp*]:
assumes *arr f*
shows «$\tau\ f : dom\ f \to \mathbf{1}$»
by (*simp add*: τ*-def assms*)

lemma τ*-simps* [*simp*]:
assumes *arr f*
shows *arr* $(\tau\ f)$ **and** *dom* $(\tau\ f) = dom\ f$ **and** *cod* $(\tau\ f) = \mathbf{1}$
using *assms* **by** (*auto simp add*: τ*-def assms*)

sublocale Ω: *constant-functor C C* **1**
using *ide-one*
by *unfold-locales auto*

sublocale τ: *natural-transformation C C map* Ω*.map* τ
unfolding τ*-def*
using *trm-simps*(*1*−*3*) *comp-cod-arr trm-naturality*
by *unfold-locales auto*

end

context *category-with-terminal-object*
begin

definition *some-terminal* (‹$\mathbf{1}^?$›)
where *some-terminal* ≡ *SOME t. terminal t*

definition *some-terminator* (‹$\mathrm{t}^?[\text{-}]$›)
where $\mathrm{t}^?[f] \equiv$ *if arr f then THE t.* «t : *dom f* → $\mathbf{1}^?$» *else null*

lemma *terminal-some-terminal* [*intro*]:
shows *terminal* $\mathbf{1}^?$
  using *some-terminal-def has-terminal someI-ex* [*of* λ*t. terminal t*] **by** *presburger*

lemma *ide-some-terminal*:
shows *ide* $\mathbf{1}^?$
  using *terminal-def* **by** *blast*

lemma *some-trm-in-hom* [*intro*]:
assumes *arr f*
shows «$\mathrm{t}^?[f]$: *dom f* → $\mathbf{1}^?$»
proof −
  have *ide* (*dom f*) **using** *assms* **by** *fastforce*
  hence ∃!*t.* «t : *dom f* → $\mathbf{1}^?$»
    using *assms some-terminator-def terminal-def terminal-some-terminal* **by** *simp*
  thus *?thesis*
    using *assms some-terminator-def* [*of f*] *theI'* [*of* λ*t.* «t : *dom f* → $\mathbf{1}^?$»] **by** *auto*
qed

lemma *some-trm-simps* [*simp*]:
assumes *arr f*
shows *arr* $\mathrm{t}^?[f]$ **and** *dom* $\mathrm{t}^?[f]$ = *dom f* **and** *cod* $\mathrm{t}^?[f] = \mathbf{1}^?$
  using *assms some-trm-in-hom* **by** *auto*

lemma *some-trm-eqI*:
assumes «t : *dom f* → $\mathbf{1}^?$»
shows $t = \mathrm{t}^?[f]$
proof −
  have *ide* (*dom f*) **using** *assms*
    by (*metis ide-dom in-homE*)
  hence ∃!*t.* «t : *dom f* → $\mathbf{1}^?$»
    using *terminal-def* [*of* $\mathbf{1}^?$] *terminal-some-terminal* **by** *auto*
  moreover have «t : *dom f* → $\mathbf{1}^?$»
    using *assms* **by** *simp*
  ultimately show *?thesis*
    using *assms some-terminator-def the1-equality* [*of* λ*t.* «t : *dom f* → $\mathbf{1}^?$» t]
        ‹*ide* (*dom f*)› *arr-dom-iff-arr*
    by *fastforce*
qed

proposition *extends-to-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object C* $\mathbf{1}^?$ (λ*a.* $\mathrm{t}^?[a]$)
  using *ide-some-terminal some-trm-eqI*
  by *unfold-locales auto*

end

lemma (**in** *category-with-terminal-object*) *binary-product-is-pullback*:
assumes *has-as-binary-product a b p q*
shows *has-as-pullback* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *p q*
proof
  interpret *elementary-category-with-terminal-object C* ‹$\mathbf{1}^?$› ‹$\lambda a.\ \mathrm{t}^?[a]$›
    using *extends-to-elementary-category-with-terminal-object* **by** *blast*
  show *cospan* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$
    using *assms* **by** *fastforce*
  show *commutative-square* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *p q*
    using *assms trm-naturality* **by** *fastforce*
  show $\bigwedge h\ k.$ *commutative-square* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *h k* $\Longrightarrow \exists!l.\ p \cdot l = h \wedge q \cdot l = k$
  proof −
    fix *h k*
    assume *1*: *commutative-square* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *h k*
    have «*h* : *dom h* → *a*» ∧ «*k* : *dom h* → *b*»
      by (*metis 1* ‹*commutative-square* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *p q*› *arr-iff-in-hom assms*
          commutative-squareE has-as-binary-productE in-homE)
    thus $\exists!l.\ p \cdot l = h \wedge q \cdot l = k$
      using *assms has-as-binary-productE* [*of a b p q*] **by** *metis*
  qed
qed

22.3 Cartesian Category

locale *cartesian-category* =
  category-with-binary-products +
  category-with-terminal-object

locale *category-with-pullbacks-and-terminal-object* =
  category-with-pullbacks +
  category-with-terminal-object
begin

  sublocale *category-with-binary-products C*
  proof
    interpret *elementary-category-with-terminal-object C* ‹$\mathbf{1}^?$› ‹$\lambda a.\ \mathrm{t}^?[a]$›
      using *extends-to-elementary-category-with-terminal-object* **by** *blast*
    show *has-binary-products*
    proof −
      have $\bigwedge a0\ a1.$ ⟦*ide a0*; *ide a1*⟧ $\Longrightarrow \exists p0\ p1.$ *has-as-binary-product a0 a1 p0 p1*
      proof −
        fix *a0 a1*
        assume *a0*: *ide a0* **and** *a1*: *ide a1*
        obtain *p0 p1* **where** *p0p1*: *has-as-pullback* $\mathrm{t}^?[a0]$ $\mathrm{t}^?[a1]$ *p0 p1*
          using *a0 a1 has-pullbacks has-pullbacks-def* **by** *force*
        have *has-as-binary-product a0 a1 p0 p1*
        proof
          show *span p0 p1*

```
        using p0p1 by blast
      show cod p0 = a0 and cod p1 = a1
        using p0p1 a0 a1 commutative-squareE has-as-pullbackE in-homI trm-simps(2)
        by metis+
      fix x f g
      assume f: «f : x → a0» and g: «g : x → a1»
      have commutative-square t?[a0] t?[a1] f g
        by (metis a0 has-as-pullbackE in-homE commutative-squareI f g p0p1 trm-naturality
            trm-simps(2))
      moreover have ⋀l. p0 · l = f ∧ p1 · l = g ⟹ «l : x → dom p0»
        using f g by blast
      ultimately show ∃!l. «l : x → dom p0» ∧ p0 · l = f ∧ p1 · l = g
        by (metis has-as-pullbackE p0p1)
    qed
    thus ∃p0 p1. has-as-binary-product a0 a1 p0 p1
      by auto
  qed
  thus ?thesis
    using has-binary-products-def by force
 qed
qed

 sublocale cartesian-category C ..

end

locale elementary-cartesian-category =
 elementary-category-with-binary-products +
 elementary-category-with-terminal-object
begin

 sublocale cartesian-category C
   using cartesian-category.intro is-category-with-binary-products
         is-category-with-terminal-object
   by auto

 proposition is-cartesian-category:
 shows cartesian-category C
   ..

end

context cartesian-category
begin

 proposition extends-to-elementary-cartesian-category:
 shows elementary-cartesian-category C some-pr0 some-pr1 1? (λa. t?[a])
   by (simp add: elementary-cartesian-category-def
       extends-to-elementary-category-with-terminal-object
```

extends-to-elementary-category-with-binary-products)

end

22.3.1 Monoidal Structure

Here we prove some facts that will later allow us to show that an elementary cartesian category is a monoidal category.

context *elementary-cartesian-category*
begin

abbreviation ι
where $\iota \equiv \mathfrak{p}_0[\mathbf{1}, \mathbf{1}]$

lemma *pr-coincidence*:
shows $\iota = \mathfrak{p}_1[\mathbf{1}, \mathbf{1}]$
using *ide-one*
by (*simp add*: *terminal-arr-unique terminal-one*)

lemma *unit-is-terminal-arr*:
shows *terminal-arr* ι
using *ide-one*
by (*simp add*: *terminal-one*)

lemma *unit-eq-trm*:
shows $\iota = \mathrm{t}[\mathbf{1} \otimes \mathbf{1}]$
by (*metis unit-is-terminal-arr cod-pr1 comp-cod-arr pr-simps*(*5*) *trm-naturality ide-one pr-coincidence trm-one*)

lemma *inverse-arrows-ι*:
shows *inverse-arrows* ι $\langle \mathbf{1}, \mathbf{1} \rangle$
using *ide-one*
by (*metis unit-is-terminal-arr cod-pr0 comp-tuple-arr ide-char ide-dom inverse-arrows-def prod-def pr-coincidence pr-dup*(*2*) *pr-simps*(*2*))

lemma *ι-is-iso*:
shows *iso* ι
using *inverse-arrows-ι* **by** *auto*

lemma *trm-tensor*:
assumes *ide a* **and** *ide b*
shows $\mathrm{t}[a \otimes b] = \iota \cdot (\mathrm{t}[a] \otimes \mathrm{t}[b])$
using *assms unit-is-terminal-arr terminal-arr-unique ide-one*
by (*simp add*: *unit-eq-trm*)

abbreviation *runit* ($\langle \mathrm{r}[\text{-}] \rangle$)
where $\mathrm{r}[a] \equiv \mathfrak{p}_1[a, \mathbf{1}]$

abbreviation *runit'* ($\langle \mathrm{r}^{-1}[\text{-}] \rangle$)

where $\mathrm{r}^{-1}[a] \equiv \langle a,\ \mathrm{t}[a]\rangle$

abbreviation *lunit* ($\langle \mathrm{l}[\text{-}]\rangle$)
where $\mathrm{l}[a] \equiv \mathfrak{p}_0[\mathbf{1},\ a]$

abbreviation *lunit'* ($\langle \mathrm{l}^{-1}[\text{-}]\rangle$)
where $\mathrm{l}^{-1}[a] \equiv \langle \mathrm{t}[a],\ a\rangle$

lemma *runit-in-hom*:
assumes *ide a*
shows «$\mathrm{r}[a] : a \otimes \mathbf{1} \to a$»
using *assms ide-one* **by** *simp*

lemma *runit'-in-hom*:
assumes *ide a*
shows «$\mathrm{r}^{-1}[a] : a \to a \otimes \mathbf{1}$»
using *assms ide-in-hom trm-in-hom* **by** *blast*

lemma *lunit-in-hom*:
assumes *ide a*
shows «$\mathrm{l}[a] : \mathbf{1} \otimes a \to a$»
using *assms ide-one* **by** *simp*

lemma *lunit'-in-hom*:
assumes *ide a*
shows «$\mathrm{l}^{-1}[a] : a \to \mathbf{1} \otimes a$»
using *assms ide-in-hom trm-in-hom* **by** *blast*

lemma *runit-naturality*:
assumes *arr f*
shows $\mathrm{r}[cod\ f] \cdot (f \otimes \mathbf{1}) = f \cdot \mathrm{r}[dom\ f]$
using *assms pr-naturality*(*2*) *ide-char ide-one* **by** *blast*

lemma *inverse-arrows-runit*:
assumes *ide a*
shows *inverse-arrows* $\mathrm{r}[a]$ $\mathrm{r}^{-1}[a]$
proof
show *ide* $(\mathrm{r}[a] \cdot \mathrm{r}^{-1}[a])$
using *assms* **by** *auto*
show *ide* $(\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a])$
proof −
have *ide* $(a \otimes \mathbf{1})$
using *assms ide-one* **by** *blast*
moreover have $\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a] = a \otimes \mathbf{1}$
proof (*intro pr-joint-monic* [*of a* **1** $\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a]$ $a \otimes \mathbf{1}$])
show *ide a* **by** *fact*
show *ide* **1**
using *ide-one* **by** *blast*
show *seq* $\mathfrak{p}_0[a,\ \mathbf{1}]$ $(\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a])$

```
      using assms ide-one runit'-in-hom [of a]
      by (intro seqI) auto
    show 𝔭₀[a, 𝟭] · r⁻¹[a] · r[a] = 𝔭₀[a, 𝟭] · (a ⊗ 𝟭)
    proof −
      have 𝔭₀[a, 𝟭] · r⁻¹[a] · r[a] = (𝔭₀[a, 𝟭] · r⁻¹[a]) · r[a]
        using comp-assoc by simp
      also have ... = t[a] · r[a]
        using assms ide-one
        by (metis in-homE pr-tuple(2) ide-char trm-in-hom)
      also have ... = t[a ⊗ 𝟭]
        using assms ide-one trm-naturality [of r[a]] by simp
      also have ... = 𝔭₀[a, 𝟭] · (a ⊗ 𝟭)
        by (simp add: assms ide-one trm-one trm-tensor)
      finally show ?thesis by blast
    qed
    show 𝔭₁[a, 𝟭] · r⁻¹[a] · r[a] = 𝔭₁[a, 𝟭] · (a ⊗ 𝟭)
      using assms
      by (metis ‹ide (r[a] · r⁻¹[a])› cod-comp cod-pr1 dom-comp ide-compE ide-one
          comp-assoc runit-naturality)
  qed
  ultimately show ?thesis by simp
 qed
qed

lemma lunit-naturality:
assumes arr f
shows C l[cod f] (𝟭 ⊗ f) = C f l[dom f]
 using assms pr-naturality(1) ide-char ide-one by blast

lemma inverse-arrows-lunit:
assumes ide a
shows inverse-arrows l[a] l⁻¹[a]
proof
 show ide (C l[a] l⁻¹[a])
  using assms by auto
 show ide (l⁻¹[a] · l[a])
 proof −
  have l⁻¹[a] · l[a] = 𝟭 ⊗ a
  proof (intro pr-joint-monic [of 𝟭 a l⁻¹[a] · l[a] 𝟭 ⊗ a])
    show ide a by fact
    show ide 𝟭
      using ide-one by blast
    show seq l[a] (l⁻¹[a] · l[a])
      using assms ‹ide (l[a] · l⁻¹[a])› by blast
    show l[a] · l⁻¹[a] · l[a] = l[a] · (𝟭 ⊗ a)
      using assms
      by (metis ‹ide (l[a] · l⁻¹[a])› cod-comp cod-pr0 dom-cod ide-compE ide-one
          comp-assoc lunit-naturality)
    show 𝔭₁[𝟭, a] · l⁻¹[a] · l[a] = 𝔭₁[𝟭, a] · (𝟭 ⊗ a)
```

```
      proof −
        have 𝔭₁[𝟭, a] · l⁻¹[a] · l[a] = (𝔭₁[𝟭, a] · l⁻¹[a]) · l[a]
          using comp-assoc by simp
        also have ... = t[a] · l[a]
          using assms ide-one
          by (metis pr-tuple(1) ide-char in-homE trm-in-hom)
        also have ... = t[𝟭 ⊗ a]
          using assms ide-one trm-naturality [of l[a]] by simp
        also have ... = 𝔭₁[𝟭, a] · (𝟭 ⊗ a)
          by (simp add: assms ide-one pr-coincidence trm-one trm-tensor)
        finally show ?thesis by simp
      qed
    qed
    moreover have ide (𝟭 ⊗ a)
      using assms ide-one by simp
    finally show ?thesis by blast
  qed
qed

lemma pr-expansion:
assumes ide a and ide b
shows 𝔭₀[a, b] = l[b] · (t[a] ⊗ b) and 𝔭₁[a, b] = r[a] · (a ⊗ t[b])
  using assms
  by (auto simp add: comp-ide-arr)

lemma comp-lunit-term-dup:
assumes ide a
shows l[a] · (t[a] ⊗ a) · d[a] = a
  using assms prod-tuple by force

lemma comp-runit-term-dup:
assumes ide a
shows r[a] · (a ⊗ t[a]) · d[a] = a
  using assms prod-tuple by force

lemma dup-coassoc:
assumes ide a
shows a[a, a, a] · (d[a] ⊗ a) · d[a] = (a ⊗ d[a]) · d[a]
proof (intro pr-joint-monic
            [of a a ⊗ a a[a, a, a] · (d[a] ⊗ a) · d[a] (a ⊗ d[a]) · d[a]])
  show ide a by fact
  show ide (a ⊗ a)
    by (simp add: assms)
  show seq 𝔭₀[a, a ⊗ a] (a[a, a, a] · (d[a] ⊗ a) · d[a])
    using assms by simp
  show 𝔭₀[a, a ⊗ a] · a[a, a, a] · (d[a] ⊗ a) · d[a] = 𝔭₀[a, a ⊗ a] · (a ⊗ d[a]) · d[a]
  proof −
    have 𝔭₀[a, a ⊗ a] · a[a, a, a] · (d[a] ⊗ a) · d[a] =
          ((𝔭₀[a, a ⊗ a] · a[a, a, a]) · (d[a] ⊗ a)) · d[a]
```

```
    using comp-assoc by simp
  also have ... = ⟨((𝔭₀[a, a] · 𝔭₁[a ⊗ a, a]) · (d[a] ⊗ a)) · d[a], (a · 𝔭₀[a, a]) · d[a]⟩
    using assms assoc-def by simp
  also have ... = d[a]
    using assms comp-assoc by simp
  also have ... = (𝔭₀[a, a ⊗ a] · (a ⊗ d[a])) · d[a]
    using assms assoc-def comp-assoc by simp
  also have ... = 𝔭₀[a, a ⊗ a] · (a ⊗ d[a]) · d[a]
    using comp-assoc by simp
  finally show ?thesis by blast
 qed
 show 𝔭₁[a, a ⊗ a] · a[a, a, a] · (d[a] ⊗ a) · d[a] = 𝔭₁[a, a ⊗ a] · (a ⊗ d[a]) · d[a]
 proof −
  have 𝔭₁[a, a ⊗ a] · a[a, a, a] · (d[a] ⊗ a) · d[a] =
       ((𝔭₁[a, a ⊗ a] · a[a, a, a]) · (d[a] ⊗ a)) · d[a]
    using comp-assoc by simp
  also have ... = ((𝔭₁[a, a] · 𝔭₁[a ⊗ a, a]) · (d[a] ⊗ a)) · d[a]
    using assms assoc-def by simp
  also have ... = a
    using assms comp-assoc by simp
  also have ... = (a · 𝔭₁[a, a]) · d[a]
    using assms comp-assoc by simp
  also have ... = (𝔭₁[a, a ⊗ a] · (a ⊗ d[a])) · d[a]
    using assms by simp
  also have ... = 𝔭₁[a, a ⊗ a] · (a ⊗ d[a]) · d[a]
    using comp-assoc by simp
  finally show ?thesis by blast
 qed
qed

lemma comp-assoc-tuple:
assumes «f0 : a → b0» and «f1 : a → b1» and «f2 : a → b2»
shows a[b0, b1, b2] · ⟨⟨f0, f1⟩, f2⟩ = ⟨f0, ⟨f1, f2⟩⟩
and a⁻¹[b0, b1, b2] · ⟨f0, ⟨f1, f2⟩⟩ = ⟨⟨f0, f1⟩, f2⟩
 using assms assoc-def assoc'-def comp-assoc by fastforce+

lemma dup-tensor:
assumes ide a and ide b
shows d[a ⊗ b] = a⁻¹[a, b, a ⊗ b] · (a ⊗ a[b, a, b]) · (a ⊗ σ (a, b) ⊗ b) ·
                 (a ⊗ a⁻¹[a, b, b]) · a[a, a, b ⊗ b] · (d[a] ⊗ d[b])
proof (intro pr-joint-monic [of a ⊗ b a ⊗ b d[a ⊗ b]])
 show ide (a ⊗ b) and ide (a ⊗ b)
  by (auto simp add: assms)
 show seq 𝔭₀[a ⊗ b, a ⊗ b] (d[a ⊗ b])
  using assms by simp
 have 1: a⁻¹[a, b, a ⊗ b] · (a ⊗ a[b, a, b]) · (a ⊗ σ (a, b) ⊗ b) ·
           (a ⊗ a⁻¹[a, b, b]) · a[a, a, b ⊗ b] · (d[a] ⊗ d[b]) =
         ⟨a ⊗ b, a ⊗ b⟩
 proof −
```

have $\mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot (a \otimes \mathrm{a}[b,\ a,\ b]) \cdot (a \otimes \sigma\ (a,\ b) \otimes b) \cdot$
$(a \otimes \mathrm{a}^{-1}[a,\ b,\ b]) \cdot \mathrm{a}[a,\ a,\ b \otimes b] \cdot (\mathrm{d}[a] \otimes \mathrm{d}[b])$
$= \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot (a \otimes \mathrm{a}[b,\ a,\ b]) \cdot (a \otimes \sigma\ (a,\ b) \otimes b) \cdot$
$(a \otimes \mathrm{a}^{-1}[a,\ b,\ b]) \cdot \langle \mathfrak{p}_1[a,\ b],\ \langle \mathfrak{p}_1[a,\ b],\ \mathrm{d}[b] \cdot \mathfrak{p}_0[a,\ b]\rangle\rangle$
proof −
have $\mathrm{a}[a,\ a,\ b \otimes b] \cdot (\mathrm{d}[a] \otimes \mathrm{d}[b]) = \langle \mathfrak{p}_1[a,\ b],\ \langle \mathfrak{p}_1[a,\ b],\ \mathrm{d}[b] \cdot \mathfrak{p}_0[a,\ b]\rangle\rangle$
using *assms assoc-def comp-assoc pr-naturality comp-cod-arr* **by** *simp*
thus *?thesis* **by** *presburger*
qed
also have $\ldots = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot$
$\langle a \cdot a \cdot a \cdot \mathfrak{p}_1[a,\ b],$
$\mathrm{a}[b,\ a,\ b] \cdot (\mathrm{s}[a,\ b] \cdot (a \otimes b) \otimes b) \cdot$
$\mathrm{a}^{-1}[a,\ b,\ b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \mathrm{d}[b \cdot \mathfrak{p}_0[a,\ b]]\rangle\rangle$
using *assms prod-tuple* **by** *simp*
also have $\ldots = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot$
$\langle \mathfrak{p}_1[a,\ b],$
$\mathrm{a}[b,\ a,\ b] \cdot (\mathrm{s}[a,\ b] \otimes b) \cdot \mathrm{a}^{-1}[a,\ b,\ b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \mathrm{d}[\mathfrak{p}_0[a,\ b]]\rangle\rangle$
proof −
have $a \cdot a \cdot a \cdot \mathfrak{p}_1[a,\ b] = \mathfrak{p}_1[a,\ b]$
using *assms comp-cod-arr* **by** *simp*
moreover have $b \cdot \mathfrak{p}_0[a,\ b] = \mathfrak{p}_0[a,\ b]$
using *assms comp-cod-arr* **by** *simp*
moreover have $\mathrm{s}[a,\ b] \cdot (a \otimes b) \otimes b = \mathrm{s}[a,\ b] \otimes b$
using *assms comp-arr-dom* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
also have $\ldots = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \mathrm{a}[b,\ a,\ b] \cdot (\mathrm{s}[a,\ b] \otimes b) \cdot$
$\langle\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle,\ \mathfrak{p}_0[a,\ b]\rangle\rangle$
proof −
have $\mathrm{a}^{-1}[a,\ b,\ b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \mathrm{d}[\mathfrak{p}_0[a,\ b]]\rangle = \langle\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle,\ \mathfrak{p}_0[a,\ b]\rangle$
using *assms comp-assoc-tuple(2)* **by** *blast*
thus *?thesis* **by** *simp*
qed
also have $\ldots = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \mathrm{a}[b,\ a,\ b] \cdot \langle \mathrm{s}[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle\rangle$
using *assms prod-tuple comp-arr-dom comp-cod-arr* **by** *simp*
also have $\ldots = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot \langle \mathfrak{p}_1[a,\ b],\ \langle \mathfrak{p}_0[a,\ b],\ \langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle\rangle\rangle$
using *assms comp-assoc-tuple(1)*
by (*metis sym-def pr-in-hom*)
also have $\ldots = \langle\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle,\ \langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle\rangle$
using *assms comp-assoc-tuple(2)*
by (*metis* ‹*ide* $(a \otimes b)$› *ide-in-hom pr-in-hom tuple-pr*)
also have $\ldots = \mathrm{d}[a \otimes b]$
using *assms* **by** *simp*
finally show *?thesis* **by** *simp*
qed
show $\mathfrak{p}_0[a \otimes b,\ a \otimes b] \cdot \mathrm{d}[a \otimes b]$
$= \mathfrak{p}_0[a \otimes b,\ a \otimes b] \cdot$
$\mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot (a \otimes \mathrm{a}[b,\ a,\ b]) \cdot (a \otimes \sigma\ (a,\ b) \otimes b) \cdot$
$(a \otimes \mathrm{a}^{-1}[a,\ b,\ b]) \cdot \mathrm{a}[a,\ a,\ b \otimes b] \cdot (\mathrm{d}[a] \otimes \mathrm{d}[b])$

```
    using assms 1 by force
  show 𝔭₁[a ⊗ b, a ⊗ b] · d[a ⊗ b]
        = 𝔭₁[a ⊗ b, a ⊗ b] ·
          a⁻¹[a, b, a ⊗ b] · (a ⊗ a[b, a, b]) · (a ⊗ σ (a, b) ⊗ b) ·
          (a ⊗ a⁻¹[a, b, b]) · a[a, a, b ⊗ b] · (d[a] ⊗ d[b])
    using assms 1 by force
qed



lemma terminal-tensor-one-one:
shows terminal (𝟭 ⊗ 𝟭)
proof
  show ide (𝟭 ⊗ 𝟭)
    using ide-one by simp
  show ⋀a. ide a ⟹ ∃!f. «f : a → 𝟭 ⊗ 𝟭»
  proof −
    fix a
    assume a: ide a
    show ∃!f. «f : a → 𝟭 ⊗ 𝟭»
    proof
      show «inv ι · t[a] : a → 𝟭 ⊗ 𝟭»
        using a ide-one inverse-arrows-ι inverse-unique trm-in-hom by fastforce
      show ⋀f. «f : a → 𝟭 ⊗ 𝟭» ⟹ f = inv ι · t[a]
        by (metis ‹«local.inv ι · t[a] : a → 𝟭 ⊗ 𝟭»› a in-homE ide-one pr-coincidence
            pr-joint-monic trm-naturality trm-simps(1) unit-eq-trm)
    qed
  qed
qed

end
```

22.3.2 Exponentials

The following prepare the way for the definition of cartesian closed categories. The notion of exponential has to be defined in relation to products. Here we use a generic choice of products for this purpose.

```
context cartesian-category
begin

  definition has-as-exponential
  where has-as-exponential b c x e ≡
        ide b ∧ ide x ∧ «e : some-prod x b → c» ∧
        (∀a g. ide a ∧ «g : some-prod a b → c» ⟶
                (∃!f. «f : a → x» ∧ g = C e (some-prod f b)))

  lemma has-as-exponentialI [intro]:
  assumes ide b and ide x and «e : some-prod x b → c»
  and ⋀a g. ⟦ide a; «g : some-prod a b → c»⟧ ⟹ ∃!f. «f : a → x» ∧ g = C e (some-prod f b)
```

shows *has-as-exponential b c x e*
using *assms has-as-exponential-def* **by** *simp*

lemma *has-as-exponentialE* [*elim*]:
assumes *has-as-exponential b c x e*
and ⟦*ide b*; *ide x*; «*e* : *some-prod x b* → *c*»;
⋀*a g*. ⟦*ide a*; «*g* : *some-prod a b* → *c*»⟧ ⟹ ∃!*f*. «*f* : *a* → *x*» ∧ *g* = *C e* (*some-prod f b*)⟧
⟹ *T*
shows *T*
using *assms has-as-exponential-def* **by** *simp*

lemma *exponentials-are-isomorphic*:
assumes *has-as-exponential b c x e* **and** *has-as-exponential b c x′ e′*
shows ∃!*h*. «*h* : *x* → *x′*» ∧ *e* = *e′* · *some-prod h b*
and ⋀*h*. ⟦«*h* : *x* → *x′*»; *e* = *e′* · (*some-prod h b*)⟧ ⟹ *iso h*
proof −
have *ide b* ∧ *ide c*
using *assms*(*1*) *has-as-exponential-def* **by** *auto*
have *ide x* ∧ «*e* : *some-prod x b* → *c*»
using *assms*(*1*) *has-as-exponential-def* **by** *blast*
have *ide x′* ∧ «*e′* : *some-prod x′ b* → *c*»
using *assms*(*2*) *has-as-exponential-def* **by** *blast*
show *1*: ∃!*h*. «*h* : *x* → *x′*» ∧ *e* = *e′* · *some-prod h b*
using *assms has-as-exponential-def* **by** *simp*
have *2*: ∃!*h*. «*h* : *x′* → *x*» ∧ *e′* = *e* · *some-prod h b*
using *assms has-as-exponential-def* **by** *simp*
have *3*: ∃!*h*. «*h* : *x* → *x*» ∧ *e* = *e* · *some-prod h b*
using *assms has-as-exponential-def* **by** *simp*
have *4*: ∃!*h*. «*h* : *x′* → *x′*» ∧ *e′* = *e′* · *some-prod h b*
using *assms has-as-exponential-def* **by** *simp*
have *5*: ⋀*h*. «*h* : *x* → *x*» ∧ *e* = *e* · *some-prod h b* ⟹ *h* = *x*
by (*metis assms*(*1*) *3 category.in-homE category-axioms comp-arr-dom has-as-exponential-def*
partial-composition.ide-in-hom partial-composition-axioms)
have *6*: ⋀*h*. «*h* : *x′* → *x′*» ∧ *e′* = *e′* · *some-prod h b* ⟹ *h* = *x′*
by (*metis assms*(*2*) *4 category.in-homE category-axioms comp-arr-dom has-as-exponential-def*
partial-composition.ide-in-hom partial-composition-axioms)
let *?f* = *THE h*. «*h* : *x′* → *x*» ∧ *e′* = *e* · *some-prod h b*
let *?g* = *THE h*. «*h* : *x* → *x′*» ∧ *e* = *e′* · *some-prod h b*
have *inverse-arrows ?f ?g*
proof
have *?g* · *?f* = *x′*
proof −
have «*?g* · *?f* : *x′* → *x′*» ∧ *e′* = *e′* · *some-prod ?g b* · *some-prod ?f b*
proof
show «*?g* · *?f* : *x′* → *x′*»
using *1 2 theI* [*of* λ*h*. «*h* : *x* → *x′*» ∧ *e* = *e′* · *some-prod h b*]
theI [*of* λ*h*. «*h* : *x′* → *x*» ∧ *e′* = *e* · *some-prod h b*]
by (*meson comp-in-homI*)
show *e′* = *e′* · *some-prod ?g b* · *some-prod ?f b*

```
      using 1 2 theI [of λh. «h : x → x'» ∧ e = e' · some-prod h b]
            theI [of λh. «h : x' → x» ∧ e' = e · some-prod h b]
      by (metis (no-types, lifting) comp-assoc)
    qed
    hence «?g · ?f : x' → x'» ∧ e' = e' · some-prod (?g · ?f) (b · b)
    by (metis (no-types, lifting) ‹ide b ∧ ide c› arrI ext
        elementary-category-with-binary-products.interchange
        extends-to-elementary-category-with-binary-products ide-def)
    thus ?thesis
    using 6
    by (simp add: ‹ide b ∧ ide c›)
  qed
  thus ide (?g · ?f)
  using ‹ide x' ∧ «e' : some-prod x' b → c»› by presburger
  have ?f · ?g = x
  proof −
  have «?f · ?g : x → x» ∧ e = e · some-prod ?f b · some-prod ?g b
  proof
  show «?f · ?g : x → x»
  using 1 2 theI [of λh. «h : x → x'» ∧ e = e' · some-prod h b]
        theI [of λh. «h : x' → x» ∧ e' = e · some-prod h b]
  by (meson comp-in-homI)
  show e = e · some-prod ?f b · some-prod ?g b
  using 1 2 theI [of λh. «h : x → x'» ∧ e = e' · some-prod h b]
        theI [of λh. «h : x' → x» ∧ e' = e · some-prod h b]
  by (metis (no-types, lifting) comp-assoc)
  qed
  hence «?f · ?g : x → x» ∧ e = e · some-prod (?f · ?g) (b · b)
  by (metis (no-types, lifting) ‹ide b ∧ ide c› arrI ext
      elementary-category-with-binary-products.interchange
      extends-to-elementary-category-with-binary-products ide-def)
  thus ?thesis
  using 5
  by (simp add: ‹ide b ∧ ide c›)
  qed
  thus ide (?f · ?g)
  using ‹ide x ∧ «e : some-prod x b → c»› by presburger
 qed
 hence iso ?g
 by blast
 moreover have ⋀h. ⟦«h : x → x'»; e = e' · (some-prod h b)⟧ ⟹ h = ?g
 using 1 theI [of λh. «h : x → x'» ∧ e = e' · some-prod h b] by auto
 ultimately show ⋀h. ⟦«h : x → x'»; e = e' · (some-prod h b)⟧ ⟹ iso h by simp
qed

end
```

22.4 Category with Finite Products

In this last section, we show that the notion "cartesian category", which we defined to be a category with binary products and terminal object, coincides with the notion "category with finite products". Due to the inability to quantify over types in HOL, we content ourselves with defining the latter notion as "has I-indexed products for every finite set I of natural numbers." We can transfer this property to finite sets at other types using the fact that products are preserved under bijections of the index sets.

```
locale category-with-finite-products =
  category C
for C :: 'c comp +
assumes has-finite-products: finite (I :: nat set) ⟹ has-products I
begin

  lemma has-finite-products':
  assumes I ≠ UNIV
  shows finite I ⟹ has-products I
  proof −
    assume I: finite I
    obtain n φ where φ: bij-betw φ {k. k < (n :: nat)} I
      using I finite-imp-nat-seg-image-inj-on inj-on-imp-bij-betw by fastforce
    show has-products I
      using assms(1) φ has-finite-products has-products-preserved-by-bijection
            category-with-finite-products.has-finite-products
      by blast
  qed

end

lemma (in category) has-binary-products-if:
assumes has-products ({0, 1} :: nat set)
shows has-binary-products
proof (unfold has-binary-products-def)
  show ∀ a0 a1. ide a0 ∧ ide a1 ⟶ (∃ p0 p1. has-as-binary-product a0 a1 p0 p1)
  proof (intro allI impI)
    fix a0 a1
    assume 1: ide a0 ∧ ide a1
    show ∃ p0 p1. has-as-binary-product a0 a1 p0 p1
    proof −
      interpret J: binary-product-shape
        by unfold-locales
      interpret D: binary-product-diagram C a0 a1
        using 1 by unfold-locales auto
      interpret discrete-diagram J.comp C D.map
        using J.is-discrete
        by unfold-locales auto
      show ∃ p0 p1. has-as-binary-product a0 a1 p0 p1
      proof (unfold has-as-binary-product-def)
```

Here we have to work around the fact that *has-finite-products* is defined in terms of *nat set*, whereas *has-as-binary-product* is defined in terms of $J.arr$ *set*.

```
      let ?φ = (λx :: nat. if x = 0 then J.FF else J.TT)
      let ?ψ = λj. if j = J.FF then 0 else 1
      have 2: ∃ a π. D.limit-cone a π
      proof −
        have bij-betw ?φ ({0, 1} :: nat set) {J.FF, J.TT}
          using bij-betwI [of ?φ {0, 1} :: nat set {J.FF, J.TT} ?ψ] by fastforce
        hence has-products {J.FF, J.TT}
          using assms has-products-def [of {J.FF, J.TT}]
                has-products-preserved-by-bijection
                  [of {0, 1} :: nat set ?φ {J.FF, J.TT}]
          by blast
        hence ∃ a. has-as-product J.comp D.map a
          using has-products-def [of {J.FF, J.TT}]
                discrete-diagram-axioms J.arr-char
          by blast
        hence ∃ a π. product-cone J.comp C D.map a π
          using has-as-product-def by blast
        thus ?thesis
          unfolding product-cone-def by simp
      qed
      obtain a π where π: D.limit-cone a π
        using 2 by auto
      interpret π: limit-cone J.comp C D.map a π
        using π by auto
      have π = D.mkCone (π J.FF) (π J.TT)
      proof −
        have ⋀a. J.ide a ⟹ π a = D.mkCone (π J.FF) (π J.TT) a
          using D.mkCone-def J.ide-char by auto
        moreover have a = dom (π J.FF)
          by simp
        moreover have D.cone a (D.mkCone (π (J.MkIde False)) (π (J.MkIde True)))
          using 1 D.cone-mkCone [of π J.FF π J.TT] by auto
        ultimately show ?thesis
          using D.mkCone-def π.natural-transformation-axioms
                D.cone-mkCone [of π J.FF π J.TT]
                natural-transformation-eqI
                  [of J.comp C π.A.map D.map π D.mkCone (π J.FF) (π J.TT)]
                cone-def [of J.comp C D.map a D.mkCone (π J.FF) (π J.TT)] J.ide-char
          by blast
      qed
      hence D.limit-cone (dom (π J.FF)) (D.mkCone (π J.FF) (π J.TT))
        using π.limit-cone-axioms by simp
      thus ∃ p0 p1. ide a0 ∧ ide a1 ∧ D.has-as-binary-product p0 p1
        using 1 by blast
    qed
  qed
qed
```

qed

sublocale *category-with-finite-products* $\subseteq$ *category-with-binary-products C*
  using *has-binary-products-if has-finite-products*
  by (*unfold-locales*, *unfold has-binary-products-def*) *simp*

proposition (**in** *category-with-finite-products*) *is-category-with-binary-products*$_{CFP}$:
shows *category-with-binary-products C*
  ..

sublocale *category-with-finite-products* $\subseteq$ *category-with-terminal-object C*
proof
  interpret *J*: *discrete-category* {} :: *nat set*
    by *unfold-locales auto*
  interpret *D*: *empty-diagram J.comp C* $\lambda j.$ *null*
    by *unfold-locales auto*
  interpret *D*: *discrete-diagram J.comp C* $\lambda j.$ *null*
    using *J.is-discrete* **by** *unfold-locales auto*
  have $\bigwedge a.$ *D.has-as-limit a* $\longleftrightarrow$ *has-as-product J.comp* ($\lambda j.$ *null*) *a*
    using *product-cone-def J.category-axioms category-axioms D.discrete-diagram-axioms*
        has-as-product-def product-cone-def
    by *metis*
  moreover have $\exists\, a.$ *has-as-product J.comp* ($\lambda j.$ *null*) *a*
    using *has-finite-products* [*of* {} :: *nat set*] *has-products-def* [*of* {} :: *nat set*]
        D.discrete-diagram-axioms
    by *blast*
  ultimately have $\exists\, a.$ *D.has-as-limit a* **by** *blast*
  thus $\exists\, a.$ *terminal a* **using** *D.has-as-limit-iff-terminal* **by** *blast*
qed

proposition (**in** *category-with-finite-products*) *is-category-with-terminal-object*$_{CFP}$:
shows *category-with-terminal-object C*
  ..

sublocale *category-with-finite-products* $\subseteq$ *cartesian-category* **..**

proposition (**in** *category-with-finite-products*) *is-cartesian-category*$_{CFP}$:
shows *cartesian-category C*
  ..

context *category*
begin

  lemma *binary-product-of-products-is-product*:
  assumes *has-as-product J0 D0 a0* **and** *has-as-product J1 D1 a1*
  and *has-as-binary-product a0 a1 p0 p1*
  and *Collect* (*partial-composition.arr J0*) $\cap$ *Collect* (*partial-composition.arr J1*) = {}
  and *partial-magma.null J0* = *partial-magma.null J1*
  shows *has-as-product*

```
      (discrete-category.comp
        (Collect (partial-composition.arr J0) ∪ Collect (partial-composition.arr J1))
        (partial-magma.null J0))
      (λi. if i ∈ Collect (partial-composition.arr J0) then D0 i
        else if i ∈ Collect (partial-composition.arr J1) then D1 i
        else null)
      (dom p0)
proof −
  obtain π0 where π0: product-cone J0 (·) D0 a0 π0
    using assms(1) has-as-product-def by blast
  obtain π1 where π1: product-cone J1 (·) D1 a1 π1
    using assms(2) has-as-product-def by blast
  interpret J0: category J0
    using π0 product-cone.axioms(1) by metis
  interpret J1: category J1
    using π1 product-cone.axioms(1) by metis
  interpret D0: discrete-diagram J0 C D0
    using π0 product-cone.axioms(3) by metis
  interpret D1: discrete-diagram J1 C D1
    using π1 product-cone.axioms(3) by metis
  interpret π0: product-cone J0 C D0 a0 π0
    using π0 by auto
  interpret π1: product-cone J1 C D1 a1 π1
    using π1 by auto
  interpret J: discrete-category ‹Collect J0.arr ∪ Collect J1.arr› J0.null
    using J0.not-arr-null assms(5) by unfold-locales auto
  interpret X: binary-product-shape .
  interpret a0xa1: binary-product-diagram C a0 a1
    using assms(3) has-as-binary-product-def
    by (simp add: binary-product-diagram.intro binary-product-diagram-axioms.intro
        category-axioms)
  have p0p1: a0xa1.has-as-binary-product p0 p1
    using assms(3) has-as-binary-product-def [of a0 a1 p0 p1] by simp

  let ?D = (λi. if i ∈ Collect J0.arr then D0 i
                else if i ∈ Collect J1.arr then D1 i
                else null)
  let ?a = dom p0
  let ?π = λi. if i ∈ Collect J0.arr then π0 i · p0
               else if i ∈ Collect J1.arr then π1 i · p1
               else null

  let ?p0p1 = a0xa1.mkCone p0 p1
  interpret p0p1: limit-cone X.comp C a0xa1.map ?a ?p0p1
    using p0p1 by simp
  have a: ide ?a
    using p0p1.ide-apex by simp
  have p0: «p0 : ?a → a0»
  using a0xa1.mkCone-def p0p1.preserves-hom [of X.FF X.FF X.FF] X.ide-char X.ide-in-hom
```

```
  by auto
have p1: «p1 : ?a → a1»
using a0xa1.mkCone-def p0p1.preserves-hom [of X.TT X.TT X.TT] X.ide-char X.ide-in-hom
  by auto

interpret D: discrete-diagram J.comp C ?D
  using assms J.arr-char J.dom-char J.cod-char J.is-discrete D0.is-discrete D1.is-discrete
        J.cod-comp J.seq-char
  by unfold-locales auto
interpret A: constant-functor J.comp C ?a
  using p0p1.ide-apex by unfold-locales simp
interpret π: natural-transformation J.comp C A.map ?D ?π
proof
  fix j
  show ¬ J.arr j ⟹ ?π j = null
    by simp
  assume j: J.arr j
  have π0j: J0.arr j ⟹ «π0 j : a0 → D0 j»
    using D0.is-discrete by auto
  have π1j: J1.arr j ⟹ «π1 j : a1 → D1 j»
    using D1.is-discrete by auto
  show arr (?π j)
    using j J.arr-char p0 p1 π0j π1j by fastforce
  show ?D j · ?π (J.dom j) = ?π j
  proof −
    have 0: J0.arr j ⟹ D0 j · π0 j · p0 = π0 j · p0
      by (metis D0.is-discrete J0.ide-char π0.naturality1 comp-assoc)
    have 1: J1.arr j ⟹ D1 j · π1 j · p1 = π1 j · p1
      by (metis D1.is-discrete J1.ide-char π1.naturality1 comp-assoc)
    show ?thesis
      using 0 1 by auto
  qed
  show ?π (J.cod j) · A.map j = ?π j
    using j comp-arr-dom p0 p1 comp-assoc by auto
qed
interpret π: cone J.comp C ?D ?a ?π ..
interpret π: product-cone J.comp C ?D ?a ?π
proof
  show ⋀a' χ'. D.cone a' χ' ⟹ ∃!f. «f : a' → ?a» ∧ D.cones-map f ?π = χ'
  proof −
    fix a' χ'
    assume χ': D.cone a' χ'
    interpret χ': cone J.comp C ?D a' χ'
      using χ' by simp
    show ∃!f. «f : a' → ?a» ∧ D.cones-map f ?π = χ'
    proof
      let ?χ0' = λi. if i ∈ Collect J0.arr then χ' i else null
      let ?χ1' = λi. if i ∈ Collect J1.arr then χ' i else null
      have 0: ⋀i. i ∈ Collect J0.arr ⟹ χ' i ∈ hom a' (D0 i)
```

```
  using J.arr-char by auto
have 1: ⋀i. i ∈ Collect J1.arr ⟹ χ' i ∈ hom a' (D1 i)
  using J.arr-char ‹Collect J0.arr ∩ Collect J1.arr = {}› by force
interpret A0': constant-functor J0 C a'
  apply unfold-locales using χ'.ide-apex by auto
interpret A1': constant-functor J1 C a'
  apply unfold-locales using χ'.ide-apex by auto
interpret χ0': cone J0 C D0 a' ?χ0'
proof (unfold-locales)
  fix j
  show ¬ J0.arr j ⟹ (if j ∈ Collect J0.arr then χ' j else null) = null
    by simp
  assume j: J0.arr j
  show arr (?χ0' j)
    using j by simp
  have 0: dom (?χ0' j) = A0'.map (J0.dom j)
    using j by simp
  have 1: cod (?χ0' j) = D0 (J0.cod j)
    using j J.arr-char J.cod-char D0.is-discrete by simp
  show D0 j · (?χ0' (J0.dom j)) = ?χ0' j
    using 1 j J.arr-char D0.is-discrete comp-cod-arr by simp
  show ?χ0' (J0.cod j) · A0'.map j = ?χ0' j
    using 0 j J.arr-char D0.is-discrete comp-arr-dom by simp
qed
interpret χ1': cone J1 C D1 a' ?χ1'
proof (unfold-locales)
  fix j
  show ¬ J1.arr j ⟹ (if j ∈ Collect J1.arr then χ' j else null) = null
    by simp
  assume j: J1.arr j
  show arr (?χ1' j)
    using j by simp
  have 0: dom (?χ1' j) = A1'.map (J1.dom j)
    using j by simp
  have 1: cod (?χ1' j) = D1 (J1.cod j)
    using assms(4) j J.arr-char J.cod-char D1.is-discrete by auto
  show D1 j · (?χ1' (J1.dom j)) = ?χ1' j
    using 1 j J.arr-char D1.is-discrete comp-cod-arr by simp
  show ?χ1' (J1.cod j) · A1'.map j = ?χ1' j
    using 0 j J.arr-char D1.is-discrete comp-arr-dom by simp
qed
define f0 where f0 = π0.induced-arrow a' ?χ0'
define f1 where f1 = π1.induced-arrow a' ?χ1'
have f0: «f0 : a' → a0»
  using f0-def π0.induced-arrowI χ0'.cone-axioms by simp
have f1: «f1 : a' → a1»
  using f1-def π1.induced-arrowI χ1'.cone-axioms by simp
have 2: a0xa1.is-rendered-commutative-by f0 f1
  using f0 f1 by auto
```

```
interpret p0p1: binary-product-cone C a0 a1 p0 p1 ..
interpret f0f1: cone X.comp C a0xa1.map a' ‹a0xa1.mkCone f0 f1›
  using 2 f0 f1 a0xa1.cone-mkCone [of f0 f1] by auto
define f where f = p0p1.induced-arrow a' (a0xa1.mkCone f0 f1)

have f: «f : a' → ?a»
  using f-def 2 f0 f1 p0p1.induced-arrowI'(1) by auto
moreover have χ': D.cones-map f ?π = χ'
proof
  fix j
  show D.cones-map f ?π j = χ' j
  proof (cases J0.arr j, cases J1.arr j)
    show ⟦J0.arr j; J1.arr j⟧ ⟹ D.cones-map f ?π j = χ' j
      using assms(4) by auto
    show ⟦J0.arr j; ¬ J1.arr j⟧ ⟹ D.cones-map f ?π j = χ' j
    proof −
      assume J0: J0.arr j and J1: ¬ J1.arr j
      have D.cones-map f ?π j = (π0 j · p0) · f
        using f J0 J1 π.cone-axioms by auto
      also have ... = π0 j · p0 · f
        using comp-assoc by simp
      also have ... = π0 j · f0
        using 2 f0 f1 f-def p0p1.induced-arrowI' by auto
      also have ... = χ' j
      proof −
        have π0 j · f0 = π0 j · π0.induced-arrow' a' χ'
          unfolding f0-def by simp
        also have ... = (λj. if J0.arr j then
                                π0 j · π0.induced-arrow a'
                                      (λi. if i ∈ Collect J0.arr then χ' i else null)
                              else null) j
          using J0 by simp
        also have ... = D0.mkCone χ' j
        proof −
          have (λj. if J0.arr j then
                      π0 j · π0.induced-arrow a'
                                (λi. if i ∈ Collect J0.arr then χ' i else null)
                    else null) =
                D0.mkCone χ'
            using f0 f0-def π0.induced-arrowI(2) [of ?χ0' a'] J0
                  D0.mkCone-cone χ0'.cone-axioms π0.cone-axioms J0
            by auto
          thus ?thesis by meson
        qed
        also have ... = χ' j
          using J0 by simp
        finally show ?thesis by blast
      qed
```

```
      finally show ?thesis by simp
    qed
    show ¬ J0.arr j ⟹ D.cones-map f ?π j = χ' j
    proof (cases J1.arr j)
      show ⟦¬ J0.arr j; ¬ J1.arr j⟧ ⟹ D.cones-map f ?π j = χ' j
        using f π.cone-axioms χ'.extensionality by auto
      show ⟦¬ J0.arr j; J1.arr j⟧ ⟹ D.cones-map f ?π j = χ' j
      proof −
        assume J0: ¬ J0.arr j and J1: J1.arr j
        have D.cones-map f ?π j = (π1 j · p1) · f
          using J0 J1 f π.cone-axioms by auto
        also have ... = π1 j · p1 · f
          using comp-assoc by simp
        also have ... = π1 j · f1
          using 2 f0 f1 f-def p0p1.induced-arrowI' by auto
        also have ... = χ' j
        proof −
          have π1 j · f1 = π1 j · π1.induced-arrow' a' χ'
            unfolding f1-def by simp
          also have ... = (λj. if J1.arr j then
                                 π1 j · π1.induced-arrow a'
                                        (λi. if i ∈ Collect J1.arr
                                             then χ' i else null)
                               else null) j
            using J1 by simp
          also have ... = D1.mkCone χ' j
          proof −
            have (λj. if J1.arr j then
                        π1 j · π1.induced-arrow a'
                               (λi. if i ∈ Collect J1.arr then χ' i else null)
                      else null) =
                 D1.mkCone χ'
              using f1 f1-def π1.induced-arrowI(2) [of ?χ1' a'] J1
                    D1.mkCone-cone [of a' χ'] χ1'.cone-axioms π1.cone-axioms J1
              by auto
            thus ?thesis by meson
          qed
          also have ... = χ' j
            using J1 by simp
          finally show ?thesis by blast
        qed
        finally show ?thesis by simp
      qed
    qed
  qed
qed
ultimately show «f : a' → ?a» ∧ D.cones-map f ?π = χ' by blast
show ⋀f'. «f' : a' → ?a» ∧ D.cones-map f' ?π = χ' ⟹ f' = f
proof −
```

fix f'
assume f': «$f' : a' \to ?a$» $\wedge$ $D.cones\text{-}map\ f'\ ?\pi = \chi'$
let $?f0' = p0 \cdot f'$
let $?f1' = p1 \cdot f'$
have 1: $a0xa1.is\text{-}rendered\text{-}commutative\text{-}by\ ?f0'\ ?f1'$
  using $f'\ p0\ p1\ p0p1.renders\text{-}commutative\ seqI'$ **by** $auto$
have $f0'$: «$?f0' : a' \to a0$»
  using $f'\ p0$ **by** $auto$
have $f1'$: «$?f1' : a' \to a1$»
  using $f'\ p1$ **by** $auto$
have $p0 \cdot f = p0 \cdot f'$
proof −
  have $D0.cones\text{-}map\ (p0 \cdot f)\ \pi 0 = ?\chi 0'$
    using $f\ p0\ \pi 0.cone\text{-}axioms\ \chi'\ \pi.cone\text{-}axioms\ comp\text{-}assoc\ assms(4)\ seqI'$
    by $fastforce$
  moreover have $D0.cones\text{-}map\ (p0 \cdot f')\ \pi 0 = ?\chi 0'$
    using $f'\ p0\ \pi 0.cone\text{-}axioms\ \pi.cone\text{-}axioms\ comp\text{-}assoc\ assms(4)\ seqI'$
    by $fastforce$
  moreover have $p0 \cdot f = f0$
    using $2\ f0\ f\text{-}def\ p0p1.induced\text{-}arrowI'(2)$ **by** $blast$
  ultimately show $?thesis$
    using $f0\ f0'\ \chi 0'.cone\text{-}axioms\ \pi 0.is\text{-}universal\ [of\ a']$ **by** $auto$
qed
moreover have $p1 \cdot f = p1 \cdot f'$
proof −
  have $D1.cones\text{-}map\ (p1 \cdot f)\ \pi 1 = ?\chi 1'$
  proof
    fix j
    show $D1.cones\text{-}map\ (p1 \cdot f)\ \pi 1\ j = ?\chi 1'\ j$
      using $f\ p1\ \pi 1.cone\text{-}axioms\ \chi'\ \pi.cone\text{-}axioms\ comp\text{-}assoc\ assms(4)\ seqI'$
      apply $auto$
      by $auto$
  qed
  moreover have $D1.cones\text{-}map\ (p1 \cdot f')\ \pi 1 = ?\chi 1'$
  proof
    fix j
    show $D1.cones\text{-}map\ (p1 \cdot f')\ \pi 1\ j = ?\chi 1'\ j$
      using $f'\ p1\ \pi 1.cone\text{-}axioms\ \pi.cone\text{-}axioms\ comp\text{-}assoc\ assms(4)\ seqI'$
      apply $auto$
      by $auto$
  qed
  moreover have $p1 \cdot f = f1$
    using $2\ f1\ f\text{-}def\ p0p1.induced\text{-}arrowI'(3)$ **by** $blast$
    ultimately show $?thesis$
  using $f1\ f1'\ \chi 1'.cone\text{-}axioms\ \pi 1.is\text{-}universal\ [of\ a']$ **by** $auto$
qed
ultimately show $f' = f$
  using $f\ f'\ p0p1.is\text{-}universal'\ [of\ a']$
  by $(metis\ (no\text{-}types,\ lifting)\ 1\ dom\text{-}comp\ in\text{-}homE\ p0p1.is\text{-}universal'\ p1\ seqI')$

```
        qed
      qed
    qed
  qed
  show has-as-product J.comp ?D ?a
    unfolding has-as-product-def
    using π.product-cone-axioms by auto
qed

end

sublocale cartesian-category ⊆ category-with-finite-products
proof
  obtain t where t: terminal t using has-terminal by blast
  { fix n :: nat
    have ⋀I :: nat set. finite I ∧ card I = n ⟹ has-products I
    proof (induct n)
      show ⋀I :: nat set. finite I ∧ card I = 0 ⟹ has-products I
      proof −
        fix I :: nat set
        assume finite I ∧ card I = 0
        hence I: I = {} by force
        thus has-products I
        proof −
          interpret elementary-category-with-terminal-object C ‹𝟭[?]› ‹λa. t[?][a]›
            using extends-to-elementary-category-with-terminal-object by blast
          interpret J: discrete-category I 0
            apply unfold-locales using I by auto
          have ⋀D. discrete-diagram J.comp C D ⟹ ∃a. has-as-product J.comp D a
          proof −
            fix D
            assume D: discrete-diagram J.comp C D
            interpret D: discrete-diagram J.comp C D using D by auto
            interpret D: empty-diagram J.comp C D
              using I J.arr-char by unfold-locales simp
            have has-as-product J.comp D t
              using t D.has-as-limit-iff-terminal has-as-product-def product-cone-def
                    J.category-axioms category-axioms D.discrete-diagram-axioms
              by metis
            thus ∃a. has-as-product J.comp D a by blast
          qed
          moreover have I ≠ UNIV
            using I by blast
          ultimately show ?thesis
            using I has-products-def
            by (metis has-terminal discrete-diagram.product-coneI discrete-diagram-def
                empty-diagram.has-as-limit-iff-terminal empty-diagram.intro
                empty-diagram-axioms.intro empty-iff has-as-product-def mem-Collect-eq)
        qed
```

```
qed
show ⋀n I :: nat set.
       ⟦ (⋀I :: nat set. finite I ∧ card I = n ⟹ has-products I);
         finite I ∧ card I = Suc n ⟧
           ⟹ has-products I
proof −
  fix n :: nat
  fix I :: nat set
  assume IH: ⋀I :: nat set. finite I ∧ card I = n ⟹ has-products I
  assume I: finite I ∧ card I = Suc n
  show has-products I
  proof −
    have card I = 1 ⟹ has-products I
      using I has-unary-products by blast
    moreover have card I ≠ 1 ⟹ has-products I
    proof −
      assume card I ≠ 1
      hence cardI: card I > 1 using I by simp
      obtain i where i: i ∈ I using cardI by fastforce
      let ?I0 = {i} and ?I1 = I − {i}
      have 1: I = ?I0 ∪ ?I1 ∧ ?I0 ∩ ?I1 = {} ∧ card ?I0 = 1 ∧ card ?I1 = n
        using i I cardI by auto
      show has-products I
      proof (unfold has-products-def, intro conjI allI impI)
        show I ≠ UNIV
          using I by auto
        fix J D
        assume D: discrete-diagram J C D ∧ Collect (partial-composition.arr J) = I
        interpret D: discrete-diagram J C D
          using D by simp
        have Null: D.J.null ∉ ?I0 ∧ D.J.null ∉ ?I1
          using D D.J.not-arr-null i by blast
        interpret J0: discrete-category ?I0 D.J.null
          using 1 Null D by unfold-locales auto
        interpret J1: discrete-category ?I1 D.J.null
          using Null by unfold-locales auto
        interpret J0uJ1: discrete-category ‹Collect J0.arr ∪ Collect J1.arr› J0.null
          using Null 1 J0.null-char J1.null-char by unfold-locales auto
        interpret D0: discrete-diagram-from-map ?I0 C D D.J.null
          using 1 J0.ide-char D.preserves-ide D D.is-discrete i by unfold-locales auto
        interpret D1: discrete-diagram-from-map ?I1 C D D.J.null
          using 1 J1.ide-char D.preserves-ide D D.is-discrete i by unfold-locales auto
        obtain a0 where a0: has-as-product J0.comp D0.map a0
          using 1 has-unary-products [of ?I0] has-products-def [of ?I0]
                D0.discrete-diagram-axioms
          by fastforce
        obtain a1 where a1: has-as-product J1.comp D1.map a1
          using 1 I IH [of ?I1] has-products-def [of ?I1] D1.discrete-diagram-axioms
          by blast
```

```
have 2: ∃ p0 p1. has-as-binary-product a0 a1 p0 p1
proof −
  have ide a0 ∧ ide a1
    using a0 a1 product-is-ide by auto
  thus ?thesis
     using a0 a1 has-binary-products has-binary-products-def by simp
qed
obtain p0 p1 where a: has-as-binary-product a0 a1 p0 p1
  using 2 by auto
let ?a = dom p0
have has-as-product J D ?a
proof −
  have D = (λj. if j ∈ Collect J0.arr then D0.map j
                else if j ∈ Collect J1.arr then D1.map j
                else null)
  proof
    fix j
    show D j = (if j ∈ Collect J0.arr then D0.map j
                else if j ∈ Collect J1.arr then D1.map j
                else null)
      using 1 D0.map-def D1.map-def D.extensionality  D J0.arr-char J1.arr-char
      by auto
  qed
  moreover have J = J0uJ1.comp
  proof −
    have ⋀j j'. J j j' = J0uJ1.comp j j'
    proof −
      fix j j'
      show J j j' = J0uJ1.comp j j'
        using D J0uJ1.arr-char J0.arr-char J1.arr-char D.is-discrete i
        apply (cases j ∈ ?I0, cases j' ∈ ?I0)
          apply simp-all
          apply auto[1]
         apply (metis D.J.comp-arr-ide D.J.comp-ide-arr D.J.ext D.J.seqE
            D.is-discrete J0.null-char J0uJ1.null-char)
        by (metis D.J.comp-arr-ide D.J.comp-ide-arr D.J.comp-ide-self
            D.J.ext D.J.seqE D.is-discrete J0.null-char J0uJ1.null-char
            mem-Collect-eq)
    qed
    thus ?thesis by blast
  qed
  moreover have Collect J0.arr ∩ Collect J1.arr = {}
    by auto
  moreover have J0.null = J1.null
    using J0.null-char J1.null-char by simp
  ultimately show has-as-product J D ?a
    using binary-product-of-products-is-product
          [of J0.comp D0.map a0 J1.comp D1.map a1 p0 p1]
        J0.arr-char J1.arr-char
```

```
                    1 a0 a1 a
                by simp
              qed
              thus ∃ a. has-as-product J D a by blast
            qed
          qed
          ultimately show has-products I by blast
        qed
      qed
    qed
  }
  hence 1: ⋀n I :: nat set. finite I ∧ card I = n ⟹ has-products I by simp
  thus ⋀I :: nat set. finite I ⟹ has-products I by blast
qed

proposition (in cartesian-category) is-category-with-finite-products:
shows category-with-finite-products C
  ..

end
```

Chapter 23

Category with Finite Limits

theory *CategoryWithFiniteLimits*
imports *CartesianCategory CategoryWithPullbacks*
begin

In this chapter we define "category with finite limits" and show that such categories coincide with those having pullbacks and a terminal object.

Since we can't quantify over types in HOL, the best we can do at defining the notion "category with finite limits" is to state it for a fixed choice of type (e.g. *nat*) for the arrows of the "diagram shape". However, we then have to go to some trouble to show the existence of finite limits for diagram shapes at other types.

locale *category-with-finite-limits* =
  category +
assumes *has-finite-limits*:
    ⟦ *category* (*J* :: *nat comp*); *finite* (*Collect* (*partial-composition.arr J*)) ⟧
      ⟹ *has-limits-of-shape J*
begin

We show that a category with finite limits has pullbacks and a terminal object and is therefore also a cartesian category.

  interpretation *category-with-pullbacks C*
  proof −
    interpret *J*: *cospan-shape*
      by *unfold-locales*
    have *1*: *finite* (*Collect J.arr*)
    proof −
      have *Collect J.arr* = {*J.AA*, *J.BB*, *J.TT*, *J.AT*, *J.BT*}
        using *J.arr-char cospan-shape.Dom.cases* **by** *auto*
      thus *?thesis* **by** *simp*
    qed
    obtain *J'* :: *nat comp* **where** *J'*: *isomorphic-categories J' J.comp*
      using *1 J.finite-imp-ex-iso-nat-comp* **by** *blast*
    interpret *J'J*: *isomorphic-categories J' J.comp*
      using *J'* **by** *simp*
    obtain *φ ψ* **where** *φψ*: *inverse-functors J.comp J' φ ψ*

```
    using J'J.iso inverse-functors-sym by blast
  interpret φψ: inverse-functors J.comp J' φ ψ
    using φψ by simp
  interpret ψ: invertible-functor J.comp J' ψ
    using φψ.inverse-functors-axioms
    by unfold-locales auto
  show category-with-pullbacks C
  proof
    show has-pullbacks
    proof (unfold has-pullbacks-def has-as-pullback-def, intro allI impI)
      fix f0 f1
      assume cospan: cospan f0 f1
      interpret D: cospan-diagram C f0 f1
        using cospan
        by (simp add: category-axioms cospan-diagram-axioms-def cospan-diagram-def)
      have 2: has-limits-of-shape J.comp
        using 1 bij-betw-finite J'J.A.category-axioms has-finite-limits ψ.bij-betw-arr-sets
              has-limits-preserved-by-isomorphism J'J.isomorphic-categories-axioms
        by blast
      obtain a χ where χ: limit-cone J.comp C D.map a χ
        using 2 D.diagram-axioms has-limits-of-shape-def by blast
      interpret χ: limit-cone J.comp C D.map a χ
        using χ by simp
      have D.map = cospan-diagram.map C f0 f1 by simp
      moreover have a = dom (χ J.AA)
        using J.arr-char χ.component-in-hom by force
      moreover have χ = cospan-diagram.mkCone (·) f0 f1 (χ J.AA) (χ J.BB)
        using D.mkCone-cone χ.cone-axioms by auto
      ultimately have limit-cone (·J) (·)
                        (cospan-diagram.map (·) f0 f1) (dom (χ J.AA))
                        (cospan-diagram.mkCone (·) f0 f1 (χ J.AA) (χ J.BB))
        using χ.limit-cone-axioms by simp
      thus ∃p0 p1. cospan f0 f1 ∧
                   limit-cone (·J) (·)
                     (cospan-diagram.map (·) f0 f1) (dom p0)
                     (cospan-diagram.mkCone (·) f0 f1 p0 p1)
        using cospan by auto
    qed
  qed
qed

lemma is-category-with-pullbacks:
shows category-with-pullbacks C
  ..

sublocale category-with-pullbacks C ..

interpretation category-with-terminal-object C
proof
```

```
  show ∃ a. terminal a
  proof −
    interpret J: discrete-category ‹{} :: nat set› 0
      by unfold-locales simp
    have 1: has-limits-of-shape J.comp
      using has-finite-limits
      by (metis Collect-empty-eq J.arr-char J.is-category empty-iff finite.emptyI)
    interpret D: diagram J.comp C ‹λ-. null›
      by unfold-locales auto
    obtain t τ where τ: D.limit-cone t τ
      using 1 D.diagram-axioms has-limits-of-shape-def by blast
    interpret τ: limit-cone J.comp C ‹λ-. null› t τ
      using τ by simp
    have terminal t
    proof
      show ide t
        using τ.ide-apex by simp
      fix a
      assume a: ide a
      show ∃!f. «f : a → t»
      proof −
        interpret a: constant-functor J.comp C a
          using a by unfold-locales
        interpret χ: cone J.comp C ‹λ-.null› a ‹λ-.null›
          apply unfold-locales
             apply simp
          using dom-null cod-null null-is-zero
          by blast+
        have ∃!f. «f : a → t» ∧ D.cones-map f τ = (λ-. null)
          using τ.induced-arrowI [of λ-.null a] χ.cone-axioms
                τ.is-universal [of a λ-. null]
          by simp
        moreover have ⋀f. «f : a → t» ⟹ D.cones-map f τ = (λ-. null)
          using τ.cone-axioms by auto
        ultimately show ?thesis by auto
      qed
    qed
    thus ?thesis by blast
  qed
qed

lemma is-category-with-terminal-object:
shows category-with-terminal-object C
  ..

sublocale category-with-terminal-object C ..

sublocale category-with-finite-products
  using has-finite-limits has-finite-products-if-has-finite-limits
```

has-limits-of-shape-def diagram-def
by *unfold-locales blast*

sublocale *cartesian-category* **..**

end

locale *category-with-pullbacks-and-terminal* =
category-with-pullbacks +
category-with-terminal-object

sublocale *category-with-finite-limits* ⊆ *category-with-pullbacks-and-terminal* **..**

Conversely, we show that a category with pullbacks and a terminal object also has finite products and equalizers, and therefore has finite limits.

context *category-with-pullbacks-and-terminal*
begin

interpretation *ECP*: *elementary-category-with-pullbacks C some-prj0 some-prj1*
using *extends-to-elementary-category-with-pullbacks* **by** *simp*

abbreviation *some-prj0′*
where *some-prj0′ a b* ≡ (*if ide a* ∧ *ide b then some-prj0* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *else null*)

abbreviation *some-prj1′*
where *some-prj1′ a b* ≡ (*if ide a* ∧ *ide b then some-prj1* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *else null*)

interpretation *ECC*: *elementary-category-with-terminal-object C* ‹$\mathbf{1}^?$› ‹λ*a*. $\mathrm{t}^?[a]$›
using *extends-to-elementary-category-with-terminal-object* **by** *blast*
interpretation *ECC*: *elementary-cartesian-category C some-prj0′ some-prj1′* ‹$\mathbf{1}^?$› ‹λ*a*. $\mathrm{t}^?[a]$›
using *ECC.trm-naturality ECP.universal*
by *unfold-locales auto*

interpretation *category-with-equalizers C*
proof (*unfold-locales*, *unfold has-equalizers-def*, *intro allI impI*)
fix *f0 f1*
assume *par*: *par f0 f1*
interpret *J*: *parallel-pair*
by *unfold-locales*
interpret *D*: *parallel-pair-diagram C f0 f1*
using *par* **by** *unfold-locales auto*
have *1*: *cospan* (*ECC.prod f1* (*dom f0*)) (*ECC.prod f0* (*dom f0*))
using *par* **by** *simp*
let *?g0* = *ECC.prod f0* (*dom f0*) · *ECC.dup* (*dom f0*)
let *?g1* = *ECC.prod f1* (*dom f1*) · *ECC.dup* (*dom f1*)
have *g0*: «*?g0* : *dom f0* → *ECC.prod* (*cod f0*) (*dom f0*)»
using *par* **by** *simp*
have *g1*: «*?g1* : *dom f1* → *ECC.prod* (*cod f1*) (*dom f1*)»
using *par* **by** *simp*

```
    define e0 where e0 = p0?[?g1, ?g0]
    define e1 where e1 = p1?[?g1, ?g0]
    have e0: «e0 : dom e0 → dom f0»
      using par 1 e0-def by auto
    have e1: «e1 : dom e0 → dom f1»
      using par 1 e1-def e0-def by auto
    have eq: e0 = e1
    proof −
      have e1 = some-prj0' (cod f1) (dom f1) · ?g1 · e1
      proof −
        have ((some-prj0' (cod f1) (dom f1) · (ECC.prod f1 (dom f1))) · ECC.dup (dom f1)) ·
e1 =
                dom f1 · e1
          using par ECC.pr-naturality(1) [of dom f1 dom f1 dom f1 f1 dom f1 cod f1]
                comp-cod-arr ECC.pr-dup(1)
          by auto
        also have ... = e1
          using par e1 comp-cod-arr by blast
        finally show ?thesis
          using comp-assoc by simp
      qed
      also have ... = some-prj0' (cod f1) (dom f1) · ?g0 · e0
        using par ECP.pullback-commutes
        unfolding commutative-square-def e0-def e1-def by simp
      also have ... = e0
      proof −
        have ((some-prj0' (cod f1) (dom f1) · (ECC.prod f0 (dom f0))) · ECC.dup (dom f0)) ·
e0 =
                dom f0 · e0
          using par ECC.pr-naturality(1) [of dom f0 dom f0 dom f1 f0 dom f0 cod f0]
                comp-cod-arr ECC.pr-dup(1) ide-dom
          by auto
        also have ... = e0
          using e0 comp-cod-arr by blast
        finally show ?thesis
          using comp-assoc by simp
      qed
      finally show ?thesis by auto
    qed
    have equalizes: D.is-equalized-by e0
    proof
      show seq f0 e0
        using par e0 by auto
      show f0 · e0 = f1 · e0
      proof −
        have f0 · e0 = (f0 · dom f0) · e0
          using par comp-arr-dom by simp
        also have ... = (f0 · (some-prj1' (dom f0) (dom f0) · ECC.dup (dom f0))) · e0
          using par ECC.pr-dup(2) by auto
```

```
    also have ... = ((f0 · some-prj1′ (dom f0) (dom f0)) · ECC.dup (dom f0)) · e0
      using comp-assoc by auto
    also have ... = some-prj1′ (cod f1) (dom f1) · ?g0 · e0
      using par ECC.pr-naturality(2) [of dom f0 dom f0 dom f1 f0 dom f0 cod f0]
      by (metis (no-types, lifting) arr-dom cod-dom dom-dom comp-assoc)
    also have ... = some-prj1′ (cod f1) (dom f1) · ?g1 · e1
      using par ECP.pullback-commutes [of ?g1 ?g0]
      unfolding commutative-square-def e0-def e1-def by simp
    also have ... = (some-prj1′ (cod f1) (dom f1) · ?g1) · e1
      using comp-assoc by simp
    also have ... = (f1 · (some-prj1′ (dom f1) (dom f1) · ECC.dup (dom f1))) · e1
      using par ECC.pr-naturality(2) [of dom f1 dom f1 dom f1 f1 dom f1 cod f1]
      by (metis (no-types, lifting) arr-dom cod-dom dom-dom comp-assoc)
    also have ... = (f1 · dom f1) · e1
      using par ECC.pr-dup(2) by auto
    also have ... = f1 · e1
      using par comp-arr-dom by simp
    also have ... = f1 · e0
      using eq by simp
    finally show ?thesis by simp
  qed
qed
show ∃ e. has-as-equalizer f0 f1 e
proof
  interpret E: constant-functor J.comp C ‹dom e0›
    using par e0 by unfold-locales auto
  interpret χ: cone J.comp C D.map ‹dom e0› ‹D.mkCone e0›
    using equalizes D.cone-mkCone e0-def by auto
  interpret χ: limit-cone J.comp C D.map ‹dom e0› ‹D.mkCone e0›
  proof
    show ⋀a′ χ′. D.cone a′ χ′ ⟹
                 ∃!f. «f : a′ → dom e0» ∧ D.cones-map f (D.mkCone e0) = χ′
    proof −
      fix a′ χ′
      assume χ′: D.cone a′ χ′
      interpret χ′: cone J.comp C D.map a′ χ′
        using χ′ by simp
      have 3: commutative-square ?g1 ?g0 (χ′ J.Zero) (χ′ J.Zero)
      proof
        show cospan ?g1 ?g0
          using par g0 g1 by simp
        show 4: span (χ′ J.Zero) (χ′ J.Zero)
          using J.arr-char by simp
        show 5: dom ?g1 = cod (χ′ J.Zero)
          using par g1 J.arr-char D.map-def by simp
        show ?g1 · χ′ J.Zero = ?g0 · χ′ J.Zero
        proof −
          have ?g1 · χ′ J.Zero = ECC.prod f1 (dom f1) · ECC.dup (dom f1) · χ′ J.Zero
            using comp-assoc by simp
```

```
    also have ... = ECC.prod f1 (dom f1) · ECC.tuple (χ' J.Zero) (χ' J.Zero)
      using par D.map-def J.arr-char comp-cod-arr by auto
    also have ... = ECC.tuple (f1 · χ' J.Zero) (χ' J.Zero)
      using par ECC.prod-tuple [of χ' J.Zero χ' J.Zero f1 dom f1]
            comp-cod-arr
      by (metis (no-types, lifting) 4 5 g1 in-homE seqI)
    also have ... = ECC.tuple (f0 · χ' J.Zero) (χ' J.Zero)
      using par D.is-equalized-by-cone χ'.cone-axioms by auto
    also have ... = ECC.prod f0 (dom f0) · ECC.tuple (χ' J.Zero) (χ' J.Zero)
      using par ECC.prod-tuple [of χ' J.Zero χ' J.Zero f0 dom f0]
            comp-cod-arr
      by (metis (no-types, lifting) 4 5 g1 in-homE seqI)
    also have ... = ECC.prod f0 (dom f0) · ECC.dup (dom f0) · χ' J.Zero
      using par D.map-def J.arr-char comp-cod-arr by auto
    also have ... = ?g0 · χ' J.Zero
      using comp-assoc by simp
    finally show ?thesis by blast
  qed
qed
show ∃!f. «f : a' → dom e0» ∧ D.cones-map f (D.mkCone e0) = χ'
proof
  define f where f = ECP.tuple (χ' J.Zero) ?g1 ?g0 (χ' J.Zero)
  have 4: e0 · f = χ' J.Zero
    using ECP.universal by (simp add: 3 e1-def eq f-def)
  have f: «f : a' → dom e0»
  proof −
    have a' = dom (χ' J.Zero)
      by (simp add: J.arr-char)
    thus ?thesis
      using 3 f-def e0-def g0 g1 ECP.tuple-in-hom ECP.pbdom-def by simp
  qed
  moreover have 5: D.cones-map f (D.mkCone e0) = χ'
  proof −
    have ⋀j. J.arr j ⟹ D.mkCone e0 j · f = χ' j
    proof −
      fix j
      assume j: J.arr j
      show D.mkCone e0 j · f = χ' j
      proof (cases j = J.Zero)
        case True
        moreover have e0 · f = χ' J.Zero
          using 4 by simp
        ultimately show ?thesis
          unfolding f-def D.mkCone-def comp-assoc
          using J.arr-char by simp
        next
        case F: False
        hence 1: (f0 · e0) · f = f0 · χ' J.Zero
          using 4 comp-assoc by simp
```

```
            also have ... = χ' j
              by (metis (no-types, lifting) F D.mkCone-cone D.mkCone-def
                  χ'.cone-axioms j)
            finally show ?thesis
              by (simp add: F D.mkCone-def j)
          qed
        qed
        thus ?thesis
          using f e0 χ.cone-axioms χ'.extensionality by auto
      qed
      ultimately show «f : a' → dom e0» ∧ D.cones-map f (D.mkCone e0) = χ'
        by simp
      fix f'
      assume f': «f' : a' → dom e0» ∧ D.cones-map f' (D.mkCone e0) = χ'
      show f' = f
      proof −
        have e0 · f' = χ' J.Zero
          using f' D.mkCone-cone D.mkCone-def χ'.cone-axioms
                comp-assoc J.arr-char χ.cone-axioms
          by auto
        thus ?thesis
          using f' 3 4 eq ECP.universal [of ?g1 ?g0 e1 · f' e0 · f'] e0-def e1-def
          by (metis (no-types, lifting))
      qed
    qed
  qed
  show has-as-equalizer f0 f1 e0
  proof −
    have par f0 f1
      by fact
    moreover have D.has-as-equalizer e0
      ..
    ultimately show ?thesis
      using has-as-equalizer-def by blast
  qed
 qed
qed

interpretation category-with-finite-products C
 by (simp add: ECC.is-cartesian-category cartesian-category.is-category-with-finite-products)

lemma has-finite-products:
shows category-with-finite-products C
  ..

lemma has-finite-limits:
shows category-with-finite-limits C
proof
```

```
    fix J :: nat comp
    assume J: category J
    interpret J: category J
      using J by simp
    assume finite: finite (Collect J.arr)
    show has-limits-of-shape J
    proof −
      have Collect (partial-composition.ide J) ⊆ Collect J.arr
        by auto
      hence 1: finite (Collect J.ide)
        using finite finite-subset by blast
      have has-products (Collect (partial-composition.ide J))
        using 1 J.ideD(1) J.not-arr-null ECC.has-finite-products by auto
      moreover have Collect (partial-composition.ide J) ≠ UNIV
        using J.not-arr-null by blast
      moreover have Collect (partial-composition.arr J) ≠ UNIV
        using J.not-arr-null by blast
      ultimately show ?thesis
        using finite 1 J.category-axioms has-limits-if-has-products
              ECC.has-finite-products' [of Collect J.ide]
              ECC.has-finite-products' [of Collect J.arr]
        by simp
    qed
  qed

  sublocale category-with-finite-limits C
    using has-finite-limits by simp

 end

end
```

Chapter 24

Cartesian Closed Category

theory *CartesianClosedCategory*
imports *CartesianCategory*
begin

A *cartesian closed category* is a cartesian category such that, for every object b, the functor *prod* - b is a left adjoint functor. A right adjoint to this functor takes each object c to the *exponential* *exp* b c. The adjunction yields a natural bijection between *hom* (*prod* a b) c and *hom* a (*exp* b c).

locale *cartesian-closed-category* =
cartesian-category +
assumes *left-adjoint-prod-ax*: $\bigwedge b.$ *ide* $b \Longrightarrow$ *left-adjoint-functor* C C ($\lambda x.$ *some-prod* x b)

locale *elementary-cartesian-closed-category* =
elementary-cartesian-category C *pr0 pr1 one trm*
for C :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ‹·› 55)
and *pr0* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹$\mathfrak{p}_0$[-, -]›)
and *pr1* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹$\mathfrak{p}_1$[-, -]›)
and *one* :: $'a$ (‹**1**›)
and *trm* :: $'a \Rightarrow 'a$ (‹t[-]›)
and *exp* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *eval* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *curry* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ +
assumes *eval-in-hom-ax*: ⟦ *ide* b; *ide* c ⟧ $\Longrightarrow$ «*eval* b c : *prod* (*exp* b c) $b \rightarrow c$»
and *ide-exp-ax* [*intro*]: ⟦ *ide* b; *ide* c ⟧ $\Longrightarrow$ *ide* (*exp* b c)
and *curry-in-hom*: ⟦ *ide* a; *ide* b; *ide* c; «g : *prod* a $b \rightarrow c$» ⟧
$\Longrightarrow$ «*curry* a b c g : $a \rightarrow$ *exp* b c»
and *uncurry-curry-ax*: ⟦ *ide* a; *ide* b; *ide* c; «g : *prod* a $b \rightarrow c$» ⟧
$\Longrightarrow$ *eval* b c · *prod* (*curry* a b c g) $b = g$
and *curry-uncurry-ax*: ⟦ *ide* a; *ide* b; *ide* c; «h : $a \rightarrow$ *exp* b c» ⟧
$\Longrightarrow$ *curry* a b c (*eval* b c · *prod* h b) = h

context *cartesian-closed-category*
begin

interpretation *elementary-cartesian-category C some-pr0 some-pr1* ‹$\mathbf{1}^?$› ‹$\lambda a.\ \mathrm{t}^?[a]$›
using *extends-to-elementary-cartesian-category* **by** *blast*

lemma *has-exponentials*:
assumes *ide b* **and** *ide c*
shows $\exists\, x\ e.\ ide\ x \wedge \langle\!\langle e : x \otimes^?\ b \to c\rangle\!\rangle \wedge$
$\quad (\forall\, a\ g.\ ide\ a \wedge \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle \longrightarrow (\exists!f.\ \langle\!\langle f : a \to x\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b)))$
proof −
interpret *F*: *left-adjoint-functor C C* ‹$\lambda x.\ x \otimes^?\ b$›
using *assms*(*1*) *left-adjoint-prod-ax* **by** *simp*
obtain *x e* **where** *e*: *terminal-arrow-from-functor C C* $(\lambda x.\ x \otimes^?\ b)$ *x c e*
using *assms F.ex-terminal-arrow* [*of c*] **by** *auto*
interpret *e*: *terminal-arrow-from-functor C C* ‹$\lambda x.\ x \otimes^?\ b$› *x c e*
using *e* **by** *simp*
have $\bigwedge a\ g.\ [\![\ ide\ a;\ \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle\ ]\!] \Longrightarrow \exists!f.\ \langle\!\langle f : a \to x\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b)$
using *e.is-terminal category-axioms F.functor-axioms*
unfolding *e.is-coext-def arrow-from-functor-def arrow-from-functor-axioms-def*
by *simp*
thus *?thesis*
using *e.arrow* **by** *metis*
qed

definition *some-exp* (‹$exp^?$›)
where *some-exp b c* $\equiv$ *SOME x. ide x* $\wedge$
$(\exists\, e.\ \langle\!\langle e : x \otimes^?\ b \to c\rangle\!\rangle \wedge$
$(\forall\, a\ g.\ ide\ a \wedge \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle$
$\longrightarrow (\exists!f.\ \langle\!\langle f : a \to x\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b))))$

definition *some-eval* (‹$eval^?$›)
where *some-eval b c* $\equiv$ *SOME e.* $\langle\!\langle e : exp^?\ b\ c \otimes^?\ b \to c\rangle\!\rangle \wedge$
$(\forall\, a\ g.\ ide\ a \wedge \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle$
$\longrightarrow (\exists!f.\ \langle\!\langle f : a \to exp^?\ b\ c\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b)))$

definition *some-Curry* (‹$Curry^?$›)
where *some-Curry a b c g* $\equiv$ *THE f.* $\langle\!\langle f : a \to exp^?\ b\ c\rangle\!\rangle \wedge g = eval^?\ b\ c \cdot (f \otimes^?\ b)$

lemma *Curry-uniqueness*:
assumes *ide b* **and** *ide c*
shows *ide* $(exp^?\ b\ c)$
and $\langle\!\langle eval^?\ b\ c : exp^?\ b\ c \otimes^?\ b \to c\rangle\!\rangle$
and $[\![\ ide\ a;\ \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle\ ]\!] \Longrightarrow$
$\exists!f.\ \langle\!\langle f : a \to exp^?\ b\ c\rangle\!\rangle \wedge g = eval^?\ b\ c \cdot (f \otimes^?\ b)$
using *assms some-exp-def some-eval-def has-exponentials*
someI-ex [*of* $\lambda x.\ ide\ x \wedge (\exists\, e.\ \langle\!\langle e : x \otimes^?\ b \to c\rangle\!\rangle \wedge$
$(\forall\, a\ g.\ ide\ a \wedge \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle$
$\longrightarrow (\exists!f.\ \langle\!\langle f : a \to x\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b))))$]
someI-ex [*of* $\lambda e.\ \langle\!\langle e : exp^?\ b\ c \otimes^?\ b \to c\rangle\!\rangle \wedge$
$(\forall\, a\ g.\ ide\ a \wedge \langle\!\langle g : a \otimes^?\ b \to c\rangle\!\rangle$
$\longrightarrow (\exists!f.\ \langle\!\langle f : a \to exp^?\ b\ c\rangle\!\rangle \wedge g = e \cdot (f \otimes^?\ b)))$]

by *auto*

lemma *ide-exp* [*intro*, *simp*]:
assumes *ide b* **and** *ide c*
shows *ide* (*exp*$^?$ *b c*)
using *assms Curry-uniqueness*(*1*) **by** *force*

lemma *eval-in-hom* [*intro*]:
assumes *ide b* **and** *ide c* **and** $x = exp^? \ b \ c \otimes^? \ b$
shows «*eval*$^?$ *b c* : $x \to c$»
using *assms Curry-uniqueness* **by** *simp*

lemma *Uncurry-Curry*:
assumes *ide a* **and** *ide b* **and** «$g : a \otimes^? \ b \to c$»
shows «*Curry*$^?$ *a b c g* : $a \to exp^? \ b \ c$» $\wedge$ $g = eval^? \ b \ c \cdot (Curry^? \ a \ b \ c \ g \otimes^? \ b)$
proof −
have *ide c*
using *assms*(*3*) **by** *auto*
thus *?thesis*
using *assms some-Curry-def Curry-uniqueness*
theI' [*of* $\lambda f.$ «$f : a \to exp^? \ b \ c$» $\wedge$ $g = eval^? \ b \ c \cdot (f \otimes^? \ b)$]
by *simp*
qed

lemma *Curry-Uncurry*:
assumes *ide b* **and** *ide c* **and** «$h : a \to exp^? \ b \ c$»
shows $Curry^? \ a \ b \ c \ (eval^? \ b \ c \cdot (h \otimes^? \ b)) = h$
proof −
have $\exists!f.$ «$f : a \to exp^? \ b \ c$» $\wedge$ $eval^? \ b \ c \cdot (h \otimes^? \ b) = eval^? \ b \ c \cdot (f \otimes^? \ b)$
proof −
have *ide a* $\wedge$ «$eval^? \ b \ c \cdot (h \otimes^? \ b) : (a \otimes^? \ b) \to c$»
proof (*intro conjI*)
show *ide a*
using *assms*(*3*) **by** *auto*
show «$eval^? \ b \ c \cdot (h \otimes^? \ b) : a \otimes^? \ b \to c$»
using *assms* **by** (*intro comp-in-homI*) *auto*
qed
thus *?thesis*
using *assms Curry-uniqueness* **by** *simp*
qed
moreover have «$h : a \to exp^? \ b \ c$» $\wedge$ $eval^? \ b \ c \cdot (h \otimes^? \ b) = eval^? \ b \ c \cdot (h \otimes^? \ b)$
using *assms* **by** *simp*
ultimately show *?thesis*
using *assms some-Curry-def Curry-uniqueness Uncurry-Curry*
the1-equality [*of* $\lambda f.$ «$f : a \to exp^? \ b \ c$» $\wedge$
$eval^? \ b \ c \cdot (h \otimes^? \ b) = eval^? \ b \ c \cdot (f \otimes^? \ b)$]
by *simp*
qed

lemma *Curry-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** «$g : a \otimes^? b \to c$»
shows «$Curry^? \ a \ b \ c \ g : a \to exp^? \ b \ c$»
  using *assms*
  by (*simp add*: *Uncurry-Curry*)

lemma *Curry-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** «$g : a \otimes^? b \to c$»
shows *arr* ($Curry^? \ a \ b \ c \ g$)
and *dom* ($Curry^? \ a \ b \ c \ g$) = *a*
and *cod* ($Curry^? \ a \ b \ c \ g$) = $exp^? \ b \ c$
  using *assms Curry-in-hom* **by** *blast*+

lemma *eval-simps* [*simp*]:
assumes *ide b* **and** *ide c* **and** $x = (exp^? \ b \ c) \otimes^? b$
shows *arr* ($eval^? \ b \ c$)
and *dom* ($eval^? \ b \ c$) = *x*
and *cod* ($eval^? \ b \ c$) = *c*
  using *assms eval-in-hom* **by** *auto*

interpretation *elementary-cartesian-closed-category C some-pr0 some-pr1*
        ‹$\mathbf{1}^?$› ‹$\lambda a.\ \mathrm{t}^?[a]$› *some-exp some-eval some-Curry*
  using *Curry-uniqueness Uncurry-Curry Curry-Uncurry*
  apply *unfold-locales* **by** *auto*

lemma *extends-to-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category C some-pr0 some-pr1*
      $\mathbf{1}^?$ ($\lambda a.\ \mathrm{t}^?[a]$) *some-exp some-eval some-Curry*
  ..

lemma *has-as-exponential*:
assumes *ide b* **and** *ide c*
shows *has-as-exponential b c* ($exp^? \ b \ c$) ($eval^? \ b \ c$)
proof
  show *ide b* **by** *fact*
  show *ide* ($exp^? \ b \ c$)
    using *assms* **by** *simp*
  show «*some-eval b c* : $exp^? \ b \ c \otimes^? b \to c$»
    using *assms* **by** *auto*
  show $\bigwedge a \ g.$ ⟦*ide a*; «$g : a \otimes^? b \to c$»⟧ ⟹
          $\exists!f.$ «$f : a \to exp^? \ b \ c$» ∧ $g = eval^? \ b \ c \cdot (f \otimes^? b)$
    by (*simp add*: *assms Curry-uniqueness*(*3*))
qed

lemma *has-as-exponential-iff*:
shows *has-as-exponential b c x e* ⟷
      ide b ∧ «$e : x \otimes^? b \to c$» ∧
      ($\exists h.$ «$h : x \to exp^? \ b \ c$» ∧ $e = eval^? \ b \ c \cdot (h \otimes^? b)$ ∧ *iso h*)
proof

assume *1*: *has-as-exponential b c x e*
moreover have *2*: *has-as-exponential b c* (*exp*$^?$ *b c*) (*eval*$^?$ *b c*)
using *1 ide-cod has-as-exponential-def in-homE*
by (*metis has-as-exponential*)
ultimately show *ide b* $\wedge$ «*e* : *x* $\otimes^?$ *b* $\to$ *c*» $\wedge$
($\exists$ *h*. «*h* : *x* $\to$ *exp*$^?$ *b c*» $\wedge$ *e* = *eval*$^?$ *b c* $\cdot$ (*h* $\otimes^?$ *b*) $\wedge$ *iso h*)
by (*metis exponentials-are-isomorphic*(*2*) *has-as-exponentialE*)
next
assume *1*: *ide b* $\wedge$ «*e* : *x* $\otimes^?$ *b* $\to$ *c*» $\wedge$
($\exists$ *h*. «*h* : *x* $\to$ *exp*$^?$ *b c*» $\wedge$ *e* = *eval*$^?$ *b c* $\cdot$ (*h* $\otimes^?$ *b*) $\wedge$ *iso h*)
have *c*: *ide c*
using *1 ide-cod in-homE* **by** *metis*
have *2*: *has-as-exponential b c* (*exp*$^?$ *b c*) (*eval*$^?$ *b c*)
by (*simp add*: *1 c eval-in-hom-ax Curry-uniqueness*(*3*) *has-as-exponential-def*)
obtain *h* **where** *h*: «*h* : *x* $\to$ *exp*$^?$ *b c*» $\wedge$ *e* = *eval*$^?$ *b c* $\cdot$ (*h* $\otimes^?$ *b*) $\wedge$ *iso h*
using *1* **by** *blast*
show *has-as-exponential b c x e*
proof (*unfold has-as-exponential-def*, *intro conjI*)
show *ide b* **and** *ide x* **and** «*e* : *x* $\otimes^?$ *b* $\to$ *c*»
using *1 h ide-dom* **by** *blast*+
show $\forall$ *y g*. *ide y* $\wedge$ «*g* : *y* $\otimes^?$ *b* $\to$ *c*» $\longrightarrow$ ($\exists!$*f*. «*f* : *y* $\to$ *x*» $\wedge$ *g* = *e* $\cdot$ (*f* $\otimes^?$ *b*))
proof (*intro allI impI*)
fix *y g*
assume *3*: *ide y* $\wedge$ «*g* : *y* $\otimes^?$ *b* $\to$ *c*»
obtain *k* **where** *k*: «*k* : *y* $\to$ *exp*$^?$ *b c*» $\wedge$ *g* = *eval*$^?$ *b c* $\cdot$ (*k* $\otimes^?$ *b*)
by (*metis 3* ‹*ide b*› *c Curry-uniqueness*(*3*))
show $\exists!$*f*. «*f* : *y* $\to$ *x*» $\wedge$ *g* = *e* $\cdot$ (*f* $\otimes^?$ *b*)
proof −
let *?f* = *inv h* $\cdot$ *k*
have *f*: «*?f* : *y* $\to$ *x*»
by (*meson comp-in-homI inv-in-hom h k*)
moreover have *g* = *e* $\cdot$ (*?f* $\otimes^?$ *b*)
proof −
have *e* $\cdot$ *some-prod ?f b* = *e* $\cdot$ *some-prod* (*inv h* $\cdot$ *k*) (*b* $\cdot$ *b*)
by (*simp add*: *1*)
also have ... = *e* $\cdot$ (*inv h* $\otimes^?$ *b*) $\cdot$ (*k* $\otimes^?$ *b*)
by (*metis* ‹*ide b*› *f arrI comp-ide-self interchange ide-compE*)
also have ... = (*e* $\cdot$ (*inv h* $\otimes^?$ *b*)) $\cdot$ (*k* $\otimes^?$ *b*)
using *comp-assoc* **by** *simp*
also have ... = *eval*$^?$ *b c* $\cdot$ (*k* $\otimes^?$ *b*)
by (*metis* ‹«*e* : *x* $\otimes^?$ *b* $\to$ *c*»› *h* ‹*ide b*› *arrI inv-prod*(*1*−*2*) *ide-is-iso*
inv-ide invert-side-of-triangle(*2*))
also have ... = *g*
using *k* **by** *blast*
finally show *?thesis* **by** *blast*
qed
moreover have $\bigwedge$*f'*. «*f'* : *y* $\to$ *x*» $\wedge$ *g* = *e* $\cdot$ (*f'* $\otimes^?$ *b*) $\Longrightarrow$ *f'* = *?f*
proof −
fix *f'*

```
          assume f': «f' : y → x» ∧ g = e · (f' ⊗? b)
          have «h · f' : y → exp? b c» ∧ g = eval? b c · (h · f' ⊗? b)
            using f' h ‹ide b› comp-assoc interchange seqI' by fastforce
          hence C h f' = C h ?f
            by (metis ‹ide b› arrI c h k Curry-Uncurry invert-side-of-triangle(1))
          thus f' = ?f
            using f h iso-cancel-left by auto
        qed
        ultimately show ?thesis by blast
      qed
    qed
  qed
qed

end

context elementary-cartesian-closed-category
begin

  lemma left-adjoint-prod:
  assumes ide b
  shows left-adjoint-functor C C (λx. x ⊗ b)
  proof −
    interpret functor C C ‹λx. x ⊗ b›
      using assms interchange
      apply unfold-locales
        apply auto
      using prod-def tuple-def
      by auto
    interpret left-adjoint-functor C C ‹λx. x ⊗ b›
    proof
      show ⋀c. ide c ⟹ ∃ x e. terminal-arrow-from-functor C C (λx. x ⊗ b) x c e
      proof −
        fix c
        assume c: ide c
        show ∃ x e. terminal-arrow-from-functor C C (λx. x ⊗ b) x c e
        proof (intro exI)
          interpret arrow-from-functor C C ‹λx. x ⊗ b› ‹exp b c› c ‹eval b c›
            using assms c eval-in-hom-ax
            by (unfold-locales, auto)
          show terminal-arrow-from-functor C C (λx. x ⊗ b) (exp b c) c (eval b c)
          proof
            show ⋀a f. arrow-from-functor C C (λx. x ⊗ b) a c f ⟹
                       ∃!g. arrow-from-functor.is-coext C C
                              (λx. x ⊗ b) (exp b c) (eval b c) a f g
            proof −
              fix a f
              assume f: arrow-from-functor C C (λx. x ⊗ b) a c f
              interpret f: arrow-from-functor C C ‹λx. x ⊗ b› a c f
```

```
            using f by simp
          show ∃!g. is-coext a f g
          proof
            have a: ide a
              using f.arrow by simp
            show is-coext a f (curry a b c f)
              unfolding is-coext-def
              using assms a c curry-in-hom uncurry-curry-ax f.arrow by simp
            show ⋀g. is-coext a f g ⟹ g = curry a b c f
              unfolding is-coext-def
              using assms a c curry-uncurry-ax f.arrow by simp
          qed
        qed
      qed
    qed
  qed
  show ?thesis ..
qed

sublocale cartesian-category C
  using is-cartesian-category by simp

sublocale cartesian-closed-category C
proof −
  interpret CCC: elementary-cartesian-category
                 C some-pr0 some-pr1 some-terminal some-terminator
    using extends-to-elementary-cartesian-category by blast
  show cartesian-closed-category C
  proof
    fix b
    assume b: ide b
    interpret left-adjoint-functor C C ‹λx. CCC.prod x b›
    proof −

      have naturally-isomorphic C C (λx. x ⊗ b) (λx. CCC.prod x b)
      proof −
        interpret CC: product-category C C ..
        interpret X: binary-functor C C C ‹λfg. fst fg ⊗ snd fg›
          using binary-functor-Prod(1) by auto
        interpret Xb: functor C C ‹λx. x ⊗ b›
          using b X.fixing-ide-gives-functor-2 by simp
        interpret prod: binary-functor C C C ‹λfg. CCC.prod (fst fg) (snd fg)›
          using CCC.binary-functor-Prod(1) by simp
        interpret prod-b: functor C C ‹λx. CCC.prod x b›
          using b prod.fixing-ide-gives-functor-2 by simp
        interpret φ: transformation-by-components C C ‹λx. x ⊗ b› ‹λx. CCC.prod x b›
                     ‹λa. CCC.tuple 𝔭₁[a, b] 𝔭₀[a, b]›
          using b CCC.prod-tuple by unfold-locales auto
```

```
        interpret φ: natural-isomorphism C C ‹λx. x ⊗ b› ‹λx. CCC.prod x b› φ.map
        proof
          fix a
          assume a: ide a
          show iso (φ.map a)
          proof
            show inverse-arrows (φ.map a) ⟨some-pr1 a b, some-pr0 a b⟩
              using a b by auto
          qed
        qed
        show ?thesis
          using naturally-isomorphic-def φ.natural-isomorphism-axioms by blast
      qed
      moreover have left-adjoint-functor C C (λx. x ⊗ b)
        using b left-adjoint-prod by simp
      ultimately show left-adjoint-functor C C (λx. CCC.prod x b)
        using left-adjoint-functor-respects-naturally-isomorphic by auto
    qed
    show ⋀f. ¬ arr f ⟹ some-prod f b = null
      using extensionality by blast
    show ⋀g f. seq g f ⟹ some-prod (g · f) b = some-prod g b · some-prod f b
      by simp
    show ⋀y. ide y ⟹ ∃x e. terminal-arrow-from-functor (·) (·) (λx. some-prod x b) x y e
      using ex-terminal-arrow by simp
  qed auto
qed

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

end

end
```

Chapter 25

The Category of Hereditarily Finite Sets

theory *HF-SetCat*
imports *CategoryWithFiniteLimits CartesianClosedCategory HereditarilyFinite.HF*
begin

This theory constructs a category whose objects are in bijective correspondence with the hereditarily finite sets and whose arrows correspond to the functions between such sets. We show that this category is cartesian closed and has finite limits. Note that up to this point we have not constructed any other interpretation for the *cartesian-closed-category* locale, but it is important to have one to ensure that the locale assumptions are consistent.

25.1 Preliminaries

We begin with some preliminary definitions and facts about hereditarily finite sets, which are better targeted toward what we are trying to do here than what already exists in *HereditarilyFinite.HF*.

The following defines when a hereditarily finite set F represents a function from a hereditarily finite set B to a hereditarily finite set C. Specifically, F must be a relation from B to C, whose domain is B, whose range is contained in C, and which is single-valued on its domain.

definition *hfun*
where *hfun B C F* $\equiv F \leq B * C \wedge$ *hfunction F* $\wedge$ *hdomain F* $= B \wedge$ *hrange F* $\leq C$

lemma *hfunI* [*intro*]:
assumes $F \leq A * B$
and $\bigwedge X.\ X \in A \Longrightarrow \exists! Y.\ \langle X,\ Y\rangle \in F$
and $\bigwedge X\ Y.\ \langle X,\ Y\rangle \in F \Longrightarrow Y \in B$
shows *hfun A B F*
unfolding *hfun-def*

using *assms hfunction-def hrelation-def is-hpair-def hrange-def hconverse-def hdomain-def*
apply (*intro conjI*)
apply *auto*
by *fast*

lemma *hfunE* [*elim*]:
assumes *hfun B C F*
and $(\bigwedge Y.\ Y \in B \Longrightarrow (\exists!Z.\ \langle Y,\ Z\rangle \in F) \wedge (\forall Z.\ \langle Y,\ Z\rangle \in F \longrightarrow Z \in C)) \Longrightarrow T$
shows *T*
proof −
have $\bigwedge Y.\ Y \in B \Longrightarrow (\exists!Z.\ \langle Y,\ Z\rangle \in F) \wedge (\forall Z.\ \langle Y,\ Z\rangle \in F \longrightarrow Z \in C)$
proof (*intro allI impI conjI*)
fix *Y*
assume *Y*: $Y \in B$
show $\exists!Z.\ \langle Y,\ Z\rangle \in F$
proof −
have $\exists Z.\ \langle Y,\ Z\rangle \in F$
using *assms Y hfun-def hdomain-def* **by** *auto*
moreover have $\bigwedge Z\ Z'.\ [\![\ \langle Y,\ Z\rangle \in F;\ \langle Y,\ Z'\rangle \in F\ ]\!] \Longrightarrow Z = Z'$
using *assms hfun-def hfunction-def* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed
show $\bigwedge Z.\ \langle Y,\ Z\rangle \in F \Longrightarrow Z \in C$
using *assms Y hfun-def* **by** *auto*
qed
thus *?thesis*
using *assms*(*2*) **by** *simp*
qed

The hereditarily finite set *hexp B C* represents the collection of all functions from *B* to *C*.

definition *hexp*
where $hexp\ B\ C = \{\!|F \in HPow\ (B * C).\ hfun\ B\ C\ F|\!\}$

lemma *hfun-in-hexp*:
assumes *hfun B C F*
shows $F \in hexp\ B\ C$
using *assms* **by** (*simp add*: *hexp-def hfun-def*)

The function *happ* applies a function *F* from *B* to *C* to an element of *B*, yielding an element of *C*.

abbreviation *happ*
where $happ \equiv app$

lemma *happ-mapsto*:
assumes $F \in hexp\ B\ C$ **and** $Y \in B$
shows $happ\ F\ Y \in C$ **and** $happ\ F\ Y \in hrange\ F$
proof −
show $happ\ F\ Y \in C$

```
    using assms app-def hexp-def app-equality hdomain-def hfun-def by auto
  show happ F Y ∈ hrange F
  proof −
    have ⟨Y, happ F Y⟩ ∈ F
      using assms app-def hexp-def app-equality hdomain-def hfun-def by auto
    thus ?thesis
      using hdomain-def hrange-def hconverse-def by auto
  qed
qed

lemma happ-expansion:
assumes hfun B C F
shows F = {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖}
proof
  fix XY
  show XY ∈ F ⟷ XY ∈ {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖}
  proof
    show XY ∈ F ⟹ XY ∈ {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖}
    proof −
      assume XY: XY ∈ F
      have XY ∈ B ∗ C
        using assms XY hfun-def by auto
      moreover have hsnd XY = happ F (hfst XY)
        using assms XY hfunE app-def [of F hfst XY] the1-equality [of λy. ⟨hfst XY, y⟩ ∈ F]
            calculation
        by auto
      ultimately show XY ∈ {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖} by simp
    qed
    show XY ∈ {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖} ⟹ XY ∈ F
    proof −
      assume XY: XY ∈ {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖}
      show XY ∈ F
        using assms XY app-def [of F hfst XY] the1-equality [of λy. ⟨hfst XY, y⟩ ∈ F]
        by fastforce
    qed
  qed
qed
```

Function *hlam* takes a function F from $A * B$ to C to a function *hlam F* from A to *hexp B C*.

```
definition hlam
where hlam A B C F =
      {‖XG ∈ A ∗ hexp B C.
        ∀ YZ. YZ ∈ hsnd XG ⟷ is-hpair YZ ∧ ⟨⟨hfst XG, hfst YZ⟩, hsnd YZ⟩ ∈ F‖}

lemma hfun-hlam:
assumes hfun (A ∗ B) C F
shows hfun A (hexp B C) (hlam A B C F)
proof
```

```
  show hlam A B C F ≤ A * hexp B C
    using assms hlam-def by auto
  show ⋀X. X ∈ A ⟹ ∃!Y. ⟨X, Y⟩ ∈ hlam A B C F
  proof
    fix X
    assume X: X ∈ A
    let ?G = {|YZ ∈ B * C. ⟨⟨X, hfst YZ⟩, hsnd YZ⟩ ∈ F|}
    have 1: ?G ∈ hexp B C
      using assms X hexp-def by fastforce
    show ⟨X, ?G⟩ ∈ hlam A B C F
      using assms X 1 is-hpair-def hfun-def hlam-def by auto
    fix Y
    assume XY: ⟨X, Y⟩ ∈ hlam A B C F
    show Y = ?G
      using assms X XY hlam-def hexp-def by fastforce
  qed
  show ⋀X Y. ⟨X, Y⟩ ∈ hlam A B C F ⟹ Y ∈ hexp B C
    using assms hlam-def hexp-def by simp
qed

lemma happ-hlam:
assumes X ∈ A and hfun (A * B) C F
shows ∃!G. ⟨X, G⟩ ∈ hlam A B C F
and happ (hlam A B C F) X = (THE G. ⟨X, G⟩ ∈ hlam A B C F)
and happ (hlam A B C F) X = {|yz ∈ B * C. ⟨⟨X, hfst yz⟩, hsnd yz⟩ ∈ F|}
and Y ∈ B ⟹ happ (happ (hlam A B C F) X) Y = happ F ⟨X, Y⟩
proof −
  show 1: ∃!G. ⟨X, G⟩ ∈ hlam A B C F
    using assms(1,2) hfun-hlam hfunE
    by (metis (full-types))
  show 2: happ (hlam A B C F) X = (THE G. ⟨X, G⟩ ∈ hlam A B C F)
    using assms app-def by simp
  show happ (happ (hlam A B C F) X) Y = happ F ⟨X, Y⟩
  proof −
    have 3: ⟨X, happ (hlam A B C F) X⟩ ∈ hlam A B C F
      using assms(1) 1 2 theI' [of λG. ⟨X, G⟩ ∈ hlam A B C F] by simp
    hence ∃!Z. happ (happ (hlam A B C F) X) = Z
      by simp
    moreover have happ (happ (hlam A B C F) X) Y = happ F ⟨X, Y⟩
      using assms(1−2) 3 hlam-def is-hpair-def app-def by simp
    ultimately show ?thesis by simp
  qed
  show happ (hlam A B C F) X = {|YZ ∈ B * C. ⟨⟨X, hfst YZ⟩, hsnd YZ⟩ ∈ F|}
  proof −
    let ?G = {|YZ ∈ B * C. ⟨⟨X, hfst YZ⟩, hsnd YZ⟩ ∈ F|}
    have 4: hfun B C ?G
    proof
      show {|YZ ∈ B * C. ⟨⟨X, hfst YZ⟩, hsnd YZ⟩ ∈ F|} ≤ B * C
        using assms by auto
```

show $\bigwedge Y.\ Y \in B \Longrightarrow \exists!Z.\ \langle Y,\ Z\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\}$
proof −
fix Y
assume Y: $Y \in B$
have XY: $\langle X,\ Y\rangle \in A * B$
using *assms Y* **by** *simp*
hence *1*: $\exists!Z.\ \langle\langle X,\ Y\rangle,\ Z\rangle \in F$
using *assms XY hfunE* [*of A * B C F*] **by** *metis*
obtain Z **where** Z: $\langle\langle X,\ Y\rangle,\ Z\rangle \in F$
using *1* **by** *auto*
have $\exists Z.\ \langle Y,\ Z\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\}$
proof −
have $\langle Y,\ Z\rangle \in B * C$
using *assms Y Z* **by** *blast*
moreover have $\langle\langle X,\ hfst\ \langle Y,\ Z\rangle\rangle,\ hsnd\ \langle Y,\ Z\rangle\rangle \in F$
using *assms Y Z* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed
moreover have $\bigwedge Z\ Z'.\ [\![\langle Y,\ Z\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\};$
$\langle Y,\ Z'\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\}]\!] \Longrightarrow Z = Z'$
using *assms Y* **by** *auto*
ultimately show $\exists!Z.\ \langle Y,\ Z\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\}$
by *auto*
qed
show $\bigwedge Y\ Z.\ \langle Y,\ Z\rangle \in \{\!|\, YZ \in B * C.\ \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F \,|\!\} \Longrightarrow Z \in C$
using *assms* **by** *simp*
qed
have $\langle X,\ ?G\rangle \in hlam\ A\ B\ C\ F$
proof −
have $\langle X,\ ?G\rangle \in A * hexp\ B\ C$
using *assms 4*
by (*simp add*: *hfun-in-hexp*)
moreover have $\forall\ YZ.\ YZ \in ?G \longleftrightarrow is\text{-}hpair\ YZ \wedge \langle\langle X,\ hfst\ YZ\rangle,\ hsnd\ YZ\rangle \in F$
using *assms 1 is-hpair-def hfun-def* **by** *auto*
ultimately show *?thesis*
using *assms 1 hlam-def* **by** *simp*
qed
thus *happ* (*hlam A B C F*) $X = ?G$
using *assms 2 4 app-equality hfun-def hfun-hlam* **by** *auto*
qed
qed

25.2 Construction of the Category

locale *hfsetcat*
begin

We construct the category of hereditarily finite sets and functions simply by applying the generic "set category" construction, using the hereditarily finite sets as the universe,

and constraining the collections of such sets that determine objects of the category to those that are finite.

```
interpretation setcat ‹TYPE(hf)› finite
  using finite-subset
  by unfold-locales blast+
interpretation set-category comp ‹λA. A ⊆ Collect terminal ∧ finite (elem-of ‘ A)›
  using is-set-category by blast

lemma set-ide-char:
shows A ∈ set ‘ Collect ide ⟷ A ⊆ Univ ∧ finite A
proof
  assume A: A ∈ set ‘ Collect ide
  show A ⊆ Univ ∧ finite A
  proof
    show A ⊆ Univ
      using A setp-set′ by auto
    obtain a where a: ide a ∧ A = set a
      using A by blast
    have finite (elem-of ‘ set a)
      using a setp-set-ide by blast
    moreover have inj-on elem-of (set a)
    proof −
      have inj-on elem-of Univ
        using bij-elem-of bij-betw-imp-inj-on by auto
      moreover have set a ⊆ Univ
        using a setp-set′ [of a] by blast
      ultimately show ?thesis
        using inj-on-subset by auto
    qed
    ultimately show finite A
      using a A finite-imageD [of elem-of set a] by blast
  qed
  next
  assume A: A ⊆ Univ ∧ finite A
  have ide (mkIde A)
    using A ide-mkIde by simp
  moreover have set (mkIde A) = A
    using A finite-imp-setp set-mkIde by presburger
  ultimately show A ∈ set ‘ Collect ide by blast
qed

lemma set-ideD:
assumes ide a
shows set a ⊆ Univ and finite (set a)
  using assms set-ide-char by auto

lemma ide-mkIdeI [intro]:
assumes A ⊆ Univ and finite A
shows ide (mkIde A) and set (mkIde A) = A
```

using *assms ide-mkIde set-mkIde* **by** *auto*

interpretation *category-with-terminal-object comp*
using *terminal-unity* **by** *unfold-locales auto*

We verify that the objects of HF are indeed in bijective correspondence with the hereditarily finite sets.

definition *ide-to-hf*
where *ide-to-hf a* = *HF* (*elem-of* ‘ *set a*)

definition *hf-to-ide*
where *hf-to-ide x* = *mkIde* (*arr-of* ‘ *hfset x*)

lemma *ide-to-hf-mapsto*:
shows *ide-to-hf* ∈ *Collect ide* → *UNIV*
by *simp*

lemma *hf-to-ide-mapsto*:
shows *hf-to-ide* ∈ *UNIV* → *Collect ide*
proof
fix *x* :: *hf*
have *finite* (*arr-of* ‘ *hfset x*)
by *simp*
moreover have *arr-of* ‘ *hfset x* ⊆ *Univ*
by (*metis* (*mono-tags*, *lifting*) *UNIV-I bij-arr-of bij-betw-def imageE image-eqI subsetI*)
ultimately have *ide* (*mkIde* (*arr-of* ‘ *hfset x*))
using *finite-imp-setp ide-mkIde* **by** *presburger*
thus *hf-to-ide x* ∈ *Collect ide*
using *hf-to-ide-def* **by** *simp*
qed

lemma *hf-to-ide-ide-to-hf*:
assumes *a* ∈ *Collect ide*
shows *hf-to-ide* (*ide-to-hf a*) = *a*
proof −
have *hf-to-ide* (*ide-to-hf a*) = *mkIde* (*arr-of* ‘ *hfset* (*HF* (*elem-of* ‘ *set a*)))
using *hf-to-ide-def ide-to-hf-def* **by** *simp*
also have ... = *a*
proof −
have *mkIde* (*arr-of* ‘ *hfset* (*HF* (*elem-of* ‘ *set a*))) = *mkIde* (*arr-of* ‘ *elem-of* ‘ *set a*)
proof −
have *finite* (*set a*)
using *assms set-ide-char* **by** *blast*
hence *finite* (*elem-of* ‘ *set a*)
by *simp*
hence *hfset* (*HF* (*elem-of* ‘ *set a*)) = *elem-of* ‘ *set a*
using *hfset-HF* [*of elem-of* ‘ *set a*] **by** *simp*
thus *?thesis* **by** *simp*
qed

```
    also have ... = a
    proof −
      have set a ⊆ Univ
        using assms set-ide-char by blast
      hence ⋀x. x ∈ set a ⟹ arr-of (elem-of x) = x
        using assms by auto
      hence arr-of ‘ elem-of ‘ set a = set a
        by force
      thus ?thesis
        using assms ide-char mkIde-set by simp
    qed
    finally show ?thesis by blast
  qed
  finally show hf-to-ide (ide-to-hf a) = a by blast
qed

lemma ide-to-hf-hf-to-ide:
assumes x ∈ UNIV
shows ide-to-hf (hf-to-ide x) = x
proof −
  have HF (elem-of ‘ set (mkIde (arr-of ‘ hfset x))) = x
  proof −
    have HF (elem-of ‘ set (mkIde (arr-of ‘ hfset x))) = HF (elem-of ‘ arr-of ‘ hfset x)
      using assms set-mkIde [of arr-of ‘ hfset x] arr-of-mapsto mkIde-def by auto
    also have ... = HF (hfset x)
    proof −
      have ⋀A. elem-of ‘ arr-of ‘ A = A
        using elem-of-arr-of by force
      thus ?thesis by metis
    qed
    also have ... = x by simp
    finally show ?thesis by blast
  qed
  thus ?thesis
    using assms ide-to-hf-def hf-to-ide-def by simp
qed

lemma bij-betw-ide-hf-set:
shows bij-betw ide-to-hf (Collect ide) (UNIV :: hf set)
  using ide-to-hf-mapsto hf-to-ide-mapsto ide-to-hf-hf-to-ide hf-to-ide-ide-to-hf
  by (intro bij-betwI) auto

lemma ide-implies-finite-set:
assumes ide a
shows finite (set a) and finite (hom unity a)
proof −
  show 1: finite (set a)
    using assms set-ide-char by blast
  show finite (hom unity a)
```

using *assms 1 bij-betw-points-and-set finite-imageD inj-img set-def* **by** *auto*
qed

We establish the connection between the membership relation defined for hereditarily finite sets and the corresponding membership relation associated with the set category.

lemma *arr-of-membI* [*intro*]:
assumes $x \in$ *ide-to-hf a*
shows *arr-of x* $\in$ *set a*
proof −
let *?X* = *inv-into* (*set a*) *elem-of x*
have *x* = *elem-of ?X* $\wedge$ *?X* $\in$ *set a*
using *assms*
by (*simp add*: *f-inv-into-f ide-to-hf-def inv-into-into*)
thus *?thesis*
by (*metis* (*no-types*, *lifting*) *arr-of-elem-of elem-set-implies-incl-in*
elem-set-implies-set-eq-singleton incl-in-def mem-Collect-eq terminal-char2)
qed

lemma *elem-of-membI* [*intro*]:
assumes *ide a* **and** *x* $\in$ *set a*
shows *elem-of x* $\in$ *ide-to-hf a*
proof −
have *finite* (*elem-of* ‘ *set a*)
using *assms ide-implies-finite-set* [*of a*] **by** *simp*
hence *elem-of x* $\in$ *hfset* (*ide-to-hf a*)
using *assms ide-to-hf-def hfset-HF* [*of elem-of* ‘ *set a*] **by** *simp*
thus *?thesis*
using *hmem-def* **by** *blast*
qed

We show that each hom-set *hom a b* is in bijective correspondence with the elements of the hereditarily finite set *hfun* (*ide-to-hf a*) (*ide-to-hf b*).

definition *arr-to-hfun*
where *arr-to-hfun f* = {⟦*XY* $\in$ *ide-to-hf* (*dom f*) $*$ *ide-to-hf* (*cod f*).
hsnd XY = *elem-of* (*Fun f* (*arr-of* (*hfst XY*)))⟧}

definition *hfun-to-arr*
where *hfun-to-arr B C F* =
mkArr (*arr-of* ‘ *hfset B*) (*arr-of* ‘ *hfset C*) (λ*x*. *arr-of* (*happ F* (*elem-of x*)))

lemma *hfun-arr-to-hfun*:
assumes *arr f*
shows *hfun* (*ide-to-hf* (*dom f*)) (*ide-to-hf* (*cod f*)) (*arr-to-hfun f*)
proof
show *arr-to-hfun f* $\leq$ *ide-to-hf* (*dom f*) $*$ *ide-to-hf* (*cod f*)
using *assms arr-to-hfun-def* **by** *auto*
show $\bigwedge X.\ X \in$ *ide-to-hf* (*dom f*) $\Longrightarrow \exists!Y.\ \langle X,\ Y\rangle \in$ *arr-to-hfun f*
proof
fix *X*

```
      assume X: X ∈ ide-to-hf (dom f)
      show ⟨X, elem-of (Fun f (arr-of X))⟩ ∈ arr-to-hfun f
      proof −
        have ⟨X, elem-of (Fun f (arr-of X))⟩ ∈ ⦃XY ∈ ide-to-hf (dom f) ∗ ide-to-hf (cod f).
                  hsnd XY = elem-of (Fun f (arr-of (hfst XY)))⦄
        proof −
          have hsnd ⟨X, elem-of (Fun f (arr-of X))⟩ =
                elem-of (Fun f (arr-of (hfst ⟨X, elem-of (Fun f (arr-of X))⟩)))
            using assms X by simp
          moreover have ⟨X, elem-of (Fun f (arr-of X))⟩ ∈ ide-to-hf (dom f) ∗ ide-to-hf (cod
f)
          proof −
            have elem-of (Fun f (arr-of X)) ∈ ide-to-hf (cod f)
            proof (intro elem-of-membI)
              show ide (cod f)
                using assms ide-cod by simp
              show Fun f (arr-of X) ∈ Cod f
                using assms X Fun-mapsto arr-of-membI by auto
            qed
            thus ?thesis
              using X by simp
          qed
          ultimately show ?thesis by simp
        qed
        thus ?thesis
          using arr-to-hfun-def by simp
      qed
      fix Y
      assume XY: ⟨X, Y⟩ ∈ arr-to-hfun f
      show Y = elem-of (Fun f (arr-of X))
        using assms X XY arr-to-hfun-def by auto
    qed
    show ⋀X Y. ⟨X, Y⟩ ∈ arr-to-hfun f ⟹ Y ∈ ide-to-hf (cod f)
      using assms arr-to-hfun-def ide-to-hf-def
            ‹arr-to-hfun f ≤ ide-to-hf (dom f) ∗ ide-to-hf (cod f)›
      by blast
  qed

  lemma arr-to-hfun-in-hexp:
  assumes arr f
  shows arr-to-hfun f ∈ hexp (ide-to-hf (dom f)) (ide-to-hf (cod f))
    using assms arr-to-hfun-def hfun-arr-to-hfun hexp-def by auto

  lemma hfun-to-arr-in-hom:
  assumes hfun B C F
  shows «hfun-to-arr B C F : hf-to-ide B → hf-to-ide C»
  proof
    let ?f = mkArr (arr-of ‘ hfset B) (arr-of ‘ hfset C) (λx. arr-of (happ F (elem-of x)))
    have 0: arr ?f
```

```
  proof −
    have arr-of ‘ hfset B ⊆ Univ ∧ arr-of ‘ hfset C ⊆ Univ
      using arr-of-mapsto by auto
    moreover have (λx. arr-of (happ F (elem-of x))) ∈ arr-of ‘ hfset B → arr-of ‘ hfset C
    proof
      fix x
      assume x: x ∈ arr-of ‘ hfset B
      have happ F (elem-of x) ∈ hfset C
        using assms x happ-mapsto hfun-in-hexp
        by (metis elem-of-arr-of HF-hfset finite-hfset hmem-HF-iff imageE)
      thus arr-of (happ F (elem-of x)) ∈ arr-of ‘ hfset C
        by simp
    qed
    ultimately show ?thesis
      using arr-mkArr
      by (meson finite-hfset finite-iff-ordLess-natLeq finite-imageI)
  qed
  show 1: arr (hfun-to-arr B C F)
    using 0 hfun-to-arr-def by simp
  show dom (hfun-to-arr B C F) = hf-to-ide B
    using 1 hfun-to-arr-def hf-to-ide-def dom-mkArr by auto
  show cod (hfun-to-arr B C F) = hf-to-ide C
    using 1 hfun-to-arr-def hf-to-ide-def cod-mkArr by auto
qed
```

The comprehension notation from *HereditarilyFinite.HF* interferes in an unfortunate way with the restriction notation from *HOL−Library.FuncSet*, making it impossible to use both in the present context.

```
lemma Fun-char:
assumes arr f
shows Fun f = restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f)
proof
  fix x
  show Fun f x = restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f) x
  proof (cases x ∈ Dom f)
    show x ∉ Dom f ⟹ ?thesis
      using assms Fun-mapsto Fun-def restrict-apply by simp
    show x ∈ Dom f ⟹ ?thesis
    proof −
      assume x: x ∈ Dom f
      have 1: hfun (ide-to-hf (dom f)) (ide-to-hf (cod f)) (arr-to-hfun f)
        using assms app-def arr-to-hfun-def hfun-arr-to-hfun
              the1-equality [of λy. ⟨elem-of x, y⟩ ∈ arr-to-hfun f elem-of (Fun f x)]
        by simp
      have 2: ∃!Y. ⟨elem-of x, Y⟩ ∈ arr-to-hfun f
        using assms x 1 hfunE elem-of-membI ide-dom
        by (metis (no-types, lifting))
      have Fun f x = arr-of (elem-of (Fun f x))
      proof −
```

```
      have Fun f x ∈ Univ
        using assms x ide-cod Fun-mapsto [of f] set-ide-char by blast
      thus ?thesis
        using arr-of-elem-of by simp
    qed
    also have ... = arr-of (happ (arr-to-hfun f) (elem-of x))
    proof −
      have ⟨elem-of x, elem-of (Fun f x)⟩ ∈ arr-to-hfun f
      proof −
        have ⟨elem-of x, elem-of (Fun f x)⟩ ∈ ide-to-hf (dom f) ∗ ide-to-hf (cod f)
          using assms x ide-dom ide-cod Fun-mapsto by fast
        moreover have elem-of (Fun f x) = elem-of (Fun f (arr-of (elem-of x)))
          by (metis (no-types, lifting) arr-of-elem-of setp-set-ide assms ide-dom subsetD x)
        ultimately show ?thesis
          using arr-to-hfun-def by auto
      qed
      moreover have ⟨elem-of x, happ (arr-to-hfun f) (elem-of x)⟩ ∈ arr-to-hfun f
        using assms x 1 2 app-equality hfun-def by blast
      ultimately show ?thesis
        using 2 by fastforce
    qed
    also have ... = restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f) x
      using assms x ide-dom by auto
    finally show ?thesis by simp
  qed
 qed
qed

lemma Fun-hfun-to-arr:
assumes hfun B C F
shows Fun (hfun-to-arr B C F) = restrict (λx. arr-of (happ F (elem-of x))) (arr-of ‘ hfset
B)
proof −
  have arr (hfun-to-arr B C F)
    using assms hfun-to-arr-in-hom by blast
  hence arr (mkArr (arr-of ‘ hfset B) (arr-of ‘ hfset C) (λx. arr-of (happ F (elem-of x))))
    using hfun-to-arr-def by simp
  thus ?thesis
    using assms hfun-to-arr-def Fun-mkArr by simp
qed

lemma arr-of-img-hfset-ide-to-hf:
assumes ide a
shows arr-of ‘ hfset (ide-to-hf a) = set a
proof −
  have arr-of ‘ hfset (ide-to-hf a) = arr-of ‘ hfset (HF (elem-of ‘ set a))
    using ide-to-hf-def by simp
  also have ... = arr-of ‘ elem-of ‘ set a
    using assms ide-implies-finite-set(1) ide-char by auto
```

```
  also have ... = set a
  proof −
    have ⋀x. x ∈ set a ⟹ arr-of (elem-of x) = x
      using assms ide-char arr-of-elem-of setp-set-ide by blast
    thus ?thesis by force
  qed
  finally show ?thesis by blast
qed

lemma hfun-to-arr-arr-to-hfun:
assumes arr f
shows hfun-to-arr (ide-to-hf (dom f)) (ide-to-hf (cod f)) (arr-to-hfun f) = f
proof −
  have 0: hfun-to-arr (ide-to-hf (dom f)) (ide-to-hf (cod f)) (arr-to-hfun f) =
          mkArr (arr-of ‘ hfset (ide-to-hf (dom f))) (arr-of ‘ hfset (ide-to-hf (cod f)))
                (λx. arr-of (happ (arr-to-hfun f) (elem-of x)))
    unfolding hfun-to-arr-def by blast
  also have ... = mkArr (Dom f) (Cod f)
                        (restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f))
  proof (intro mkArr-eqI)
    show 1: arr-of ‘ hfset (ide-to-hf (dom f)) = Dom f
      using assms arr-of-img-hfset-ide-to-hf ide-dom by simp
    show 2: arr-of ‘ hfset (ide-to-hf (cod f)) = Cod f
      using assms arr-of-img-hfset-ide-to-hf ide-cod by simp
    show arr (mkArr (arr-of ‘ hfset (ide-to-hf (dom f))) (arr-of ‘ hfset (ide-to-hf (cod f)))
                    (λx. arr-of (happ (arr-to-hfun f) (elem-of x))))
      using 0 1 2
      by (metis (no-types, lifting) arrI assms hfun-arr-to-hfun hfun-to-arr-in-hom)
    show ⋀x. x ∈ arr-of ‘ hfset (ide-to-hf (dom f)) ⟹
                arr-of (happ (arr-to-hfun f) (elem-of x)) =
                restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f) x
      using assms 1 by simp
  qed
  also have ... = mkArr (Dom f) (Cod f) (Fun f)
    using assms Fun-char mkArr-eqI by simp
  also have ... = f
    using assms mkArr-Fun by blast
  finally show ?thesis by simp
qed

lemma arr-to-hfun-hfun-to-arr:
assumes hfun B C F
shows arr-to-hfun (hfun-to-arr B C F) = F
proof −
  have arr-to-hfun (hfun-to-arr B C F) =
       ⦃XY ∈ ide-to-hf (dom (hfun-to-arr B C F)) ∗ ide-to-hf (cod (hfun-to-arr B C F)).
         hsnd XY = elem-of (Fun (hfun-to-arr B C F)) (arr-of (hfst XY))⦄
    unfolding arr-to-hfun-def by blast
  also have
```

```
    ... = {‖XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C)).
             hsnd XY = elem-of (Fun (hfun-to-arr B C F) (arr-of (hfst XY)))‖}
   using assms hfun-to-arr-in-hom [of B C F] hf-to-ide-def
   by (metis (no-types, lifting) in-homE)
 also have
    ... = {‖XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C)).
           hsnd XY = elem-of (restrict (λx. arr-of (happ F (elem-of x))) (arr-of ‘ hfset B)
                                       (arr-of (hfst XY)))‖}
   using assms Fun-hfun-to-arr by simp
 also have
    ... = {‖XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C)).
             hsnd XY = elem-of (arr-of (happ F (elem-of (arr-of (hfst XY)))))‖}
 proof −
   have
     1: ⋀XY. XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset
C))
                  ⟹ arr-of (hfst XY) ∈ arr-of ‘ hfset B
   proof −
     fix XY
     assume
       XY: XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C))
     have hfst XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B))
       using XY by auto
     thus arr-of (hfst XY) ∈ arr-of ‘ hfset B
       using assms arr-of-membI [of hfst XY mkIde (arr-of ‘ hfset B)] set-mkIde
       by (metis (mono-tags, lifting) arrI arr-mkArr hfun-to-arr-def hfun-to-arr-in-hom)
   qed
   show ?thesis
   proof −
     have
       ⋀XY. (XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset
C)) ∧
             hsnd XY = elem-of (restrict (λx. arr-of (happ F (elem-of x))) (arr-of ‘ hfset B)
                                         (arr-of (hfst XY))))
             ⟷
             (XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C))
∧
               hsnd XY = elem-of (arr-of (happ F (elem-of (arr-of (hfst XY))))))
       using 1 by auto
     thus ?thesis by blast
   qed
 qed
 also have
   ... = {‖XY ∈ ide-to-hf (mkIde (arr-of ‘ hfset B)) ∗ ide-to-hf (mkIde (arr-of ‘ hfset C)).
            hsnd XY = happ F (hfst XY)‖}
   by simp
 also have ... = {‖XY ∈ B ∗ C. hsnd XY = happ F (hfst XY)‖}
   using assms hf-to-ide-def ide-to-hf-hf-to-ide by force
 also have ... = F
```

using *assms happ-expansion* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *bij-betw-hom-hfun*:
assumes *ide a* **and** *ide b*
shows *bij-betw arr-to-hfun* (*hom a b*) {*F*. *hfun* (*ide-to-hf a*) (*ide-to-hf b*) *F*}
proof (*intro bij-betwI*)
show *arr-to-hfun* ∈ *hom a b* → {*F*. *hfun* (*ide-to-hf a*) (*ide-to-hf b*) *F*}
using *assms arr-to-hfun-in-hexp hexp-def hfun-arr-to-hfun* **by** *blast*
show *hfun-to-arr* (*ide-to-hf a*) (*ide-to-hf b*)
∈ {*F*. *hfun* (*ide-to-hf a*) (*ide-to-hf b*) *F*} → *hom a b*
using *assms hfun-to-arr-in-hom*
by (*metis* (*no-types*, *lifting*) *Pi-I hf-to-ide-ide-to-hf mem-Collect-eq*)
show ⋀*x*. *x* ∈ *hom a b* ⟹ *hfun-to-arr* (*ide-to-hf a*) (*ide-to-hf b*) (*arr-to-hfun x*) = *x*
using *assms hfun-to-arr-arr-to-hfun* **by** *blast*
show ⋀*y*. *y* ∈ {*F*. *hfun* (*ide-to-hf a*) (*ide-to-hf b*) *F*} ⟹
arr-to-hfun (*hfun-to-arr* (*ide-to-hf a*) (*ide-to-hf b*) *y*) = *y*
using *assms arr-to-hfun-hfun-to-arr* **by** *simp*
qed

We next relate composition of arrows in the category to the corresponding operation on hereditarily finite sets.

definition *hcomp*
where *hcomp G F* =
{‖*XZ* ∈ *hdomain F* ∗ *hrange G*. *hsnd XZ* = *happ G* (*happ F* (*hfst XZ*))‖}

lemma *hfun-hcomp*:
assumes *hfun A B F* **and** *hfun B C G*
shows *hfun A C* (*hcomp G F*)
proof
show *hcomp G F* ≤ *A* ∗ *C*
using *assms hcomp-def hfun-def* **by** *auto*
show ⋀*X*. *X* ∈ *A* ⟹ ∃!*Y*. ⟨*X*, *Y*⟩ ∈ *hcomp G F*
proof
fix *X*
assume *X*: *X* ∈ *A*
show ⟨*X*, *happ G* (*happ F X*)⟩ ∈ *hcomp G F*
unfolding *hcomp-def*
using *assms X hfunE happ-mapsto hfun-in-hexp*
by (*metis* (*mono-tags*, *lifting*) *HCollect-iff hfst-conv hfun-def hsnd-conv timesI*)
show ⋀*X Y*. ⟦*X* ∈ *A*; ⟨*X*, *Y*⟩ ∈ *hcomp G F*⟧ ⟹ *Y* = *happ G* (*happ F X*)
unfolding *hcomp-def* **by** *simp*
qed
show ⋀*X Y*. ⟨*X*, *Y*⟩ ∈ *hcomp G F* ⟹ *Y* ∈ *C*
unfolding *hcomp-def*
using *assms hfunE happ-mapsto hfun-in-hexp*
by (*metis HCollectE hfun-def hsubsetCE timesD2*)
qed

lemma *arr-to-hfun-comp*:
assumes *seq g f*
shows *arr-to-hfun (comp g f) = hcomp (arr-to-hfun g) (arr-to-hfun f)*
proof −
  have *1*: *hdomain (arr-to-hfun f) = ide-to-hf (dom f)*
    using *assms hfun-arr-to-hfun hfun-def* **by** *blast*
  have *arr-to-hfun (comp g f) =*
      {|XZ ∈ ide-to-hf (dom f) ∗ ide-to-hf (cod g).
        hsnd XZ = elem-of (Fun (comp g f) (arr-of (hfst XZ)))|}
    unfolding *arr-to-hfun-def comp-def*
    using *assms* **by** *fastforce*
  also have *... = {|XZ ∈ hdomain (arr-to-hfun f) ∗ hrange (arr-to-hfun g).*
                  hsnd XZ = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))|}
  proof
    fix *XZ*
    have *hfst XZ ∈ hdomain (arr-to-hfun f)*
          ⟹ hsnd XZ ∈ ide-to-hf (cod g) ∧
               hsnd XZ = elem-of (Fun (comp g f) (arr-of (hfst XZ)))
             ⟷
            hsnd XZ ∈ hrange (arr-to-hfun g) ∧
              hsnd XZ = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))
    proof
      assume *XZ*: *hfst XZ ∈ hdomain (arr-to-hfun f)*
      have *2*: *arr-of (hfst XZ) ∈ Dom f*
        using *XZ 1 hfsetcat.arr-of-membI* **by** *auto*
      have *3*: *arr-of (happ (arr-to-hfun f) (hfst XZ)) ∈ Dom g*
        using *assms XZ 2*
        by (*metis* (*no-types*, *lifting*) *1 happ-mapsto*(*1*) *hfsetcat.arr-of-membI*
            arr-to-hfun-in-hexp seqE)
      have *4*: *elem-of (Fun (comp g f) (arr-of (hfst XZ))) =*
             happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))
      proof −
        have *elem-of (Fun (comp g f) (arr-of (hfst XZ))) =*
              elem-of (restrict (Fun g o Fun f) (Dom f) (arr-of (hfst XZ)))
          using *assms Fun-comp Fun-char* **by** *simp*
        also have *... = elem-of ((Fun g o Fun f) (arr-of (hfst XZ)))*
          using *XZ 2* **by** *auto*
        also have *... = elem-of (Fun g (Fun f (arr-of (hfst XZ))))*
          by *simp*
        also have
         ... = elem-of (Fun g (restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f)
                                  (arr-of (hfst XZ))))
        proof −
          have *Fun f = restrict (λx. arr-of (happ (arr-to-hfun f) (elem-of x))) (Dom f)*
            using *assms Fun-char* [*of f*] **by** *blast*
          thus *?thesis* **by** *simp*
        qed
        also have *... = elem-of (Fun g (arr-of (happ (arr-to-hfun f) (hfst XZ))))*

```
      using 2 by simp
    also have ... = elem-of (restrict (λx. arr-of (happ (arr-to-hfun g) (elem-of x))) (Dom
g)
                                      (arr-of (happ (arr-to-hfun f) (hfst XZ))))
      proof −
        have Fun g = restrict (λx. arr-of (happ (arr-to-hfun g) (elem-of x))) (Dom g)
          using assms Fun-char [of g] by blast
        thus ?thesis by simp
      qed
      also have ... = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))
        using 3 by simp
      finally show ?thesis by blast
    qed
    have 5: elem-of (Fun (comp g f) (arr-of (hfst XZ))) ∈ hrange (arr-to-hfun g)
    proof −
      have happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ)) ∈ hrange (arr-to-hfun g)
       using assms 1 3 XZ hfun-arr-to-hfun happ-mapsto arr-to-hfun-in-hexp arr-to-hfun-def
        by (metis (no-types, lifting) seqE)
      thus ?thesis
        using XZ 4 by simp
    qed
    show hsnd XZ ∈ ide-to-hf (cod g) ∧
           hsnd XZ = elem-of (Fun (comp g f) (arr-of (hfst XZ)))
                ⟹
         hsnd XZ ∈ hrange (arr-to-hfun g) ∧
           hsnd XZ = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))
      using XZ 4 5 by simp
    show hsnd XZ ∈ hrange (arr-to-hfun g) ∧
           hsnd XZ = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))
                ⟹
         hsnd XZ ∈ ide-to-hf (cod g) ∧
           hsnd XZ = elem-of (Fun (comp g f) (arr-of (hfst XZ)))
      using assms XZ 1 4
      by (metis (no-types, lifting) arr-to-hfun-in-hexp happ-mapsto(1) seqE)
  qed
  thus XZ ∈ {|XZ ∈ ide-to-hf (dom f) ∗ ide-to-hf (cod g).
                hsnd XZ = elem-of (Fun (comp g f) (arr-of (hfst XZ)))|}
       ⟷
       XZ ∈ {|XZ ∈ hdomain (arr-to-hfun f) ∗ hrange (arr-to-hfun g).
              hsnd XZ = happ (arr-to-hfun g) (happ (arr-to-hfun f) (hfst XZ))|}
    using 1 is-hpair-def by auto
qed
also have ... = hcomp (arr-to-hfun g) (arr-to-hfun f)
  using assms arr-to-hfun-def hcomp-def by simp
finally show ?thesis by simp
qed

lemma hfun-to-arr-hcomp:
assumes hfun A B F and hfun B C G
```

```
shows hfun-to-arr A C (hcomp G F) = comp (hfun-to-arr B C G) (hfun-to-arr A B F)
proof −
  have 1: arr-to-hfun (hfun-to-arr A C (hcomp G F)) =
          arr-to-hfun (comp (hfun-to-arr B C G) (hfun-to-arr A B F))
  proof −
    have arr-to-hfun (comp (hfun-to-arr B C G) (hfun-to-arr A B F)) =
         hcomp (arr-to-hfun (hfun-to-arr B C G)) (arr-to-hfun (hfun-to-arr A B F))
      using assms arr-to-hfun-comp hfun-to-arr-in-hom by blast
    also have ... = hcomp G F
      using assms by (simp add: arr-to-hfun-hfun-to-arr)
    also have ... = arr-to-hfun (hfun-to-arr A C (hcomp G F))
    proof −
      have hfun A C (hcomp G F)
        using assms hfun-hcomp by simp
      thus ?thesis
        by (simp add: arr-to-hfun-hfun-to-arr)
    qed
    finally show ?thesis by simp
  qed
  show ?thesis
  proof −
    have hfun-to-arr A C (hcomp G F) ∈ hom (hf-to-ide A) (hf-to-ide C)
      using assms hfun-hcomp hf-to-ide-def hfun-to-arr-in-hom by auto
    moreover have comp (hfun-to-arr B C G) (hfun-to-arr A B F)
                     ∈ hom (hf-to-ide A) (hf-to-ide C)
      using assms hfun-to-arr-in-hom hf-to-ide-def
      by (metis (no-types, lifting) comp-in-homI mem-Collect-eq)
    moreover have inj-on arr-to-hfun (hom (hf-to-ide A) (hf-to-ide C))
    proof −
      have ide (hf-to-ide A) ∧ ide (hf-to-ide C)
        using assms hf-to-ide-mapsto by auto
      thus ?thesis
        using bij-betw-hom-hfun [of hf-to-ide A hf-to-ide C] bij-betw-imp-inj-on
        by auto
    qed
    ultimately show ?thesis
      using 1 inj-on-def [of arr-to-hfun hom (hf-to-ide A) (hf-to-ide C)] by simp
  qed
qed
```

25.3 Binary Products

The category of hereditarily finite sets has binary products, given by cartesian product of sets in the usual way.

```
definition prod
where prod a b = hf-to-ide (ide-to-hf a ∗ ide-to-hf b)

definition pr0
```

```
where pr0 a b = (if ide a ∧ ide b then
                   mkArr (set (prod a b)) (set b) (λx. arr-of (hsnd (elem-of x)))
                 else null)

definition pr1
where pr1 a b = (if ide a ∧ ide b then
                   mkArr (set (prod a b)) (set a) (λx. arr-of (hfst (elem-of x)))
                 else null)

definition tuple
where tuple f g = mkArr (set (dom f)) (set (prod (cod f) (cod g)))
                        (λx. arr-of (hpair (elem-of (Fun f x)) (elem-of (Fun g x))))

lemma ide-prod:
assumes ide a and ide b
shows ide (prod a b)
  using assms prod-def hf-to-ide-mapsto ide-to-hf-mapsto by auto

lemma pr1-in-hom [intro]:
assumes ide a and ide b
shows «pr1 a b : prod a b → a»
proof
  show 0: arr (pr1 a b)
  proof −
    have set (prod a b) ⊆ Univ ∧ finite (set (prod a b))
      using assms ide-implies-finite-set(1) set-ideD(1) ide-prod by presburger
    moreover have set a ⊆ Univ ∧ finite (set a)
      using assms ide-char set-ide-char by blast
    moreover have (λx. arr-of (hfst (elem-of x))) ∈ set (prod a b) → set a
    proof (unfold prod-def)
      show (λx. arr-of (hfst (elem-of x))) ∈ set (hf-to-ide (ide-to-hf a ∗ ide-to-hf b)) → set a
      proof
        fix x
          assume x: x ∈ set (hf-to-ide (ide-to-hf a ∗ ide-to-hf b))
          have elem-of x ∈ hfset (ide-to-hf a ∗ ide-to-hf b)
            using assms ide-char x
           by (metis (no-types, lifting) prod-def elem-of-membI HF-hfset UNIV-I hmem-HF-iff
                ide-prod ide-to-hf-hf-to-ide)
          hence hfst (elem-of x) ∈ ide-to-hf a
            by (metis HF-hfset finite-hfset hfst-conv hmem-HF-iff timesE)
          thus arr-of (hfst (elem-of x)) ∈ set a
            using arr-of-membI by simp
      qed
    qed
    ultimately show ?thesis
      unfolding pr1-def
      using assms arr-mkArr finite-imp-setp by presburger
  qed
  show dom (pr1 a b) = prod a b
```

```
    using assms 0 ide-char ide-prod dom-mkArr
    by (metis (no-types, lifting) mkIde-set pr1-def)
  show cod (pr1 a b) = a
    using assms 0 ide-char ide-prod cod-mkArr
    by (metis (no-types, lifting) mkIde-set pr1-def)
qed

lemma pr1-simps [simp]:
assumes ide a and ide b
shows arr (pr1 a b) and dom (pr1 a b) = prod a b and cod (pr1 a b) = a
  using assms pr1-in-hom by blast+

lemma pr0-in-hom [intro]:
assumes ide a and ide b
shows «pr0 a b : prod a b → b»
proof
  show 0: arr (pr0 a b)
  proof −
    have set (prod a b) ⊆ Univ ∧ finite (set (prod a b))
      using setp-set-ide assms ide-implies-finite-set(1) ide-prod by presburger
    moreover have set b ⊆ Univ ∧ finite (set b)
      using assms ide-char set-ide-char by blast
    moreover have (λx. arr-of (hsnd (elem-of x))) ∈ set (prod a b) → set b
    proof (unfold prod-def)
     show (λx. arr-of (hsnd (elem-of x))) ∈ set (hf-to-ide (ide-to-hf a ∗ ide-to-hf b)) → set b
      proof
        fix x
          assume x: x ∈ set (hf-to-ide (ide-to-hf a ∗ ide-to-hf b))
          have elem-of x ∈ hfset (ide-to-hf a ∗ ide-to-hf b)
            using assms ide-char x
           by (metis (no-types, lifting) prod-def elem-of-membI HF-hfset UNIV-I hmem-HF-iff
               ide-prod ide-to-hf-hf-to-ide)
          hence hsnd (elem-of x) ∈ ide-to-hf b
            by (metis HF-hfset finite-hfset hsnd-conv hmem-HF-iff timesE)
          thus arr-of (hsnd (elem-of x)) ∈ set b
            using arr-of-membI by simp
      qed
    qed
    ultimately show ?thesis
      unfolding pr0-def
      using assms arr-mkArr finite-imp-setp by presburger
  qed
  show dom (pr0 a b) = prod a b
    using assms 0 ide-char ide-prod dom-mkArr
    by (metis (no-types, lifting) mkIde-set pr0-def)
  show cod (pr0 a b) = b
    using assms 0 ide-char ide-prod cod-mkArr
    by (metis (no-types, lifting) mkIde-set pr0-def)
qed
```

lemma *pr0-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* (*pr0 a b*) **and** *dom* (*pr0 a b*) = *prod a b* **and** *cod* (*pr0 a b*) = *b*
  using *assms pr0-in-hom* **by** *blast+*

lemma *arr-of-tuple-elem-of-membI*:
assumes *span f g* **and** *x* ∈ *Dom f*
shows *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩ ∈ *set* (*prod* (*cod f*) (*cod g*))
proof −
  have *Fun f x* ∈ *set* (*cod f*)
    using *assms Fun-mapsto* **by** *blast*
  moreover have *Fun g x* ∈ *set* (*cod g*)
    using *assms Fun-mapsto* **by** *auto*
  ultimately have ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩
                    ∈ *ide-to-hf* (*cod f*) ∗ *ide-to-hf* (*cod g*)
    using *assms ide-cod* **by** *auto*
  moreover have *set* (*prod* (*cod f*) (*cod g*)) ⊆ *Univ*
    using *setp-set-ide assms*(*1*) *ide-cod ide-prod* **by** *presburger*
  ultimately show *?thesis*
    using *prod-def arr-of-membI ide-to-hf-hf-to-ide* **by** *auto*
qed

lemma *tuple-in-hom* [*intro*]:
assumes *span f g*
shows «*tuple f g* : *dom f* → *prod* (*cod f*) (*cod g*)»
proof
  show *1*: *arr* (*tuple f g*)
  proof −
    have *Dom f* ⊆ *Univ* ∧ *finite* (*Dom f*)
      using *assms set-ideD*(*1*) *ide-dom ide-implies-finite-set*(*1*) **by** *presburger*
    moreover have *set* (*prod* (*cod f*) (*cod g*)) ⊆ *Univ* ∧ *finite* (*set* (*prod* (*cod f*) (*cod g*)))
      using *assms set-ideD*(*1*) *ide-cod ide-prod ide-implies-finite-set*(*1*) **by** *presburger*
    moreover have (λ*x*. *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩)
                    ∈ *Dom f* → *set* (*prod* (*cod f*) (*cod g*))
      using *assms arr-of-tuple-elem-of-membI* **by** *simp*
    ultimately show *?thesis*
      using *assms ide-prod tuple-def arr-mkArr ide-dom ide-cod* **by** *simp*
  qed
  show *dom* (*tuple f g*) = *dom f*
    using *assms 1 dom-mkArr ide-dom mkIde-set tuple-def* **by** *auto*
  show *cod* (*tuple f g*) = *prod* (*cod f*) (*cod g*)
    using *assms 1 cod-mkArr ide-cod mkIde-set tuple-def ide-prod* **by** *auto*
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* (*cod f*) (*cod g*)

using *assms tuple-in-hom* **by** *blast+*

lemma *Fun-pr1*:
assumes *ide a* **and** *ide b*
shows *Fun* (*pr1 a b*) = *restrict* (λ*x*. *arr-of* (*hfst* (*elem-of x*))) (*set* (*prod a b*))
using *assms pr1-def Fun-mkArr arr-char pr1-simps*(*1*) **by** *presburger*

lemma *Fun-pr0*:
assumes *ide a* **and** *ide b*
shows *Fun* (*pr0 a b*) = *restrict* (λ*x*. *arr-of* (*hsnd* (*elem-of x*))) (*set* (*prod a b*))
using *assms pr0-def Fun-mkArr arr-char pr0-simps*(*1*) **by** *presburger*

lemma *Fun-tuple*:
assumes *span f g*
shows *Fun* (*tuple f g*) = *restrict* (λ*x*. *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩) (*Dom f*)
proof −
have *arr* (*tuple f g*)
using *assms tuple-in-hom* **by** *blast*
thus *?thesis*
using *assms tuple-def Fun-mkArr* **by** *simp*
qed

lemma *pr1-tuple*:
assumes *span f g*
shows *comp* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*) = *f*
proof (*intro arr-eqI*$_{SC}$)
have *pr1*: «*pr1* (*cod f*) (*cod g*) : *prod* (*cod f*) (*cod g*) → *cod f*»
using *assms ide-cod* **by** *blast*
have *tuple*: «*tuple f g* : *dom f* → *prod* (*cod f*) (*cod g*)»
using *assms* **by** *blast*
show *par*: *par* (*comp* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*)) *f*
using *assms pr1-in-hom tuple-in-hom*
by (*metis* (*no-types*, *lifting*) *comp-in-homI*′ *ide-cod in-homE*)
show *Fun* (*comp* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*)) = *Fun f*
proof −
have *seq*: *seq* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*)
using *par* **by** *blast*
have *Fun* (*comp* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*)) =
restrict (*Fun* (*pr1* (*cod f*) (*cod g*)) ∘ *Fun* (*tuple f g*)) (*Dom* (*tuple f g*))
using *pr1 tuple seq Fun-comp* **by** *simp*
also have ... = *restrict*
(*Fun* (*mkArr* (*set* (*prod* (*cod f*) (*cod g*))) (*Cod f*)
(λ*x*. *arr-of* (*hfst* (*elem-of x*)))) ∘
Fun (*mkArr* (*Dom f*) (*set* (*prod* (*cod f*) (*cod g*)))
(λ*x*. *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩)))
(*Dom* (*tuple f g*))
unfolding *pr1-def tuple-def*
using *assms ide-cod* **by** *presburger*

```
    also have
      ... = restrict
              (restrict (λx. arr-of (hfst (elem-of x))) (set (prod (cod f) (cod g))) o
                 restrict (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩) (Dom f))
              (Dom f)
    proof −
     have Fun (mkArr (set (prod (cod f) (cod g))) (Cod f) (λx. arr-of (hfst (elem-of x)))) =
           restrict (λx. arr-of (hfst (elem-of x))) (set (prod (cod f) (cod g)))
       using assms Fun-mkArr ide-prod pr1
       by (metis (no-types, lifting) arrI ide-cod pr1-def)
     moreover have Fun (mkArr (Dom f) (set (prod (cod f) (cod g)))
                                (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩)) =
                   restrict (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩) (Dom f)
       using assms Fun-mkArr ide-prod ide-cod tuple-def tuple arrI by simp
     ultimately show ?thesis
       using assms tuple-simps(2) by simp
    qed
    also have
      ... = restrict
              ((λx. arr-of (hfst (elem-of x))) o (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g
x)⟩))
              (Dom f)
      using assms tuple tuple-def arr-of-tuple-elem-of-membI by auto
    also have ... = restrict (Fun f) (Dom f)
    proof
      fix x
      have restrict ((λx. arr-of (hfst (elem-of x))) o (λx. arr-of ⟨elem-of (Fun f x), elem-of
(Fun g x)⟩))
                    (Dom f) x =
           restrict (λx. arr-of (elem-of (Fun f x))) (Dom f) x
        by simp
      also have ... = restrict (Fun f) (Dom f) x
      proof (cases x ∈ Dom f)
        show x ∉ Dom f ⟹ ?thesis by simp
        assume x: x ∈ Dom f
        have Fun f x ∈ Cod f
          using assms x Fun-mapsto arr-char by blast
        moreover have Cod f ⊆ Univ
          using setp-set-ide assms ide-cod by blast
        ultimately show ?thesis
          using assms arr-of-elem-of Fun-mapsto by auto
      qed
      finally show restrict ((λx. arr-of (hfst (elem-of x))) ∘
                             (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩))
                            (Dom f) x =
                   restrict (Fun f) (Dom f) x
        by blast
    qed
    also have ... = Fun f
```

```
      using assms par Fun-mapsto Fun-mkArr mkArr-Fun
      by (metis (no-types, lifting))
    finally show ?thesis by blast
  qed
qed

lemma pr0-tuple:
assumes span f g
shows comp (pr0 (cod f) (cod g)) (tuple f g) = g
proof (intro arr-eqI_SC)
  have pr0: «pr0 (cod f) (cod g) : prod (cod f) (cod g) → cod g»
    using assms ide-cod by blast
  have tuple: «tuple f g : dom f → prod (cod f) (cod g)»
    using assms by blast
  show par: par (comp (pr0 (cod f) (cod g)) (tuple f g)) g
    using assms pr0-in-hom tuple-in-hom
    by (metis (no-types, lifting) comp-in-homI' ide-cod in-homE)
  show Fun (comp (pr0 (cod f) (cod g)) (tuple f g)) = Fun g
  proof -
    have seq: seq (pr0 (cod f) (cod g)) (tuple f g)
      using par by blast
    have Fun (comp (pr0 (cod f) (cod g)) (tuple f g)) =
        restrict (Fun (pr0 (cod f) (cod g)) ∘ Fun (tuple f g)) (Dom (tuple f g))
      using pr0 tuple seq Fun-comp by simp
    also have
      ... = restrict
            (Fun (mkArr (set (prod (cod f) (cod g))) (Cod g)
                        (λx. arr-of (hsnd (elem-of x)))) ∘
             Fun (mkArr (Dom f) (set (prod (cod f) (cod g)))
                        (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩)))
            (Dom (tuple f g))
      unfolding pr0-def tuple-def
      using assms ide-cod by presburger
    also have ... = restrict
                      (restrict (λx. arr-of (hsnd (elem-of x))) (set (prod (cod f) (cod g))) o
                       restrict (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩) (Dom g))
                      (Dom g)
    proof -
      have Fun (mkArr (set (prod (cod f) (cod g))) (Cod g) (λx. arr-of (hsnd (elem-of x))))
=
           restrict (λx. arr-of (hsnd (elem-of x))) (set (prod (cod f) (cod g)))
        using assms Fun-mkArr ide-prod arrI
        by (metis (no-types, lifting) ide-cod pr0 pr0-def)
      moreover have Fun (mkArr (Dom f) (set (prod (cod f) (cod g)))
                               (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩)) =
                    restrict (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩) (Dom f)
        using assms Fun-mkArr ide-prod ide-cod tuple-def tuple arrI by simp
      ultimately show ?thesis
        using assms tuple-simps(2) by simp
```

```
      qed
      also have ... = restrict
                        ((λx. arr-of (hsnd (elem-of x))) o (λx. arr-of ⟨elem-of (Fun f x), elem-of
(Fun g x)⟩))
                        (Dom g)
        using assms tuple tuple-def arr-of-tuple-elem-of-membI by auto
      also have ... = restrict (Fun g) (Dom g)
      proof
        fix x
        have restrict ((λx. arr-of (hsnd (elem-of x)))
                        o (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩))
                      (Dom g) x =
             restrict (λx. arr-of (elem-of (Fun g x))) (Dom g) x
          by simp
        also have ... = restrict (Fun g) (Dom g) x
        proof (cases x ∈ Dom g)
          show x ∉ Dom g ⟹ ?thesis by simp
          assume x: x ∈ Dom g
          have Fun g x ∈ Cod g
            using assms x Fun-mapsto arr-char by blast
          moreover have Cod g ⊆ Univ
            using assms set-ideD(1) ide-cod by blast
          ultimately show ?thesis
            using assms arr-of-elem-of Fun-mapsto by auto
        qed
        finally show restrict ((λx. arr-of (hsnd (elem-of x))) ∘
                                 (λx. arr-of ⟨elem-of (Fun f x), elem-of (Fun g x)⟩))
                              (Dom g) x =
                     restrict (Fun g) (Dom g) x
          by blast
      qed
      also have ... = Fun g
        using assms par Fun-mapsto Fun-mkArr mkArr-Fun
        by (metis (no-types, lifting))
      finally show ?thesis by blast
    qed
  qed

  lemma tuple-pr:
  assumes ide a and ide b and «h : dom h → prod a b»
  shows tuple (comp (pr1 a b) h) (comp (pr0 a b) h) = h
  proof (intro arr-eqI_SC)
    have pr0: «pr0 a b : prod a b → b»
      using assms pr0-in-hom ide-cod by blast
    have pr1: «pr1 a b : prod a b → a»
      using assms pr1-in-hom ide-cod by blast
    have tuple: «tuple (comp (pr1 a b) h) (comp (pr0 a b) h) : dom h → prod a b»
      using assms pr0 pr1
      by (metis (no-types, lifting) cod-comp dom-comp pr0-simps(3) pr1-simps(3)
```

```
    seqI' tuple-in-hom)
show par: par (tuple (comp (pr1 a b) h) (comp (pr0 a b) h)) h
  using assms tuple by (metis (no-types, lifting) in-homE)
show Fun (tuple (comp (pr1 a b) h) (comp (pr0 a b) h)) = Fun h
proof −
  have 1: Fun (comp (pr1 a b) h) =
        restrict (restrict (λx. arr-of (hfst (elem-of x))) (set (prod a b)) ∘ Fun h) (Dom h)
    using assms pr1 Fun-comp Fun-pr1 seqI' by auto
  have 2: Fun (comp (pr0 a b) h) =
        restrict (restrict (λx. arr-of (hsnd (elem-of x))) (set (prod a b)) ∘ Fun h) (Dom h)
    using assms pr0 Fun-comp Fun-pr0 seqI' by auto
  have Fun (tuple (comp (pr1 a b) h) (comp (pr0 a b) h)) =
      restrict (λx. arr-of ⟨elem-of (restrict
                          (restrict (λx. arr-of (hfst (elem-of x))) (set (prod a b)) ∘ Fun h)
                              (Dom h) x),
                    elem-of (restrict
                        (restrict (λx. arr-of (hsnd (elem-of x))) (set (prod a b)) ∘ Fun h)
                              (Dom h) x)⟩)
            (Dom h)
  proof −
    have Dom (comp (pr1 a b) h) = Dom h
      using assms pr1-in-hom
      by (metis (no-types, lifting) in-homE dom-comp seqI)
    moreover have arr (mkArr (Dom (comp (pr1 a b) h))
                     (set (prod (cod (comp (pr1 a b) h)) (cod (comp (pr0 a b) h))))
                     (λx. arr-of ⟨elem-of (Fun (comp (pr1 a b) h) x),
                           elem-of (Fun (comp (pr0 a b) h) x)⟩))
      using tuple unfolding tuple-def by blast
    ultimately show ?thesis
      using 1 2 tuple tuple-def
          Fun-mkArr [of Dom (comp (pr1 a b) h)
                     set (prod (cod (comp (pr1 a b) h))
                               (cod (comp (pr0 a b) h)))
                     λx. arr-of ⟨elem-of (Fun (comp (pr1 a b) h) x),
                           elem-of (Fun (comp (pr0 a b) h) x)⟩]
      by simp
  qed
  also have ... = Fun h
  proof
    let ?f = ...
    fix x
    show ?f x = Fun h x
    proof −
      have x ∉ Dom h ⟹ ?f x = Fun h x
      proof −
        assume x: x ∉ Dom h
        have restrict ?f (Dom h) x = undefined
          using assms x restrict-apply by auto
        also have ... = Fun h x
```

```
        proof −
          have arr h
            using assms by blast
          thus ?thesis
            using assms x Fun-mapsto [of h] extensional-arb [of Fun h Dom h x]
            by simp
        qed
        finally show ?thesis by auto
      qed
      moreover have x ∈ Dom h ⟹ ?f x = Fun h x
      proof −
        assume x: x ∈ Dom h
        have 1: Fun h x ∈ set (prod a b)
        proof −
          have Fun h x ∈ Cod h
            using assms x Fun-mapsto [of h] by blast
          moreover have Cod h = set (prod a b)
            using assms ide-prod
            by (metis (no-types, lifting) in-homE)
          ultimately show ?thesis by fast
        qed
        have ?f x = arr-of ⟨hfst (elem-of (Fun h x)), hsnd (elem-of (Fun h x))⟩
          using x 1 by simp
        also have ... = arr-of (elem-of (Fun h x))
        proof −
          have elem-of (Fun h x) ∈ ide-to-hf a ∗ ide-to-hf b
            using assms x 1 par
            by (metis (no-types, lifting) prod-def elem-of-membI UNIV-I ide-prod
                ide-to-hf-hf-to-ide)
          thus ?thesis
            using x is-hpair-def by auto
        qed
        also have ... = Fun h x
          using 1 arr-of-elem-of assms set-ideD(1) ide-prod by blast
        finally show ?thesis by blast
      qed
      ultimately show ?thesis by blast
    qed
  qed
  finally show ?thesis by blast
 qed
qed

interpretation HF′: elementary-category-with-binary-products comp pr0 pr1
proof
 show ⋀a b. ⟦ide a; ide b⟧ ⟹ span (pr1 a b) (pr0 a b)
   using pr0-simps(1) pr0-simps(2) pr1-simps(1) pr1-simps(2) by auto
 show ⋀a b. ⟦ide a; ide b⟧ ⟹ cod (pr0 a b) = b
   using pr0-simps(1−3) by blast
```

```
  show ⋀a b. ⟦ide a; ide b⟧ ⟹ cod (pr1 a b) = a
    using pr1-simps(1−3) by blast
  show ⋀f g. span f g ⟹
             ∃!l. comp (pr1 (cod f) (cod g)) l = f ∧ comp (pr0 (cod f) (cod g)) l = g
  proof
    fix f g
    assume fg: span f g
    show comp (pr1 (cod f) (cod g)) (tuple f g) = f ∧
         comp (pr0 (cod f) (cod g)) (tuple f g) = g
      using fg pr0-simps pr1-simps tuple-simps pr0-tuple pr1-tuple by presburger
    show ⋀l. ⟦comp (pr1 (cod f) (cod g)) l = f ∧ comp (pr0 (cod f) (cod g)) l = g⟧
               ⟹ l = tuple f g
    proof −
      fix l
      assume l: comp (pr1 (cod f) (cod g)) l = f ∧ comp (pr0 (cod f) (cod g)) l = g
      show l = tuple f g
        using fg l tuple-pr
        by (metis (no-types, lifting) arr-iff-in-hom ide-cod seqE pr0-simps(2))
    qed
  qed
qed
```

For reasons of economy of locale parameters, the notion *prod* is a defined notion of the *elementary-category-with-binary-products* locale. However, we need to be able to relate this notion to that of cartesian product of hereditarily finite sets, which we have already used to give a definition of *prod*. The locale assumptions for *elementary-cartesian-closed-category* refer specifically to *HF′.prod*, even though in the end the notion itself does not depend on that choice. To be able to show that the locale assumptions of *elementary-cartesian-closed-category* are satisfied, we need to use a choice of products that we can relate to the cartesian product of hereditarily finite sets. We therefore need to show that our previously defined *prod* coincides (on objects) with the one defined in the *elementary-category-with-binary-products* locale; *i.e. HF′.prod*. Note that the latter is defined for all arrows, not just identity arrows, so we need to use that for the subsequent definitions and proofs.

```
lemma prod-ide-eq:
assumes ide a and ide b
shows prod a b = HF′.prod a b
  using assms prod-def HF′.pr-simps(2) HF′.prod-def pr0-simps(2) by presburger

lemma tuple-span-eq:
assumes span f g
shows tuple f g = HF′.tuple f g
  using assms tuple-def HF′.tuple-def
  by (metis (no-types, lifting) HF′.tuple-eqI ide-cod pr0-tuple pr1-tuple)
```

25.4 Exponentials

We now turn our attention to exponentials.

definition *exp*
where *exp b c* = *hf-to-ide* (*hexp* (*ide-to-hf b*) (*ide-to-hf c*))

definition *eval*
where *eval b c* = *mkArr* (*set* (*HF′.prod* (*exp b c*) *b*)) (*set c*)
(λ*x*. *arr-of* (*happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*))))

definition Λ
where Λ *a b c f* = *mkArr* (*set a*) (*set* (*exp b c*))
(λ*x*. *arr-of* (*happ* (*hlam* (*ide-to-hf a*) (*ide-to-hf b*) (*ide-to-hf c*)
(*arr-to-hfun f*))
(*elem-of x*)))

lemma *ide-exp*:
assumes *ide b* **and** *ide c*
shows *ide* (*exp b c*)
using *assms exp-def hf-to-ide-mapsto ide-to-hf-mapsto* **by** *auto*

lemma *hfset-ide-to-hf*:
assumes *ide a*
shows *hfset* (*ide-to-hf a*) = *elem-of* ‘ *set a*
using *assms ide-to-hf-def ide-implies-finite-set*(*1*) **by** *auto*

lemma *eval-in-hom* [*intro*]:
assumes *ide b* **and** *ide c*
shows *in-hom* (*eval b c*) (*HF′.prod* (*exp b c*) *b*) *c*
proof
show *1*: *arr* (*eval b c*)
proof (*unfold eval-def arr-mkArr*, *intro conjI*)
show *set* (*HF′.prod* (*exp b c*) *b*) ⊆ *Univ*
using *HF′.ide-prod assms set-ideD*(*1*) *ide-exp* **by** *presburger*
show *set c* ⊆ *Univ*
using *assms set-ideD*(*1*) **by** *blast*
show (λ*x*. *arr-of* (*happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*))))
∈ *set* (*HF′.prod* (*exp b c*) *b*) → *set c*
proof
fix *x*
assume *x* ∈ *set* (*HF′.prod* (*exp b c*) *b*)
hence *x*: *x* ∈ *set* (*prod* (*exp b c*) *b*)
using *assms prod-ide-eq ide-exp* **by** *auto*
show *arr-of* (*happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*))) ∈ *set c*
proof (*intro arr-of-membI*)
show *happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*)) ∈ *ide-to-hf c*
proof −
have *1*: *elem-of x* ∈ *ide-to-hf* (*exp b c*) ∗ *ide-to-hf b*
proof −

```
        have elem-of x ∈ ide-to-hf (prod (exp b c) b)
          using assms x elem-of-membI ide-prod ide-exp by simp
        thus ?thesis
          using assms x prod-def ide-to-hf-hf-to-ide by auto
      qed
      have hfst (elem-of x) ∈ hexp (ide-to-hf b) (ide-to-hf c)
        using assms 1 x exp-def ide-to-hf-hf-to-ide by auto
      moreover have hsnd (elem-of x) ∈ ide-to-hf b
        using assms 1 by auto
      ultimately show ?thesis
        using happ-mapsto [of hfst (elem-of x) ide-to-hf b ide-to-hf c
                           hsnd (elem-of x)]
        by simp
    qed
  qed
  show finite (elem-of ‘ set (HF′.prod (exp b c) b))
    using HF′.ide-prod setp-set-ide assms ide-exp by presburger
  show finite (elem-of ‘ set c)
    using setp-set-ide assms(2) by blast
qed
show dom (eval b c) = HF′.prod (exp b c) b
  using assms 1 ide-char HF′.ide-prod ide-exp dom-mkArr eval-def
  by (metis (no-types, lifting) mkIde-set)
show cod (eval b c) = c
  using assms 1 ide-char cod-mkArr eval-def
  by (metis (no-types, lifting) mkIde-set)
qed

lemma eval-simps [simp]:
assumes ide b and ide c
shows arr (eval b c)
and dom (eval b c) = HF′.prod (exp b c) b
and cod (eval b c) = c
  using assms eval-in-hom by blast+

lemma hlam-arr-to-hfun-in-hexp:
assumes ide a and ide b and ide c
and in-hom f (prod a b) c
shows hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c) (arr-to-hfun f)
        ∈ hexp (ide-to-hf a) (ide-to-hf (exp b c))
  using assms hfun-in-hexp hfun-hlam
  by (metis (no-types, lifting) prod-def HCollect-iff in-homE UNIV-I
      arr-to-hfun-in-hexp exp-def hexp-def ide-to-hf-hf-to-ide)

lemma lam-in-hom [intro]:
assumes ide a and ide b and ide c
and in-hom f (prod a b) c
shows in-hom (Λ a b c f) a (exp b c)
```

```
proof
  show 1: arr (Λ a b c f)
  proof (unfold Λ-def arr-mkArr, intro conjI)
    show set a ⊆ Univ
      using assms(1) set-ideD(1) by blast
    show set (exp b c) ⊆ Univ
      using assms(2−3) set-ideD(1) ide-exp ide-char by blast
    show finite (elem-of ‘ set a)
      using assms(1) set-ideD(1) setp-set-ide by presburger
    show finite (elem-of ‘ set (exp b c))
      using assms(2−3) set-ideD(1) setp-set-ide ide-exp by presburger
    show (λx. arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c) (arr-to-hfun f))
                           (elem-of x)))
           ∈ set a → set (exp b c)
    proof
      fix x
      assume x: x ∈ set a
      show arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c) (arr-to-hfun f))
                        (elem-of x))
             ∈ set (exp b c)
        using assms x hlam-arr-to-hfun-in-hexp ide-to-hf-def elem-of-membI happ-mapsto
              arr-of-membI
        by meson
    qed
  qed
  show dom (Λ a b c f) = a
    using assms(1) 1 Λ-def ide-char dom-mkArr mkIde-set by auto
  show cod (Λ a b c f) = exp b c
    using assms(2−3) 1 Λ-def cod-mkArr ide-exp mkIde-set by auto
qed

lemma lam-simps [simp]:
assumes ide a and ide b and ide c
and in-hom f (prod a b) c
shows arr (Λ a b c f)
and dom (Λ a b c f) = a
and cod (Λ a b c f) = exp b c
  using assms lam-in-hom by blast+

lemma Fun-lam:
assumes ide a and ide b and ide c
and in-hom f (prod a b) c
shows Fun (Λ a b c f) =
      restrict (λx. arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c) (arr-to-hfun f))
                                 (elem-of x)))
               (set a)
  using assms arr-char lam-simps(1) Λ-def Fun-mkArr by simp

lemma Fun-eval:
```

assumes *ide b* **and** *ide c*
shows *Fun (eval b c) = restrict (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of x))))*
(set (HF'.prod (exp b c) b))
using *assms arr-char eval-simps(1) eval-def Fun-mkArr* **by** *force*

lemma *Fun-prod*:
assumes *arr f* **and** *arr g* **and** *x ∈ set (prod (dom f) (dom g))*
shows *Fun (HF'.prod f g) x = arr-of ⟨elem-of (Fun f (arr-of (hfst (elem-of x)))),*
elem-of (Fun g (arr-of (hsnd (elem-of x))))⟩
proof −
have *1*: *span (comp f (pr1 (dom f) (dom g))) (comp g (pr0 (dom f) (dom g)))*
using *assms*
by (*metis* (*no-types*, *lifting*) *HF'.prod-def HF'.prod-simps(1) HF'.tuple-ext not-arr-null*)
have *2*: *Dom (comp f (pr1 (dom f) (dom g))) = set (prod (dom f) (dom g))*
using *assms*
by (*metis* (*mono-tags*, *lifting*) *1 dom-comp ide-dom pr0-simps(2)*)
have *3*: *Dom (comp g (pr0 (dom f) (dom g))) = set (prod (dom f) (dom g))*
using *assms 1 2* **by** *force*
have *Fun (HF'.prod f g) x =*
Fun (HF'.tuple (comp f (pr1 (dom f) (dom g))) (comp g (pr0 (dom f) (dom g)))) x
using *assms(3) HF'.prod-def* **by** *simp*
also have *... = restrict (λx. arr-of ⟨elem-of (Fun (comp f (pr1 (dom f) (dom g))) x),*
elem-of (Fun (comp g (pr0 (dom f) (dom g))) x)⟩)
(Dom (comp f (pr1 (dom f) (dom g))))
x
using *assms 1 tuple-span-eq Fun-tuple* **by** *simp*
also have *... = arr-of ⟨elem-of (Fun (comp f (pr1 (dom f) (dom g))) x),*
elem-of (Fun (comp g (pr0 (dom f) (dom g))) x)⟩
using *assms(3) 2* **by** *simp*
also have *... = arr-of ⟨elem-of (Fun f (arr-of (hfst (elem-of x)))),*
elem-of (Fun g (arr-of (hsnd (elem-of x))))⟩
proof −
have *Fun (comp f (pr1 (dom f) (dom g))) x = Fun f (arr-of (hfst (elem-of x)))*
proof −

have *4*: *seq f (pr1 (dom f) (dom g))*
using *assms 1* **by** *blast*
have *Fun (comp f (pr1 (dom f) (dom g))) x =*
restrict (Fun f ∘ Fun (pr1 (dom f) (dom g))) (Dom (pr1 (dom f) (dom g))) x
using *assms 1 Fun-comp* [*of f pr1 (dom f) (dom g)*]
by (*metis* (*no-types*, *lifting*))
also have *... = (Fun f ∘ Fun (pr1 (dom f) (dom g))) x*
proof −
have *x ∈ Dom (pr1 (dom f) (dom g))*
using *assms 1 2 4*
by (*metis* (*no-types*, *lifting*) *dom-comp*)
thus *?thesis* **by** *simp*
qed
also have *... = Fun f (Fun (pr1 (dom f) (dom g)) x)*

```
      by simp
    also have ... = Fun f (arr-of (hfst (elem-of x)))
      using assms 1 Fun-pr1 [of dom f dom g] ide-dom by simp
    finally show ?thesis by blast
  qed
  moreover
  have Fun (comp g (pr0 (dom f) (dom g))) x = Fun g (arr-of (hsnd (elem-of x)))
  proof −
    have 4: seq g (pr0 (dom f) (dom g))
      using assms 1 by blast
    have Fun (comp g (pr0 (dom f) (dom g))) x =
          restrict (Fun g ∘ Fun (pr0 (dom f) (dom g))) (Dom (pr0 (dom f) (dom g))) x
      using assms 1 Fun-comp [of g pr0 (dom f) (dom g)]
      by (metis (no-types, lifting))
    also have ... = (Fun g ∘ Fun (pr0 (dom f) (dom g))) x
    proof −
      have x ∈ Dom (pr0 (dom f) (dom g))
        using assms 1 2 4
        by (metis (no-types, lifting) dom-comp)
      thus ?thesis by simp
    qed
    also have ... = Fun g (Fun (pr0 (dom f) (dom g)) x)
      by simp
    also have ... = Fun g (arr-of (hsnd (elem-of x)))
      using assms 1 Fun-pr0 [of dom f dom g] ide-dom by simp
    finally show ?thesis by blast
  qed
  ultimately show ?thesis by simp
 qed
 finally show ?thesis by simp
qed

lemma prod-in-terms-of-tuple:
assumes arr f and arr g
shows HF'.prod f g =
      tuple (comp f (pr1 (dom f) (dom g))) (comp g (pr0 (dom f) (dom g)))
 using assms HF'.prod-def tuple-span-eq
 by (metis (no-types, lifting) HF'.prod-simps(1) HF'.tuple-ext not-arr-null)

lemma eval-prod-lam:
assumes ide a and ide b and ide c
and in-hom g (prod a b) c
shows comp (eval b c) (HF'.prod (Λ a b c g) b) = g
proof −
 have ide-dom-lam: ide (dom (Λ a b c g))
   using assms lam-in-hom [of a b c g] ide-dom by blast
 have ide-dom-b: ide (dom b)
   using assms ide-dom ideD(1) by blast
 define Λ-pr1 where Λ-pr1 = comp (Λ a b c g) (pr1 (dom (Λ a b c g)) (dom b))
```

define *b-pr0* **where** *b-pr0 = comp b (pr0 (dom (Λ a b c g)) (dom b))*
have *lam-pr1*: *in-hom Λ-pr1 (prod a b) (exp b c)*
proof (*unfold Λ-pr1-def*, *intro comp-in-homI*)
  show *in-hom (pr1 (dom (Λ a b c g)) (dom b)) (prod a b) a*
    using *assms ide-dom-lam ide-dom-b ideD(2) lam-simps(2) pr1-in-hom* **by** *auto*
  show *in-hom (Λ a b c g) a (exp b c)*
    using *assms lam-in-hom* **by** *simp*
qed
have *b-pr0*: *in-hom b-pr0 (prod a b) b*
  using *assms b-pr0-def*
  by (*metis* (*no-types*, *lifting*) *comp-in-homI hfsetcat.lam-simps(2) ideD(2)*
      ide-in-hom pr0-in-hom)
have *1*: *span Λ-pr1 b-pr0*
  using *lam-pr1 b-pr0*
  by (*metis* (*no-types*, *lifting*) *in-homE*)
have *tuple*: *in-hom (tuple Λ-pr1 b-pr0) (prod a b) (prod (exp b c) b)*
  using *1 lam-pr1 b-pr0 tuple-in-hom* [*of Λ-pr1 b-pr0*]
  by (*metis* (*mono-tags*, *lifting*) *in-homE*)
define *Λ-pr1′* **where** *Λ-pr1′ = comp (Λ a b c g) (pr1 a b)*
define *b-pr0′* **where** *b-pr0′ = pr0 a b*
have *lam-pr1-eq*: *Λ-pr1 = Λ-pr1′*
  using *assms Λ-pr1-def Λ-pr1′-def ideD(2) lam-simps(2)* **by** *auto*
have *b-pr0-eq*: *b-pr0 = b-pr0′*
  using *assms b-pr0-def b-pr0′-def b-pr0 comp-ide-arr*
  by (*metis* (*no-types*, *lifting*) *ideD(2) in-homE lam-simps(2)*)
have *Fun-pr0*: *Fun (pr0 a b) = restrict (λx. arr-of (hsnd (elem-of x))) (set (prod a b))*
  using *assms Fun-pr0* **by** *simp*
have *Fun-lam-pr1*: *Fun Λ-pr1 =*
                restrict (Fun (Λ a b c g) o
                        restrict (λx. arr-of (hfst (elem-of x))) (set (prod a b)))
                (set (prod a b))
  using *assms 1 Fun-comp Fun-pr1 lam-pr1-eq Λ-pr1′-def*
  by (*metis* (*no-types*, *lifting*) *pr1-simps(2)*)
have *comp (eval b c) (HF′.prod (Λ a b c g) b) = comp (eval b c) (tuple Λ-pr1 b-pr0)*
  using *assms Λ-pr1-def b-pr0-def 1 prod-in-terms-of-tuple ideD(1) lam-simps(1)*
  by *presburger*
also have *5*: *... = comp (eval b c) (tuple Λ-pr1′ b-pr0′)*
  using *lam-pr1-eq b-pr0-eq* **by** *simp*
also have *... = g*
proof (*intro arr-eqI*$_{SC}$)
  have *2*: *arr (comp (eval b c) (tuple Λ-pr1 b-pr0))*
    using *assms tuple arr-char*
    by (*metis* (*no-types*, *lifting*) *in-homE seqI eval-simps(1−2) ide-exp prod-ide-eq*)
  have *3*: *arr g*
    using *assms* **by** *blast*
  have *tuple′*: *in-hom (tuple Λ-pr1′ b-pr0′) (prod a b) (prod (exp b c) b)*
    using *tuple lam-pr1-eq b-pr0-eq* **by** *blast*
  have *4*: *Dom g = set (prod a b)*
    using *assms*

```
  by (metis (no-types, lifting) in-homE)
show par: par (comp (eval b c) (tuple Λ-pr1' b-pr0')) g
  using assms tuple' 2 3 5
  by (metis (no-types, lifting) cod-comp dom-comp in-homE eval-simps(3))
show Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) = Fun g
proof
  fix x
  have x ∉ set (prod a b) ⟹ Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x = Fun g x
  proof −
    have 5: Fun g ∈ extensional (Dom g)
      using assms 3 Fun-mapsto by simp
    moreover have Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) ∈ extensional (Dom g)
      using 5 par Fun-mapsto by (metis (no-types, lifting) Int-iff)
    ultimately show x ∉ set (prod a b) ⟹
                    Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x = Fun g x
      using 4 extensional-arb [of Fun g Dom g x]
            extensional-arb [of Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) Dom g x]
      by force
  qed
  moreover have x ∈ set (prod a b) ⟹
                  Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x = Fun g x
  proof −
    assume x: x ∈ set (prod a b)
    have 6: Dom (tuple Λ-pr1' b-pr0') = set (prod a b)
      using assms 4 tuple' par
      by (metis (no-types, lifting) in-homE)
    have Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x =
         Fun (eval b c) (Fun (tuple Λ-pr1' b-pr0') x)
    proof −
      have Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x =
           (Fun (eval b c) ∘ Fun (tuple Λ-pr1' b-pr0')) x
        using assms par x 6 Fun-comp [of eval b c tuple Λ-pr1' b-pr0'] by auto
      also have ... = Fun (eval b c) (Fun (tuple Λ-pr1' b-pr0') x)
        by simp
      finally show ?thesis by blast
    qed
    also have ... = restrict (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of x))))
                             (set (HF'.prod (exp b c) b))
                             (Fun (tuple Λ-pr1' b-pr0') x)
      using assms Fun-eval by simp
    also have ... = (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of x))))
                    (Fun (tuple Λ-pr1' b-pr0') x)
    proof −
      have Fun (tuple Λ-pr1' b-pr0') x ∈ set (HF'.prod (exp b c) b)
      proof −
        have x ∈ Dom (tuple Λ-pr1' b-pr0')
          using x 6 by blast
        moreover have Cod (tuple Λ-pr1' b-pr0') = set (HF'.prod (exp b c) b)
          by (metis (no-types, lifting) in-homE assms(2−3) ide-exp
```

```
        prod-ide-eq tuple')
    moreover have arr (tuple Λ-pr1' b-pr0')
      using tuple' by blast
    ultimately show ?thesis
      using tuple' Fun-mapsto [of tuple Λ-pr1' b-pr0'] by auto
  qed
  thus ?thesis
    using restrict-apply by simp
qed
also have ... = (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of x))))
                  (arr-of ⟨elem-of (Fun Λ-pr1' x), elem-of (Fun b-pr0' x)⟩)
proof −
  have 7: Dom Λ-pr1' = set (prod a b)
    using assms
    by (metis (no-types, lifting) 1 comp-ide-arr ideD(2)
        b-pr0-def lam-pr1-eq lam-simps(2) pr0-simps(2))
  moreover have span Λ-pr1' b-pr0'
    using assms 1 b-pr0-eq lam-pr1-eq by auto
  moreover have x ∈ Dom Λ-pr1'
    using x 7 by simp
  ultimately have Fun (tuple Λ-pr1' b-pr0') x =
                  arr-of ⟨elem-of (Fun Λ-pr1' x), elem-of (Fun b-pr0' x)⟩
    using assms x restrict-apply Fun-tuple by simp
  thus ?thesis by simp
qed
also have ... = arr-of (happ (elem-of (Fun Λ-pr1' x)) (elem-of (Fun b-pr0' x)))
  using assms by simp
also have ... = arr-of (happ (elem-of (arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b)
                                                    (ide-to-hf c) (arr-to-hfun g))
                                              (hfst (elem-of x)))))
                             (elem-of (arr-of (hsnd (elem-of x)))))
proof −
  have Fun b-pr0' x = arr-of (hsnd (elem-of x))
    using assms x Fun-pr0 b-pr0'-def by simp
  moreover have Fun Λ-pr1' x =
                arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c)
                                   (arr-to-hfun g))
                             (hfst (elem-of x)))
  proof −
    have Fun Λ-pr1' x =
          restrict (Fun (Λ a b c g) o Fun (pr1 a b)) (Dom (pr1 a b)) x
      using assms x Fun-pr1 Fun-comp lam-pr1-eq Fun-lam-pr1 pr1-simps(1−2)
      by presburger
    also have ... = Fun (Λ a b c g) (Fun (pr1 a b) x)
      using assms x restrict-apply Fun-lam-pr1 Fun-pr1 calculation lam-pr1-eq
      by auto
    also have ... = restrict (λx. arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b)
                                                     (ide-to-hf c) (arr-to-hfun g))
                                               (elem-of x)))
```

```
                                (set a)
                                (Fun (pr1 a b) x)
          using assms x Fun-lam by simp
        also have ... = arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c)
                                       (arr-to-hfun g))
                                (elem-of (Fun (pr1 a b) x)))
        proof −
          have Fun (pr1 a b) x ∈ set a
          proof −
            have x ∈ Dom (pr1 a b)
              using assms x pr1-simps(1−2) by auto
            moreover have Cod (pr1 a b) = set a
              using assms HF'.cod-pr1 pr1-simps(1) by auto
            moreover have arr (pr1 a b)
              using assms arr-char by blast
            ultimately show ?thesis
              using Fun-mapsto [of pr1 a b] by auto
          qed
          thus ?thesis
            using restrict-apply by simp
        qed
        also have ... = arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c)
                                       (arr-to-hfun g))
                                (hfst (elem-of x)))
          using assms x Fun-pr1 Fun-lam [of a b c g] by simp
        finally show ?thesis by simp
      qed
      ultimately show ?thesis by simp
    qed
    also have ... = arr-of (happ (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c)
                                         (arr-to-hfun g))
                                  (hfst (elem-of x)))
                            (hsnd (elem-of x)))
      by simp
    also have ... = arr-of (happ (arr-to-hfun g) (elem-of x))
      using assms x happ-hlam
      by (metis (no-types, lifting) prod-def elem-of-membI HCollect-iff ide-dom
          in-homE UNIV-I arr-to-hfun-in-hexp hexp-def hfst-conv hsnd-conv
          ide-to-hf-hf-to-ide timesE)
    also have ... = Fun g x
      using assms x 3 4 Fun-char [of g] restrict-apply [of Fun g Dom g x]
      by simp
    finally show ?thesis by simp
  qed
  ultimately show Fun (comp (eval b c) (tuple Λ-pr1' b-pr0')) x = Fun g x
    by auto
qed
finally show ?thesis by simp
```

qed

lemma *lam-eval-prod*:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom h a* (*exp b c*)
shows Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*)) = *h*
proof (*intro arr-eqI*$_{SC}$)
have *0*: *in-hom* (*comp* (*eval b c*) (*HF′.prod h b*)) (*prod a b*) *c*
proof
show *in-hom* (*HF′.prod h b*) (*prod a b*) (*HF′.prod* (*exp b c*) *b*)
proof
show *1*: *arr* (*HF′.prod h b*)
using *assms HF′.prod-in-hom′*
by (*metis* (*no-types*, *lifting*) *ideD*(*1*) *in-homE*)
show *dom* (*HF′.prod h b*) = *prod a b*
using *assms 1*
by (*metis* (*no-types*, *lifting*) *HF′.prod-simps*(*2*) *ideD*(*1−2*) *in-homE prod-ide-eq*)
show *cod* (*HF′.prod h b*) = *HF′.prod* (*exp b c*) *b*
using *assms 1*
by (*metis* (*no-types*, *lifting*) *HF′.prod-simps*(*3*) *ideD*(*1*,*3*) *in-homE*)
qed
show *in-hom* (*eval b c*) (*HF′.prod* (*exp b c*) *b*) *c*
using *assms* **by** *blast*
qed
have *1*: *in-hom* (Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*))) *a* (*exp b c*)
using *assms 0* **by** *blast*
have *2*: *Fun* (*comp* (*eval b c*) (*HF′.prod h b*)) =
restrict (*Fun* (*eval b c*) ∘ *Fun* (*HF′.prod h b*))
(*set* (*HF′.prod a b*))
proof −
have *seq* (*eval b c*) (*HF′.prod h b*)
using *assms 1*
by (*metis* (*no-types*, *lifting*) *0 in-homE*)
moreover have *Dom* (*HF′.prod h b*) = *set* (*HF′.prod a b*)
using *assms*
by (*metis* (*no-types*, *lifting*) *HF′.prod-simps*(*2*) *ideD*(*1−2*) *in-homE*)
ultimately show *?thesis*
using *assms Fun-comp* [*of eval b c HF′.prod h b*] **by** *simp*
qed
show *par*: *par* (Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*))) *h*
using *assms 1*
by (*metis* (*no-types*, *lifting*) *in-homE*)
show *Fun* (Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*))) = *Fun h*
proof
fix *x*
show *Fun* (Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*))) *x* = *Fun h x*
proof −
have *x* ∉ *set a* ⟹ *?thesis*
using *assms 1 Fun-mapsto*

```
      extensional-arb [of Fun h set a x]
      extensional-arb [of Fun (Λ a b c (comp (eval b c) (HF'.prod h b)))
                       set a x]
  by (metis (no-types, lifting) 0 Int-iff lam-simps(2) par)
moreover have x ∈ set a ⟹ ?thesis
proof −
  assume x: x ∈ set a
  have 3: dom (comp (eval b c) (HF'.prod h b)) = HF'.prod a b
    using assms 0 in-homE prod-ide-eq by auto
  have 4: cod (comp (eval b c) (HF'.prod h b)) = c
    using assms 0 by blast
  have 5: dom (comp (eval b c) (HF'.prod h b)) = HF'.prod a b
    using assms 3
    by (metis (mono-tags, lifting))
  have 6: cod (comp (eval b c) (HF'.prod h b)) = c
    using assms 4 by (metis (no-types, lifting))
  have 7: arr-to-hfun (comp (eval b c) (HF'.prod h b)) =
          ⦇xy ∈ ide-to-hf (HF'.prod a b) ∗ ide-to-hf c.
            hsnd xy = elem-of (Fun (comp (eval b c) (HF'.prod h b)) (arr-of (hfst xy)))⦈
    unfolding arr-to-hfun-def
    using 2 5 6 by metis
  have Fun (Λ a b c (comp (eval b c) (HF'.prod h b))) x =
       arr-of (happ (hlam (ide-to-hf a) (ide-to-hf b) (ide-to-hf c)
                      (arr-to-hfun (comp (eval b c) (HF'.prod h b))))
                (elem-of x))
    using assms 0 x Fun-lam by auto
  also have ... = arr-of ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
                      ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩
                        ∈ arr-to-hfun (comp (eval b c) (HF'.prod h b))⦈
  proof −
    have seq (eval b c) (HF'.prod h b)
      using assms 0 by blast
    moreover have ide-to-hf (dom (comp (eval b c) (HF'.prod h b))) =
                  ide-to-hf a ∗ ide-to-hf b
      using assms 1 3
      by (metis (no-types, lifting) prod-def UNIV-I ide-to-hf-hf-to-ide prod-ide-eq)
    moreover have ide-to-hf (cod (comp (eval b c) (HF'.prod h b))) = ide-to-hf c
      using assms 2 4 by auto
    ultimately show ?thesis
      using assms 0 x happ-hlam(3) elem-of-membI
            hfun-arr-to-hfun [of comp (eval b c) (HF'.prod h b)]
      by simp
  qed
  also have ... = arr-of ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
                      hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                             (arr-of ⟨elem-of x, hfst yz⟩))⦈
  proof −
    have ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
           ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩
```

```
        ∈ arr-to-hfun (comp (eval b c) (HF'.prod h b))⦈ =
      {⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
                hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                       (arr-of ⟨elem-of x, hfst yz⟩))⦈
  proof
    fix yz
    show yz ∈ ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
                  ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩
                    ∈ arr-to-hfun (comp (eval b c) (HF'.prod h b))⦈ ⟷
         yz ∈ {⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
                  hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                         (arr-of ⟨elem-of x, hfst yz⟩))⦈}
    proof −
      have yz ∈ ide-to-hf b ∗ ide-to-hf c ⟹
          ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩ ∈ arr-to-hfun (comp (eval b c) (HF'.prod h b))
            ⟷ hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                    (arr-of ⟨elem-of x, hfst yz⟩))
      proof −
        assume yz: yz ∈ ide-to-hf b ∗ ide-to-hf c
        have ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩
                ∈ arr-to-hfun (comp (eval b c) (HF'.prod h b))
              ⟷
             ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩ ∈ ide-to-hf (HF'.prod a b) ∗ ide-to-hf c ∧
             hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                    (arr-of ⟨elem-of x, hfst yz⟩))
          using 7 by auto
        moreover have ⟨⟨elem-of x, hfst yz⟩, hsnd yz⟩
                        ∈ ide-to-hf (prod a b) ∗ ide-to-hf c
        proof −
          have ⟨elem-of x, hfst yz⟩ ∈ ide-to-hf (HF'.prod a b)
            using assms x yz
            by (metis (no-types, lifting) prod-def elem-of-membI UNIV-I hfst-conv
                ide-to-hf-hf-to-ide prod-ide-eq timesE times-iff)
          thus ?thesis
            using yz assms(1−2) prod-ide-eq by auto
        qed
        ultimately show ?thesis
          using assms(1−2) prod-ide-eq by auto
      qed
      thus ?thesis by auto
    qed
  qed
  thus ?thesis by simp
qed
also have ... = arr-of {⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈}
proof −
  have {⦇yz ∈ ide-to-hf b ∗ ide-to-hf c.
          hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                 (arr-of ⟨elem-of x, hfst yz⟩))⦈ =
```

```
    {|yz ∈ ide-to-hf b * ide-to-hf c. yz ∈ elem-of (Fun h x)|}
proof −
  have ⋀yz. yz ∈ ide-to-hf b * ide-to-hf c ⟹
              hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                                     (arr-of ⟨elem-of x, hfst yz⟩))
            ⟷
              yz ∈ elem-of (Fun h x)
  proof −
    fix yz
    assume yz: yz ∈ ide-to-hf b * ide-to-hf c
    have 7: arr-of ⟨elem-of x, hfst yz⟩ ∈ set (HF'.prod a b)
      using assms x yz arr-of-membI
      by (metis (no-types, lifting) prod-def elem-of-membI UNIV-I hfst-conv
          ide-to-hf-hf-to-ide prod-ide-eq timesE times-iff)
    have 8: Fun h x ∈ set (exp b c)
    proof −
      have Fun h x ∈ Cod h
        using assms x Fun-mapsto by blast
      moreover have Cod h = set (exp b c)
        using assms 0 lam-simps(3) par by auto
      ultimately show ?thesis by blast
    qed
    show hsnd yz = elem-of (Fun (comp (eval b c) (HF'.prod h b))
                               (arr-of ⟨elem-of x, hfst yz⟩))
          ⟷
         yz ∈ elem-of (Fun h x)
    proof −
      have Fun (comp (eval b c) (HF'.prod h b)) (arr-of ⟨elem-of x, hfst yz⟩) =
           arr-of (happ (elem-of (Fun h x)) (hfst yz))
      proof −
        have Fun (comp (eval b c) (HF'.prod h b)) (arr-of ⟨elem-of x, hfst yz⟩) =
             restrict (Fun (eval b c) ∘ Fun (HF'.prod h b))
                      (set (HF'.prod a b))
                      (arr-of ⟨elem-of x, hfst yz⟩)
          using assms x yz 2 by simp
        also have ... = Fun (eval b c)
                            (Fun (HF'.prod h b) (arr-of ⟨elem-of x, hfst yz⟩))
          using 7 by simp
        also have ... = Fun (eval b c)
                            (arr-of ⟨elem-of (Fun h x),
                                 elem-of (Fun b (arr-of (hfst yz)))⟩)
        proof −
          have Fun (HF'.prod h b) (arr-of ⟨elem-of x, hfst yz⟩) =
               arr-of ⟨elem-of (Fun h x), elem-of (Fun b (arr-of (hfst yz)))⟩
          proof −
            have Fun (HF'.prod h b) (arr-of ⟨elem-of x, hfst yz⟩) =
                 arr-of ⟨elem-of (Fun h (arr-of (hfst (elem-of (arr-of ⟨elem-of x, hfst
yz⟩))))),
                         elem-of (Fun b (arr-of (hsnd (elem-of (arr-of ⟨elem-of x, hfst
```

```
yz⟩)))))⟩
                proof −
                  have arr-of ⟨elem-of x, hfst yz⟩ ∈ set (prod (dom h) (dom b))
                    using assms x yz 7
                    by (metis (no-types, lifting) ideD(2) in-homE prod-ide-eq)
                  thus ?thesis
                    using assms x yz Fun-prod ideD(1) by blast
                qed
                also have ... = arr-of ⟨elem-of (Fun h (arr-of (elem-of x))),
                                        elem-of (Fun b (arr-of (hfst yz)))⟩
                  using assms x yz by simp
                  also have ... = arr-of ⟨elem-of (Fun h x), elem-of (Fun b (arr-of (hfst
yz)))⟩
                  using assms(1) set-ideD(1) x by force
                finally show ?thesis by simp
              qed
              thus ?thesis by simp
            qed
            also have ... = Fun (eval b c) (arr-of ⟨elem-of (Fun h x), hfst yz⟩)
              using assms x yz Fun-ide ide-char arr-of-membI by auto
              also have ... = restrict (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of
x))))
                                       (set (HF'.prod (exp b c) b))
                                       (arr-of ⟨elem-of (Fun h x), hfst yz⟩)
              using assms Fun-eval [of b c] by simp
            also have ... = (λx. arr-of (happ (hfst (elem-of x)) (hsnd (elem-of x))))
                            (arr-of ⟨elem-of (Fun h x), hfst yz⟩)
            proof −
              have arr-of ⟨elem-of (Fun h x), hfst yz⟩
                     ∈ set (HF'.prod (exp b c) b)
              proof −
                have 1: ide-to-hf (HF'.prod (exp b c) b) =
                        HF (elem-of ‘ set (HF'.prod (exp b c) b))
                  unfolding ide-to-hf-def by blast
                have ⟨elem-of (Fun h x), hfst yz⟩
                       ∈ HF (elem-of ‘ set (HF'.prod (exp b c) b))
                  using assms x yz 1 8 Fun-mapsto [of h]
                  by (metis (no-types, lifting) prod-def elem-of-membI UNIV-I
                      hfst-conv ide-exp ide-to-hf-hf-to-ide prod-ide-eq timesE times-iff)
                thus ?thesis
                  using assms x yz 1 arr-of-membI [of ⟨elem-of (Fun h x), hfst yz⟩]
                  by auto
              qed
              thus ?thesis by simp
            qed
            also have ... = arr-of (happ (elem-of (Fun h x)) (hfst yz))
              by simp
            finally show ?thesis by simp
          qed
```

```
      hence 9: elem-of (Fun (comp (eval b c) (HF'.prod h b))
                            (arr-of ⟨elem-of x, hfst yz⟩)) =
               happ (elem-of (Fun h x)) (hfst yz)
        by simp
      show ?thesis
      proof −
        have hsnd yz = happ (elem-of (Fun h x)) (hfst yz)
              ⟷ yz ∈ elem-of (Fun h x)
        proof
          have 10: ∃!z. ⟨hfst yz, z⟩ ∈ elem-of (Fun h x)
          proof −
            have hfun (ide-to-hf b) (ide-to-hf c) (elem-of (Fun h x))
              using assms x 8
              by (metis (no-types, lifting) elem-of-membI HCollect-iff UNIV-I
                  exp-def hexp-def ide-exp ide-to-hf-hf-to-ide)
            thus ?thesis
              using assms yz
                    hfunE [of ide-to-hf b ide-to-hf c elem-of (Fun h x)]
              by (metis (no-types, lifting) hfst-conv timesE)
          qed
          show yz ∈ elem-of (Fun h x)
                ⟹ hsnd yz = happ (elem-of (Fun h x)) (hfst yz)
          proof −
            assume yz1: yz ∈ elem-of (Fun h x)
            show hsnd yz = happ (elem-of (Fun h x)) (hfst yz)
              unfolding app-def
              using assms x yz yz1 10 hfun-arr-to-hfun arr-to-hfun-def
                    the1-equality
                      [of λy. ⟨hfst yz, y⟩ ∈ elem-of (Fun h x) hsnd yz]
              by (metis (no-types, lifting) hfst-conv hsnd-conv timesE)
          qed
          show hsnd yz = happ (elem-of (Fun h x)) (hfst yz)
                ⟹ yz ∈ elem-of (Fun h x)
            unfolding app-def
            using assms x yz 10
                  theI' [of λy. ⟨hfst yz, y⟩ ∈ elem-of (Fun h x)]
            by (metis (no-types, lifting) hfst-conv hsnd-conv timesE)
        qed
        thus ?thesis
          using 9 by simp
      qed
    qed
    thus ?thesis by blast
  qed
  thus ?thesis by simp
qed
also have ... = Fun h x
proof −
```

```
    have H: Fun h x = restrict (λx. arr-of (happ (arr-to-hfun h) (elem-of x))) (Dom h)
x
    proof −
      have arr h
        using assms by blast
      thus ?thesis
        using assms x Fun-char by simp
    qed
    also have ... = arr-of (happ (arr-to-hfun h) (elem-of x))
      using assms x par
      by (metis (no-types, lifting) 0 lam-simps(2) restrict-apply)
    also have ... = arr-of (THE g. ⟨elem-of x, g⟩ ∈ arr-to-hfun h)
      using app-def by simp
    also have ... = arr-of ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈
    proof −
      have ex-un-g: ∃!g. ⟨elem-of x, g⟩ ∈ arr-to-hfun h
        using assms x arr-to-hfun-def hfun-arr-to-hfun
              hfunE [of ide-to-hf a ide-to-hf (exp b c) arr-to-hfun h]
        by (metis (no-types, lifting) elem-of-membI in-homE)
      moreover have
        ⟨elem-of x, ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈⟩
          ∈ arr-to-hfun h
      proof −
        have elem-of (Fun h x) =
              ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈
        proof
          fix yz
          show yz ∈ elem-of (Fun h x) ⟷
                yz ∈ ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈
          proof
            show yz ∈ elem-of (Fun h x)
                  ⟹ yz ∈ ⦇yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦈
            proof −
              assume yz: yz ∈ elem-of (Fun h x)
              have yz ∈ ide-to-hf b ∗ ide-to-hf c
              proof −
                have elem-of (Fun h x) ∈ hexp (ide-to-hf b) (ide-to-hf c)
                proof −
                  have ide (hf-to-ide (hexp (ide-to-hf b) (ide-to-hf c)))
                    using assms exp-def ide-exp by auto
                  moreover have
                    Fun h x ∈ set (hf-to-ide (hexp (ide-to-hf b) (ide-to-hf c)))
                  proof −
                    have Fun h x ∈ Cod h
                      using assms x Fun-mapsto by blast
                    moreover have
                      Cod h = set (hf-to-ide (hexp (ide-to-hf b) (ide-to-hf c)))
                      using assms 0 exp-def lam-simps(3) par by auto
                    ultimately show ?thesis by blast
```

```
            qed
            ultimately show ?thesis
              using elem-of-membI [of hf-to-ide (hexp (ide-to-hf b) (ide-to-hf c))
                                      Fun h x]
              by (simp add: ide-to-hf-hf-to-ide)
          qed
          thus ?thesis
            using assms yz hexp-def by auto
        qed
        thus ?thesis
          using assms x yz by blast
      qed
      show yz ∈ ⦃yz ∈ ide-to-hf b ∗ ide-to-hf c. yz ∈ elem-of (Fun h x)⦄
            ⟹ yz ∈ elem-of (Fun h x)
        using assms by simp
    qed
  qed
  moreover have arr-of (elem-of x) = x
    using arr-of-elem-of assms(1) set-ideD(1) x by blast
  ultimately show ?thesis
    using assms x arr-to-hfun-def ex-un-g by auto
qed
ultimately show ?thesis
  using assms x theI' [of λg. ⟨elem-of x, g⟩ ∈ arr-to-hfun h]
  by fastforce
qed
finally show ?thesis
  using assms x by simp
qed
finally show ?thesis by simp
qed
ultimately show Fun (Λ a b c (comp (eval b c) (HF'.prod h b))) x = Fun h x
  by blast
qed
qed
qed
```

25.5 The Main Results

```
interpretation cartesian-closed-category comp
proof −
  interpret elementary-category-with-terminal-object comp some-terminal some-terminator
    using extends-to-elementary-category-with-terminal-object by blast
  interpret elementary-cartesian-closed-category comp pr0 pr1
                some-terminal some-terminator exp eval Λ
    using ide-exp eval-in-hom lam-in-hom prod-ide-eq eval-prod-lam lam-eval-prod
    by unfold-locales auto
  show cartesian-closed-category comp
    using is-cartesian-closed-category by simp
```

```
    qed

    theorem is-cartesian-closed-category:
    shows cartesian-closed-category comp
      ..

    theorem is-category-with-finite-limits:
    shows category-with-finite-limits comp
    proof
      fix J :: 'j comp
      assume J: category J
      interpret J: category J
        using J by simp
      assume finite: finite (Collect J.arr)
      have has-products (Collect J.ide)
      proof −
        have Collect J.ide ≠ UNIV
          using J.not-arr-null by blast
        moreover have finite (Collect J.ide)
        proof −
          have Collect J.ide ⊆ Collect J.arr
            by auto
          thus ?thesis
            using finite J.ideD(1) finite-subset by blast
        qed
        ultimately show ?thesis
          using finite has-finite-products' by simp
      qed
      moreover have has-products (Collect J.arr)
      proof −
        have Collect J.arr ≠ UNIV
          using J.not-arr-null by blast
        thus ?thesis
          using finite has-finite-products' by simp
      qed
      ultimately show has-limits-of-shape J
        using J.category-axioms has-limits-if-has-products [of J] by simp
    qed

  end

end
theory HF-SetCat-Interp
imports HF-SetCat
begin
```

Here we demonstrate the possibility of making a top-level interpretation of the *hfsetcat* locale. See theory *SetCat-Interp* for further discussion on why we do this.

```
  interpretation HF-Sets: hfsetcat .
```

end

Chapter 26

ZFC SetCat

In the statement and proof of the Yoneda Lemma given in theory *Yoneda*, we sidestepped the issue, of not having a category of "all" sets, by axiomatizing the notion of a "set category", showing that for every category we could obtain a hom-functor into a set category at a higher type, and then proving the Yoneda lemma for that particular hom-functor. This is perhaps the best we can do within HOL, because HOL does not provide any type that contains a universe of sets with the closure properties usually associated with a category *Set* of sets and functions between them. However, a significant aspect of category theory involves considering "all" algebraic structures of a particular kind as the objects of a "large" category having nice closure or completeness properties. Being able to consider a category of sets that is "small-complete", or a cartesian closed category of sets and functions that includes some infinite sets as objects, are basic examples of this kind of situation.

The purpose of this section is to demonstrate that, although it cannot be done in pure HOL, if we are willing to accept the existence of a type V whose inhabitants correspond to sets satisfying the axioms of ZFC, then it is possible to construct, for example, the "large" category of sets and functions as it is usually understood in category theory. Moreover, assuming the existence of such a type is essentially all we have to do; all the category theory we have developed so far still applies. Specifically, what we do in this section is to use theory *ZFC-in-HOL*, which provides an axiomatization of a set-theoretic universe V, to construct a "set category" *ZFC-SetCat*, whose objects correspond to V-sets, whose arrows correspond to functions between V-sets, and which has the small-completeness property traditionally ascribed to the category of all small sets and functions between them.

theory *ZFC-SetCat*
imports *ZFC-in-HOL.ZFC-Cardinals Limit*
begin

The following locale constructs the category of classes and functions between them and shows that it is small complete. The category is obtained simply as the replete set category at type V. This is not yet the category of sets we want, because it contains objects corresponding to "large" V-sets.

```
locale ZFC-class-cat
begin

  sublocale replete-setcat ‹TYPE(V)› .

  lemma admits-small-V-tupling:
  assumes small (I :: V set)
  shows admits-tupling I
  proof (unfold admits-tupling-def)
    let ?π = λf. UP (VLambda (ZFC-in-HOL.set I) (DN o f))
    have ?π ∈ (I → Univ) ∩ extensional' I → Univ
      using UP-mapsto by force
    moreover have inj-on ?π ((I → Univ) ∩ extensional' I)
    proof
      fix f g
      assume f: f ∈ (I → Univ) ∩ extensional' I
      assume g: g ∈ (I → Univ) ∩ extensional' I
      assume eq: UP (VLambda (ZFC-in-HOL.set I) (DN ∘ f)) =
                 UP (VLambda (ZFC-in-HOL.set I) (DN ∘ g))
      have 1: VLambda (ZFC-in-HOL.set I) (DN ∘ f) = VLambda (ZFC-in-HOL.set I) (DN
∘ g)
        using f g eq
        by (meson injD inj-UP)
      show f = g
      proof
        fix x
        have x ∉ I ⟹ f x = g x
          using f g extensional'-def [of I] by auto
        moreover have x ∈ I ⟹ f x = g x
        proof −
          assume x: x ∈ I
          hence (DN o f) x = (DN o g) x
            using assms 1 x elts-of-set VLambda-eq-D2 by fastforce
          thus f x = g x
            using f g x comp-apply UP-DN
            by (metis IntD1 PiE bij-arr-of bij-betw-imp-surj-on)
        qed
        ultimately show f x = g x by blast
      qed
    qed
    ultimately show ∃π. π ∈ (I → Univ) ∩ extensional' I → Univ ∧
                        inj-on π ((I → Univ) ∩ extensional' I)
      by blast
  qed

  corollary admits-small-tupling:
  assumes small I
  shows admits-tupling I
  proof −
```

```
  obtain φ where φ: inj-on φ I ∧ φ ‘ I ∈ range elts
    using assms small-def by metis
  have admits-tupling (φ ‘ I)
    using φ admits-small-V-tupling by fastforce
  moreover have inv: bij-betw (inv-into I φ) (φ ‘ I) I
    by (simp add: φ bij-betw-inv-into inj-on-imp-bij-betw)
  ultimately show ?thesis
    using admits-tupling-respects-bij by blast
qed

lemma has-small-products:
assumes small (I :: 'i set) and I ≠ UNIV
shows has-products I
proof −
  have 1: ⋀I :: V set. small I ⟹ has-products I
    using big-UNIV
    by (metis admits-small-tupling has-products-iff-admits-tupling)
  obtain V-of where V-of: inj-on V-of I ∧ V-of ‘ I ∈ range elts
    using assms small-def by metis
  have bij-betw (inv-into I V-of) (V-of ‘ I) I
    using V-of bij-betw-inv-into bij-betw-imageI by metis
  moreover have small (V-of ‘ I)
    using assms by auto
  ultimately show ?thesis
    using assms 1 has-products-preserved-by-bijection by blast
qed

theorem has-small-limits:
assumes small (UNIV :: 'i set)
shows has-limits (undefined :: 'i)
proof −
  have ∀ I :: 'i set. I ≠ UNIV ⟶ admits-tupling I
    using assms smaller-than-small subset-UNIV admits-small-tupling by auto
  thus ?thesis
    using assms has-limits-iff-admits-tupling by blast
qed

end
```

We now construct the desired category of small sets and functions between them, as a full subcategory of the category of classes and functions. To show that this subcategory is small complete, we show that the inclusion creates small products; that is, a small product of objects corresponding to small sets itself corresponds to a small set.

```
locale ZFC-set-cat
begin

interpretation Cls: ZFC-class-cat .

definition setp
```

where *setp A* ≡ *A* ⊆ *Cls.Univ* ∧ *small A*

sublocale *sub-set-category Cls.comp* ‹λ*A*. *A* ⊆ *Cls.Univ*› *setp*
using *small-Un smaller-than-small setp-def*
apply *unfold-locales*
apply *simp-all*
apply *force*
by *auto*

lemma *is-sub-set-category*:
shows *sub-set-category Cls.comp* (λ*A*. *A* ⊆ *Cls.Univ*) *setp*
using *sub-set-category-axioms* **by** *blast*

interpretation *incl*: *full-inclusion-functor Cls.comp* ‹λ*a*. *Cls.ide a* ∧ *setp* (*Cls.set a*)›
..

The following functions establish a bijection between the identities of the category and the elements of type V; which in turn are in bijective correspondence with small V-sets.

definition *V-of-ide* :: *V setcat.arr* ⇒ *V*
where *V-of-ide a* ≡ *ZFC-in-HOL.set* (*Cls.DN* ‘ *Cls.set a*)

definition *ide-of-V* :: *V* ⇒ *V setcat.arr*
where *ide-of-V A* ≡ *Cls.mkIde* (*Cls.UP* ‘ *elts A*)

lemma *bij-betw-ide-V*:
shows *V-of-ide* ∈ *Collect ide* → *UNIV*
and *ide-of-V* ∈ *UNIV* → *Collect ide*
and [*simp*]: *ide a* ⟹ *ide-of-V* (*V-of-ide a*) = *a*
and [*simp*]: *V-of-ide* (*ide-of-V A*) = *A*
and *bij-betw V-of-ide* (*Collect ide*) *UNIV*
and *bij-betw ide-of-V UNIV* (*Collect ide*)
proof −
have *Univ* = *Cls.Univ*
using *terminal-char* **by** *presburger*
show *1*: *V-of-ide* ∈ *Collect ide* → *UNIV*
by *blast*
show *2*: *ide-of-V* ∈ *UNIV* → *Collect ide*
proof
fix *A* :: *V*
have *small* (*elts A*)
by *blast*
have *Cls.UP* ‘ *elts A* ⊆ *Univ* ∧ *small* (*Cls.UP* ‘ *elts A*)
using *Cls.UP-mapsto terminal-char* **by** *blast*
hence *ide* (*mkIde* (*Cls.UP* ‘ *elts A*))
using *ide-mkIde* ‹*Univ* = *Cls.Univ*› *setp-def* **by** *auto*
thus *ide-of-V A* ∈ *Collect ide*
using *ide-of-V-def ide-char*$_{SSC}$ *setp-def*
by (*metis* (*no-types*, *lifting*) *Cls.ide-mkIde Cls.set-mkIde arr-mkIde ide-char*′

```
      mem-Collect-eq)
qed
show 3: ⋀a. ide a ⟹ ide-of-V (V-of-ide a) = a
proof −
  fix a
  assume a: ide a
  have ide-of-V (V-of-ide a) =
        Cls.mkIde (Cls.UP ‘ elts (ZFC-in-HOL.set (Cls.DN ‘ Cls.set a)))
    unfolding ide-of-V-def V-of-ide-def by simp
  also have ... = Cls.mkIde (Cls.UP ‘ Cls.DN ‘ Cls.set a)
    using setp-set-ide a ide-char_SSC setp-def by force
  also have ... = Cls.mkIde (Cls.set a)
  proof −
    have Cls.set a ⊆ Cls.Univ
      using a ide-char_SSC setp-def by blast
    hence Cls.UP ‘ Cls.DN ‘ Cls.set a = Cls.set a
    proof −
      have ⋀x. x ∈ Cls.set a ⟹ x ∈ Cls.UP ‘ Cls.DN ‘ Cls.set a
        using Cls.UP-DN ‹Cls.set a ⊆ Cls.Univ›
        by (metis Cls.bij-arr-of bij-betw-def image-inv-into-cancel)
      moreover have ⋀x. x ∈ Cls.UP ‘ Cls.DN ‘ Cls.set a ⟹ x ∈ Cls.set a
        using ‹Cls.set a ⊆ Cls.Univ›
        by (metis Cls.bij-arr-of bij-betw-def image-inv-into-cancel)
      ultimately show ?thesis by blast
    qed
    thus ?thesis by argo
  qed
  also have ... = a
    using a Cls.mkIde-set ide-char_SbC by blast
  finally show ide-of-V (V-of-ide a) = a by simp
qed
show 4: ⋀A. V-of-ide (ide-of-V A) = A
proof −
  fix A
  have V-of-ide (ide-of-V A) =
        ZFC-in-HOL.set (Cls.DN ‘ Cls.set (Cls.mkIde (Cls.UP ‘ elts A)))
    unfolding V-of-ide-def ide-of-V-def by simp
  also have ... = ZFC-in-HOL.set (Cls.DN ‘ Cls.UP ‘ elts A)
    using Cls.set-mkIde [of Cls.UP ‘ elts A] Cls.UP-mapsto by fastforce
  also have ... = ZFC-in-HOL.set (elts A)
    using Cls.DN-UP by force
  also have ... = A by simp
  finally show V-of-ide (ide-of-V A) = A by blast
qed
show bij-betw V-of-ide (Collect ide) UNIV
  using 1 2 3 4
  by (intro bij-betwI) auto
show bij-betw ide-of-V UNIV (Collect ide)
  using 1 2 3 4
```

by (*intro bij-betwI*) *blast+*
qed

Next, we establish bijections between the hom-sets of the category and certain subsets of V whose elements represent functions.

definition *V-of-arr* :: *V setcat.arr* ⇒ *V*
where *V-of-arr f* ≡ *VLambda* (*V-of-ide* (*dom f*)) (*Cls.DN o Cls.Fun f o Cls.UP*)

definition *arr-of-V* :: *V setcat.arr* ⇒ *V setcat.arr* ⇒ *V* ⇒ *V setcat.arr*
where *arr-of-V a b F* ≡ *Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP o app F o Cls.DN*)

definition *vfun*
where *vfun A B f* ≡ *f* ∈ *elts* (*VPow* (*vtimes A B*)) ∧ *elts A* = *Domain* (*pairs f*) ∧
single-valued (*pairs f*)

lemma *small-Collect-vfun*:
shows *small* (*Collect* (*vfun A B*))
unfolding *vfun-def*
by (*metis* (*full-types*) *small-Collect small-elts*)

lemma *vfunI*:
assumes *f* ∈ *elts A* → *elts B*
shows *vfun A B* (*VLambda A f*)
proof (*unfold vfun-def*, *intro conjI*)
show *VLambda A f* ∈ *elts* (*VPow* (*vtimes A B*))
using *assms VLambda-def* **by** *auto*
show *elts A* = *Domain* (*pairs* (*VLambda A f*))
using *assms VLambda-def* [*of A f*]
by (*metis Domain-fst fst-pairs-VLambda*)
show *single-valued* (*pairs* (*VLambda A f*))
using *assms VLambda-def single-valued-def pairs-iff-elts* **by** *fastforce*
qed

lemma *app-vfun-mapsto*:
assumes *vfun A B F*
shows *app F* ∈ *elts A* → *elts B*
proof
have *F* ∈ *elts* (*VPow* (*vtimes A B*)) ∧ *elts A* = *Domain* (*pairs F*) ∧ *single-valued* (*pairs F*)
using *assms vfun-def* **by** *simp*
hence *1*: *F* ∈ *elts* (*VPi A* (λ-. *B*)) ∧ *elts A* = *Domain* (*pairs F*) ∧ *single-valued* (*pairs F*)
unfolding *VPi-def*
by (*metis* (*no-types*, *lifting*) *down elts-of-set mem-Collect-eq subsetI*)
fix *x*
assume *x*: *x* ∈ *elts A*
have *x* ∈ *Domain* (*pairs F*)
using *assms x vfun-def* **by** *blast*
thus *app F x* ∈ *elts B*
using *x 1 VPi-D* [*of F A* λ-. *B x*] **by** *blast*

qed

lemma *bij-betw-hom-vfun*:
shows *V-of-arr* ∈ *hom a b* → *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))
and ⟦*ide a*; *ide b*⟧ ⟹ *arr-of-V a b* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)) → *hom a b*
and *f* ∈ *hom a b* ⟹ *arr-of-V a b* (*V-of-arr f*) = *f*
and ⟦*ide a*; *ide b*; *F* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))⟧
  ⟹ *V-of-arr* (*arr-of-V a b F*) = *F*
and ⟦*ide a*; *ide b*⟧
  ⟹ *bij-betw V-of-arr* (*hom a b*) (*Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)))
and ⟦*ide a*; *ide b*⟧
  ⟹ *bij-betw* (*arr-of-V a b*) (*Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))) (*hom a b*)
proof −
  show *1*: ⋀*a b*. *V-of-arr* ∈ *hom a b* → *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))
  proof
    fix *a b f*
    assume *f*: *f* ∈ *hom a b*
    have *V-of-arr f* = *VLambda* (*V-of-ide* (*dom f*)) (*Cls.DN* ∘ *Cls.Fun f* ∘ *Cls.UP*)
      unfolding *V-of-arr-def* **by** *simp*
    moreover have *vfun* (*V-of-ide a*) (*V-of-ide b*) ...
    proof −
      have *Cls.DN* ∘ *Cls.Fun f* ∘ *Cls.UP* ∈ *elts* (*V-of-ide a*) → *elts* (*V-of-ide b*)
      proof
        fix *x*
        assume *x*: *x* ∈ *elts* (*V-of-ide a*)
        have (*Cls.DN* ∘ *Cls.Fun f* ∘ *Cls.UP*) *x* = *Cls.DN* (*Cls.Fun f* (*Cls.UP x*))
          by *simp*
        moreover have ... ∈ *elts* (*V-of-ide b*)
        proof −
          have *Cls.UP x* ∈ *Cls.Dom f*
            by (*metis* (*no-types*, *lifting*) *Cls.arr-dom-iff-arr Cls.arr-mkIde Cls.set-mkIde*
              bij-betw-ide-V(*3*) *arr-char*$_{SbC}$ *dom-simp f ide-char*$_{SSC}$ *ide-of-V-def image-eqI*
              in-homE mem-Collect-eq x)
          hence *Cls.DN* (*Cls.Fun f* (*Cls.UP x*)) ∈ *Cls.DN* ‘ *Cls.Cod f*
            using *f in-hom-char*$_{FSbC}$ *arr-char*$_{SbC}$ *Cls.Fun-mapsto* [*of f*] **by** *blast*
          thus *Cls.DN* (*Cls.Fun f* (*Cls.UP x*)) ∈ *elts* (*V-of-ide b*)
            by (*metis* (*no-types*, *lifting*) *V-of-ide-def arrE cod-char*$_{SbC}$ *elts-of-set f*
              in-homE mem-Collect-eq replacement setp-def)
        qed
        ultimately show (*Cls.DN* ∘ *Cls.Fun f* ∘ *Cls.UP*) *x* ∈ *elts* (*V-of-ide b*) **by** *argo*
      qed
      thus *?thesis*
        using *f vfunI* **by** *blast*
    qed
    ultimately show *V-of-arr f* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)) **by** *simp*
  qed
  show *2*: ⋀*a b*. ⟦*ide a*; *ide b*⟧
    ⟹ *arr-of-V a b* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)) → *hom a b*
  proof −

fix *a b*
assume *a*: *ide a* **and** *b*: *ide b*
show *arr-of-V a b* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)) → *hom a b*
proof
fix *F*
assume *F*: *F* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))
have *3*: *app F* ∈ *elts* (*V-of-ide a*) → *elts* (*V-of-ide b*)
using *F app-vfun-mapsto* [*of V-of-ide a V-of-ide b F*] **by** *blast*
have *4*: *app F* ∈ *Cls.DN* ‘ (*Cls.set a*) → *Cls.DN* ‘ (*Cls.set b*)
using *3 V-of-ide-def a b ide-char*$_{SSC}$ *setp-def* **by** *auto*
have *arr-of-V a b F* = *Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*)
unfolding *arr-of-V-def* **by** *simp*
moreover have ... ∈ *hom a b*
proof
show «*Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*) : *a* → *b*»
proof
have *4*: *Cls.arr* (*Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*))
proof −
have *Cls.set a* ⊆ *Cls.Univ* ∧ *Cls.set b* ⊆ *Cls.Univ*
using *a b ide-char*$_{SSC}$ *setp-def* **by** *blast*
moreover have *Cls.UP* ∘ *app F* ∘ *Cls.DN* ∈ *Cls.set a* → *Cls.set b*
proof
fix *x*
assume *x*: *x* ∈ *Cls.set a*
have (*Cls.UP* ∘ *app F* ∘ *Cls.DN*) *x* = *Cls.UP* (*app F* (*Cls.DN x*))
by *simp*
moreover have ... ∈ *Cls.set b*
by (*metis* (*no-types*, *lifting*) *4 Cls.arr-mkIde Cls.ide-char*′ *Cls.set-mkIde*
PiE V-of-ide-def bij-betw-ide-V(*3*) *b elts-of-set ide-char*$_{SSC}$
ide-of-V-def replacement rev-image-eqI x setp-def)
ultimately show (*Cls.UP* ∘ *app F* ∘ *Cls.DN*) *x* ∈ *Cls.set b*
by *auto*
qed
ultimately show *?thesis* **by** *blast*
qed
show *5*: *arr* (*Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*))
using *a b 4 arr-char*$_{SbC}$ *ide-char*$_{SbC}$ **by** *auto*
show *dom* (*Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*)) = *a*
using *a 4 5 dom-char*$_{SbC}$ *ide-char*$_{SbC}$ **by** *auto*
show *cod* (*Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP* ∘ *app F* ∘ *Cls.DN*)) = *b*
using *b 4 5 cod-char*$_{SbC}$ *ide-char*$_{SbC}$ **by** *auto*
qed
qed
ultimately show *arr-of-V a b F* ∈ *hom a b* **by** *auto*
qed
qed
show *3*: ⋀*a b f*. *f* ∈ *hom a b* ⟹ *arr-of-V a b* (*V-of-arr f*) = *f*
proof −
fix *a b f*

assume f: $f \in hom\ a\ b$
have *4*: $\bigwedge x.\ x \in Cls.set\ a$
$\Longrightarrow (Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN)\ x = Cls.Fun\ f\ x$
proof −
fix x
assume x: $x \in Cls.set\ a$
have $(Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN)\ x =$
$Cls.UP\ (Cls.DN\ (Cls.Fun\ f\ (Cls.UP\ (Cls.DN\ x))))$
by *simp*
also have ... = $Cls.UP\ (Cls.DN\ (Cls.Fun\ f\ x))$
using x *Cls.UP-DN*
by (*metis* (*no-types*, *lifting*) *Cls.elem-set-implies-incl-in Cls.incl-in-def*
Cls.setp-set-ide bij-betw-def replete-setcat.bij-arr-of subset-eq)
also have ... = $Cls.Fun\ f\ x$
proof −
have $x \in Cls.Dom\ f$
using $x\ f$ $dom\text{-}char_{SbC}$ **by** *fastforce*
hence $Cls.Fun\ f\ x \in Cls.Cod\ f$
using $x\ f$ *Cls.Fun-mapsto* $in\text{-}hom\text{-}char_{FSbC}$ **by** *blast*
hence $Cls.Fun\ f\ x \in Cls.Univ$
using f $cod\text{-}char_{SbC}$ $in\text{-}hom\text{-}char_{FSbC}$
by (*metis* (*no-types*, *lifting*) *Cls.elem-set-implies-incl-in Cls.incl-in-def*
Cls.setp-set-ide subsetD)
thus *?thesis*
by (*meson bij-betw-inv-into-right replete-setcat.bij-arr-of*)
qed
finally show $(Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN)\ x = Cls.Fun\ f\ x$
by *blast*
qed
have *5*: $Cls.arr\ (Cls.mkArr\ (Cls.set\ a)\ (Cls.set\ b)$
$(Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN))$
proof −
have $Cls.set\ a \subseteq Cls.Univ \wedge Cls.set\ b \subseteq Cls.Univ$
using f $ide\text{-}char_{SbC}$ *codomains-def domains-def in-hom-def* **by** *force*
moreover have $Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN$
$\in Cls.set\ a \to Cls.set\ b$
proof
fix x
assume x: $x \in Cls.set\ a$
hence *6*: $x \in Cls.Dom\ f$
using f **by** (*metis* (*no-types*, *lifting*) $dom\text{-}char_{SbC}$ *in-homE mem-Collect-eq*)
have $(Cls.UP \circ (Cls.DN \circ Cls.Fun\ f \circ Cls.UP) \circ Cls.DN)\ x = Cls.Fun\ f\ x$
using *4* $f\ x$ **by** *blast*
moreover have ... $\in Cls.Cod\ f$
using *4 6* f *Cls.Fun-mapsto*
by (*metis* (*no-types*, *lifting*) *Cls.arr-dom-iff-arr Cls.elem-set-implies-incl-in*
Cls.ideD(*1*) *Cls.incl-in-def IntE PiE*)
moreover have ... = $Cls.set\ b$
using f **by** (*metis* (*no-types*, *lifting*) $cod\text{-}char_{SbC}$ *in-homE mem-Collect-eq*)

```
    ultimately show (Cls.UP ∘ (Cls.DN ∘ Cls.Fun f ∘ Cls.UP) ∘ Cls.DN) x ∈ Cls.set b
      by auto
  qed
  ultimately show ?thesis by blast
qed
have arr-of-V a b (V-of-arr f) =
      Cls.mkArr (Cls.set a) (Cls.set b)
            (Cls.UP ∘ app (VLambda (V-of-ide (dom f)) (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                  ∘ Cls.DN)
  unfolding arr-of-V-def V-of-arr-def by simp
also have ... = Cls.mkArr (Cls.set a) (Cls.set b)
                        (Cls.UP ∘ (Cls.DN ∘ Cls.Fun f ∘ Cls.UP) ∘ Cls.DN)
proof (intro Cls.mkArr-eqI')
  show 6: ⋀x. x ∈ Cls.set a ⟹
            (Cls.UP ∘ app (VLambda (V-of-ide (dom f)) (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                  ∘ Cls.DN) x =
              (Cls.UP ∘ (Cls.DN ∘ Cls.Fun f ∘ Cls.UP) ∘ Cls.DN) x
  proof -
    fix x
    assume x: x ∈ Cls.set a
    have (Cls.UP ∘ app (VLambda (V-of-ide (dom f)) (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                ∘ Cls.DN) x =
          Cls.UP (app (VLambda (V-of-ide (dom f)) (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                  (Cls.DN x))
      by simp
    also have ... = (Cls.UP ∘ (Cls.DN ∘ Cls.Fun f ∘ Cls.UP) ∘ Cls.DN) x
    proof -
      have Cls.DN x ∈ elts (V-of-ide (dom f))
        using f
       by (metis (no-types, lifting) V-of-ide-def elts-of-set ide-char_SSC ide-dom image-eqI
          in-homE mem-Collect-eq replacement x setp-def)
      thus ?thesis
        using beta by auto
    qed
    ultimately show (Cls.UP ∘ app (VLambda (V-of-ide (dom f))
                                    (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                            ∘ Cls.DN) x =
                    (Cls.UP ∘ (Cls.DN ∘ Cls.Fun f ∘ Cls.UP) ∘ Cls.DN) x
      by argo
  qed
  show Cls.arr (Cls.mkArr (Cls.set a) (Cls.set b)
              (Cls.UP ∘ app (VLambda (V-of-ide (local.dom f))
                                    (Cls.DN ∘ Cls.Fun f ∘ Cls.UP))
                    ∘ Cls.DN))
    using 5 6 Cls.mkArr-eqI by auto
qed
also have ... = Cls.mkArr (Cls.set a) (Cls.set b) (Cls.Fun f)
  using 4 5 by force
also have ... = f
```

```
    using f Cls.mkArr-Fun
  by (metis (no-types, lifting) arr-char_SbC cod-simp dom-char_SbC in-homE mem-Collect-eq)
  finally show arr-of-V a b (V-of-arr f) = f by blast
qed
show 4: ⋀a b F. ⟦ide a; ide b; F ∈ Collect (vfun (V-of-ide a) (V-of-ide b))⟧
                ⟹ V-of-arr (arr-of-V a b F) = F
proof −
  fix a b F
  assume a: ide a and b: ide b
  assume F: F ∈ Collect (vfun (V-of-ide a) (V-of-ide b))
  have F ∈ elts (VPow (vtimes (V-of-ide a) (V-of-ide b))) ∧
        elts (V-of-ide a) = Domain (pairs F) ∧ single-valued (pairs F)
    using F vfun-def by simp
  hence 5: F ∈ elts (VPi (V-of-ide a) (λ-. V-of-ide b))
    unfolding VPi-def
    by (metis (no-types, lifting) down elts-of-set mem-Collect-eq subsetI)
  let ?f = Cls.mkArr (Cls.set a) (Cls.set b) (Cls.UP ∘ app F ∘ Cls.DN)
  have 6: Cls.arr ?f
  proof −
    have Cls.set a ⊆ Cls.Univ ∧ Cls.set b ⊆ Cls.Univ
      using a b ide-char_SSC setp-def by blast
    moreover have Cls.UP ∘ app F ∘ Cls.DN ∈ Cls.set a → Cls.set b
    proof
      fix x
      assume x: x ∈ Cls.set a
      have (Cls.UP ∘ app F ∘ Cls.DN) x = Cls.UP (app F (Cls.DN x))
        by simp
      moreover have ... ∈ Cls.set b
      proof −
        have app F (Cls.DN x) ∈ Cls.DN ‘ Cls.set b
          using a b ide-char_SSC x F app-vfun-mapsto [of V-of-ide a V-of-ide b F]
                V-of-ide-def setp-def
          by auto
        thus ?thesis
          using ‹Cls.set a ⊆ Cls.Univ ∧ Cls.set b ⊆ Cls.Univ›
          by (metis Cls.bij-arr-of bij-betw-def imageI image-inv-into-cancel)
      qed
      ultimately show (Cls.UP ∘ app F ∘ Cls.DN) x ∈ Cls.set b by auto
    qed
    ultimately show ?thesis by blast
  qed
  have V-of-arr (arr-of-V a b F) = VLambda (V-of-ide (dom ?f)) (Cls.DN ∘ Cls.Fun ?f ∘
Cls.UP)
    unfolding V-of-arr-def arr-of-V-def by simp
  also have ... = VLambda (V-of-ide a) (Cls.DN ∘ Cls.Fun ?f ∘ Cls.UP)
    unfolding V-of-ide-def
    using a b 6 ide-char_SSC V-of-ide-def dom-char_SbC Cls.dom-mkArr arrI_SbC by auto
  also have ... = VLambda (V-of-ide a)
                          (Cls.DN ∘
```

```
                    restrict (Cls.UP ∘ app F ∘ Cls.DN) (Cls.set a) ∘ Cls.UP)
    using 6 Cls.Fun-mkArr by simp
  also have ... = VLambda (V-of-ide a) (app F)
  proof −
    have 7: ⋀x. x ∈ elts (V-of-ide a) ⟹
                (Cls.DN ∘ restrict (Cls.UP ∘ app F ∘ Cls.DN) (Cls.set a) ∘ Cls.UP) x =
                app F x
      unfolding V-of-ide-def
      using a
      apply simp
      by (metis (no-types, lifting) Cls.bij-arr-of a bij-betw-def empty-iff ide-char_SSC
          image-eqI image-inv-into-cancel setp-def)
    have 8: ⋀x. x ∈ elts (V-of-ide a) ⟹
                (Cls.DN ∘ restrict (Cls.UP ∘ app F ∘ Cls.DN) (Cls.set a) ∘ Cls.UP) x
                  ∈ elts (V-of-ide b)
      using 5 7 VPi-D by fastforce
    have VLambda (V-of-ide a) (app F) ∈ elts (VPi (V-of-ide a) (λ-. V-of-ide b))
      using 5 VPi-I VPi-D by auto
    moreover have VLambda (V-of-ide a)
                    (Cls.DN ∘
                       restrict (Cls.UP ∘ app F ∘ Cls.DN) (Cls.set a) ∘ Cls.UP)
                  ∈ elts (VPi (V-of-ide a) (λ-. V-of-ide b))
      using 8 VPi-I by auto
    moreover have ⋀x. x ∈ elts (V-of-ide a) ⟹
                  app (VLambda (V-of-ide a)
                     (Cls.DN ∘
                        restrict (Cls.UP ∘ app F ∘ Cls.DN) (Cls.set a) ∘
                          Cls.UP)) x =
                  app F x
      using 7 beta by auto
    ultimately show ?thesis
      using fun-ext by simp
  qed
  also have ... = F
    using 5 eta [of F V-of-ide a λ-. V-of-ide b] by auto
  finally show V-of-arr (arr-of-V a b F) = F by blast
qed
show ⟦ide a; ide b⟧
      ⟹ bij-betw V-of-arr (hom a b) (Collect (vfun (V-of-ide a) (V-of-ide b)))
  using 1 2 3 4
  apply (intro bij-betwI)
    apply blast
   apply blast
  by auto
show ⟦ide a; ide b⟧
      ⟹ bij-betw (arr-of-V a b) (Collect (vfun (V-of-ide a) (V-of-ide b))) (hom a b)
  using 1 2 3 4
  apply (intro bij-betwI)
    apply blast
```

```
    apply blast
  by auto
qed

lemma small-hom:
shows small (hom a b)
proof (cases ide a ∧ ide b)
  assume 1: ¬ (ide a ∧ ide b)
  have ⋀f. ¬ «f : a → b»
    using 1 in-hom-def ide-cod ide-dom by blast
  hence hom a b = {}
    by blast
  thus ?thesis by simp
  next
  assume 1: ide a ∧ ide b
  show ?thesis
    using 1 bij-betw-hom-vfun(5) small-Collect-vfun small-def
    by (metis (no-types, lifting) bij-betw-def small-iff-range)
qed
```

We can now show that the inclusion of the subcategory into the ambient category *Cls* creates small products. To do this, we consider a product in *Cls* of objects of the subcategory indexed by a small set *I*. Since *Cls* is a replete set category, by a previous result we know that the elements of a product object p in *Cls* correspond to its points; that is, to the elements of *hom unity p*. The elements of *hom unity p* in turn correspond to I-tuples. By carrying out the construction of the set of I-tuples in V and exploiting the bijections between homs of the subcatgory and V-sets, we can obtain an injection of *hom unity p* to the extension of a V-set, thus showing *hom unity p* is small. Since *hom unity p* is small, it determines an object of the subcategory, which must then be a product in the subcategory, in view of the fact that the subcategory is full.

```
lemma has-small-V-products:
assumes small (I :: V set)
shows has-products I
proof (unfold has-products-def, intro conjI impI allI)
  show I ≠ UNIV
    using assms big-UNIV by blast
  fix J D
  assume D: discrete-diagram J comp D ∧ Collect (partial-composition.arr J) = I
  interpret J: category J
    using D discrete-diagram-def by blast
  interpret D: discrete-diagram J comp D
    using D by blast
  interpret incloD: composite-functor J comp Cls.comp D map ..
  interpret incloD: discrete-diagram J Cls.comp incloD.map
    using D.is-discrete
    by unfold-locales auto
  interpret incloD: diagram-in-set-category J Cls.comp ‹λA. A ⊆ Cls.Univ› incloD.map
    ..
```

have *1*: *small* (*Collect J.ide*)
using *assms D D.is-discrete* **by** *argo*
show ∃ *a*. *has-as-product J D a*
proof −
have *2*: ∃ *a*. *Cls.has-as-product J incloD.map a*
proof −
have *Collect J.ide* ≠ *UNIV*
using *J.ide-def* **by** *blast*
thus *?thesis*
using *1 D.is-discrete Cls.has-small-products* [*of Collect J.ide*]
Cls.has-products-def [*of Collect J.ide*] *incloD.discrete-diagram-axioms*
by *presburger*
qed
obtain Π*D* **where** Π*D*: *Cls.has-as-product J incloD.map* Π*D*
using *2* **by** *blast*
interpret Π*D*: *constant-functor J Cls.comp* Π*D*
using Π*D Cls.product-is-ide*
by *unfold-locales auto*
obtain π **where** π: *product-cone J Cls.comp incloD.map* Π*D* π
using Π*D Cls.has-as-product-def* **by** *blast*
interpret π: *product-cone J Cls.comp incloD.map* Π*D* π
using π **by** *blast*
have *small* (*Cls.hom Cls.unity* Π*D*)
proof −
obtain φ **where** φ: *bij-betw* φ (*Cls.hom Cls.unity* Π*D*) (*incloD.cones Cls.unity*)
using *incloD.limits-are-sets-of-cones* π*.limit-cone-axioms* **by** *blast*
let *?J* = *ZFC-in-HOL.set* (*Collect J.arr*)
let *?V-of-point* = λ*x*. *VLambda ?J* (λ*j*. *V-of-arr* (φ *x j*))
let *?Tuples* = *VPi ?J* (λ*j*. *ZFC-in-HOL.set* (*V-of-arr* ‘ *hom Cls.unity* (*D j*)))
have *V-of-point*: *?V-of-point* ∈ *Cls.hom Cls.unity* Π*D* → *elts ?Tuples*
proof
fix *x*
assume *x*: *x* ∈ *Cls.hom Cls.unity* Π*D*
have φ*x*: φ *x* ∈ *incloD.cones Cls.unity*
using φ *x*
unfolding *bij-betw-def* **by** *blast*
interpret φ*x*: *cone J Cls.comp incloD.map Cls.unity* ‹φ *x*›
using φ*x* **by** *blast*
have ⋀*j*. *J.arr j* ⟹ *V-of-arr* (φ *x j*)
∈ *elts* (*ZFC-in-HOL.set* (*V-of-arr* ‘ *hom Cls.unity* (*D j*)))
proof −
fix *j*
assume *j*: *J.arr j*
have φ *x j* ∈ *hom Cls.unity* (*D j*)
by (*metis* (*mono-tags*, *lifting*) *D.preserves-ide* φ*x.component-in-hom cod-simp*
ideD(*1*,*3*) *in-hom-char*$_{FSbC}$ *incloD.is-discrete incloD.preserves-cod j map-simp*
mem-Collect-eq o-apply terminal-char2 terminal-unity$_{SSC}$)
moreover have *V-of-arr* ‘ *hom Cls.unity* (*D j*) =
elts (*ZFC-in-HOL.set* (*V-of-arr* ‘ *hom Cls.unity* (*D j*)))

```
      using small-hom replacement by simp
    ultimately show V-of-arr (φ x j)
                    ∈ elts (ZFC-in-HOL.set (V-of-arr ‘ hom Cls.unity (D j)))
      using j φx bij-betw-hom-vfun(1) by blast
  qed
  thus ?V-of-point x ∈ elts ?Tuples
    using VPi-I by fastforce
qed
have ?V-of-point ‘ Cls.hom Cls.unity ΠD ∈ range elts
proof −
  have ?V-of-point ‘ Cls.hom Cls.unity ΠD ⊆ elts ?Tuples
    using V-of-point by blast
  thus ?thesis
    using smaller-than-small down-raw by auto
qed
moreover have inj-on ?V-of-point (Cls.hom Cls.unity ΠD)
proof
  fix x y
  assume x: x ∈ Cls.hom Cls.unity ΠD and y: y ∈ Cls.hom Cls.unity ΠD
  and eq: ?V-of-point x = ?V-of-point y
  have φx: φ x ∈ incloD.cones Cls.unity
    using φ x
    unfolding bij-betw-def by blast
  have φy: φ y ∈ incloD.cones Cls.unity
    using φ y
    unfolding bij-betw-def by blast
  interpret φx: cone J Cls.comp incloD.map Cls.unity ‹φ x›
    using φx by blast
  interpret φy: cone J Cls.comp incloD.map Cls.unity ‹φ y›
    using φy by blast
  have φ x = φ y
  proof −
    have ⋀j. j ∈ elts ?J ⟹ φ x j = φ y j
    proof −
      fix j
      assume j: j ∈ elts ?J
      hence 3: J.arr j
        by (simp add: 1 incloD.is-discrete)
      have 4: ide (D j)
        using 3 incloD.is-discrete D.preserves-ide by force
      have 5:ide Cls.unity
        using Cls.terminal-unity_SC terminal-char terminal-def by auto
      have φxj: φ x j ∈ hom Cls.unity (D j)
      using 3 4 5 incloD.is-discrete φx.preserves-hom φx.A.map-simp in-hom-char_FSbC
        by (metis (no-types, lifting) J.ide-char φx.component-in-hom ideD(1) map-simp
            mem-Collect-eq o-apply)
      have φyj: φ y j ∈ hom Cls.unity (D j)
      using 3 4 5 incloD.is-discrete φy.preserves-hom φx.A.map-simp in-hom-char_FSbC
        by (metis (no-types, lifting) J.ide-char φy.component-in-hom ideD(1) map-simp
```

```
        mem-Collect-eq o-apply)
    show φ x j = φ y j
    proof −
      have ⋀j. j ∈ elts ?J ⟹ V-of-arr (φ x j) = V-of-arr (φ y j)
        using x eq VLambda-eq-D2 by blast
      thus ?thesis
        using V-of-arr-def
        by (metis (mono-tags, lifting) j φxj φyj bij-betw-hom-vfun(3) mem-Collect-eq)
    qed
  qed
  moreover have elts ?J = Collect J.arr
    by (simp add: 1 incloD.is-discrete)
  ultimately show ?thesis
    using φx.extensionality φy.extensionality
    by (metis HOL.ext mem-Collect-eq)
 qed
 thus x = y
  using x y φ bij-betw-imp-inj-on inj-on-def
  by (metis (no-types, lifting))
qed
ultimately show small (Cls.hom Cls.unity ΠD)
 using small-def by blast
qed
hence small (Cls.set ΠD)
 by (simp add: Cls.set-def)
hence 2: ide ΠD
 using ide-char_SSC Cls.setp-set-ide Cls.product-is-ide ΠD
 unfolding setp-def
 by blast
interpret ΠD': constant-functor J comp ΠD
 using 2 by unfold-locales
interpret π': cone J comp D ΠD π
proof −
 have ⋀j. J.arr j ⟹ «π j : ΠD → D j»
 proof
  fix j
  assume j: J.arr j
  show 3: arr (π j)
   by (metis (mono-tags, lifting) 2 D.as-nat-trans.preserves-cod D.is-discrete
     D.preserves-ide π.component-in-hom ideD(1) ideD(3) in-homE in-hom-char_FSbC
     j map-simp o-apply)
  show dom (π j) = ΠD
   using 3 arr-char_SbC dom-char_SbC π.preserves-dom by auto
  show cod (π j) = D j
   using 3 arr-char_SbC cod-char_SbC π.preserves-cod
   by (metis (no-types, lifting) Cls.ideD(3) D.preserves-arr functor.preserves-ide
     incloD.is-discrete incloD.is-functor incloD.preserves-cod j map-simp o-apply)
 qed
 moreover have D.mkCone π = π
```

```
    using π.extensionality null-char by auto
  ultimately show cone J comp D ΠD π
    using 2 D.cone-mkCone [of ΠD π] by simp
qed
interpret π': product-cone J comp D ΠD π
proof
  fix a χ'
  assume χ': D.cone a χ'
  interpret χ': cone J comp D a χ'
    using χ' by blast
  have a: Cls.ide a
    using ide-char_SbC χ'.A.value-is-ide by blast
  moreover have ⋀j. J.arr j ⟹ Cls.in-hom (χ' j) a (incloD.map j)
  proof −
    fix j
    assume j: J.arr j
    have «χ' j : a → D j»
      using j D.is-discrete χ'.component-in-hom by force
    thus Cls.in-hom (χ' j) a (incloD.map j)
      using a j D.is-discrete in-hom-char_FSbC map-simp by auto
  qed
  ultimately have 3: incloD.cone a (incloD.mkCone χ')
    using incloD.cone-mkCone [of a χ'] by blast
  interpret χ: cone J Cls.comp incloD.map a ‹incloD.mkCone χ'›
    using 3 by blast
  have univ: ∃!f. Cls.in-hom f a ΠD ∧ incloD.cones-map f π = incloD.mkCone χ'
    using χ.cone-axioms π.is-universal [of a incloD.mkCone χ'] by blast
  have 4: incloD.mkCone χ' = χ'
    using D.as-nat-trans.preserves-reflects-arr D.preserves-arr Limit.cone-def
          χ' χ'.extensionality identity-functor.intro identity-functor.map-def
          incloD.as-nat-trans.extensionality o-apply
    by fastforce
  have 5: D.mkCone π = π
    using π.extensionality null-char by auto
  have 6: ⋀f. Cls.in-hom f a ΠD ⟹ incloD.cones-map f π = D.cones-map f π
  proof −
    fix f
    assume f: Cls.in-hom f a ΠD
    have incloD.cones-map f π = (λj. if J.arr j then Cls.comp (π j) f else Cls.null)
      using f π.cone-axioms by auto
    also have ... = (λj. if J.arr j then comp (π j) f else null)
    proof −
      have ⋀j. J.arr j ⟹ Cls.comp (π j) f = comp (π j) f
        using f 2 comp-char in-hom-char_FSbC seq-char_SbC
        by (metis (no-types, lifting) Cls.ext Cls.in-homE χ'.ide-apex
            π'.preserves-reflects-arr arr-char_SbC ide-char_SSC)
      thus ?thesis
        using null-char by auto
    qed
```

```
      also have ... = D.cones-map f π
      proof −
        have π ∈ D.cones (cod f)
        proof −
          have «f : a → ΠD»
            using f 2 in-hom-char_FSbC χ'.ide-apex ideD(1) by presburger
          thus ?thesis
            using f π'.cone-axioms by blast
        qed
        thus ?thesis
          using ‹π ∈ D.cones (cod f)› by simp
      qed
      finally show incloD.cones-map f π = D.cones-map f π by blast
    qed
    moreover have ⋀f. «f : a → ΠD» ⟹ Cls.in-hom f a ΠD
      using in-hom-char_FSbC by blast
    show ∃!f. «f : a → ΠD» ∧ D.cones-map f π = χ'
    proof −
      have ∃f. «f : a → ΠD» ∧ D.cones-map f π = χ'
        using 2 4 5 6 univ χ'.ide-apex ideD(1) in-hom-char_FSbC
        by (metis (no-types, lifting))
      moreover have ⋀f g. ⟦«f : a → ΠD» ∧ D.cones-map f π = χ';
                           «g : a → ΠD» ∧ D.cones-map g π = χ'⟧
                             ⟹ f = g
        using 2 4 5 6 univ χ'.ide-apex ideD(1) in-hom-char_FSbC by auto
      ultimately show ?thesis by blast
    qed
  qed
  show ?thesis
    using π'.product-cone-axioms has-as-product-def by blast
 qed
qed

corollary has-small-products:
assumes small I and I ≠ UNIV
shows has-products I
proof −
 have 1: ⋀I :: V set. small I ⟹ has-products I
   using has-small-V-products by blast
 obtain φ where φ: inj-on φ I ∧ φ ‘ I ∈ range elts
   using assms small-def by metis
 have bij-betw (inv-into I φ) (φ ‘ I) I
   using φ bij-betw-inv-into bij-betw-imageI by metis
 moreover have small (φ ‘ I)
   using assms by auto
 ultimately show ?thesis
   using assms 1 has-products-preserved-by-bijection by blast
qed
```

```
  theorem has-small-limits:
  assumes category (J :: 'j comp) and small (Collect (partial-composition.arr J))
  shows has-limits-of-shape J
  proof −
    interpret J: category J
      using assms by blast
    have small (Collect J.ide)
      using assms smaller-than-small [of Collect J.arr Collect J.ide] by fastforce
    moreover have Collect J.ide ≠ UNIV
      using J.ide-def by blast
    moreover have Collect J.arr ≠ UNIV
      using J.not-arr-null by blast
    ultimately show has-limits-of-shape J
      using assms has-small-products has-limits-if-has-products [of J] by blast
  qed

  sublocale concrete-set-category comp setp UNIV Cls.UP
  proof
    show Cls.UP ∈ UNIV → Univ
      using Cls.UP-mapsto terminal-char by presburger
    show inj Cls.UP
      using Cls.inj-UP by blast
  qed

  lemma is-concrete-set-category:
  shows concrete-set-category comp setp UNIV Cls.UP
    ..

end
```

In pure HOL (without ZFC), we were able to show that every category C has a "hom functor", but there was necessarily a dependence of the target set category of the hom functor on the arrow type of C. Using the construction of the present theory, we can now show that every "locally small" category C has a hom functor, whose target is the same set category for all such C. To obtain such a hom functor requires a choice, for each hom-set *hom a b* of C, of an injection of *hom a b* to the extension of a V-set.

```
locale locally-small-category =
  category +
  assumes locally-small: ⟦ide a; ide b⟧ ⟹ small (hom b a)
begin

  interpretation Cop: dual-category C ..
  interpretation CopxC: product-category Cop.comp C ..
  interpretation S: ZFC-set-cat .

  definition Hom
  where Hom ≡ λ(b, a). S.UP o (SOME φ. φ ' hom b a ∈ range elts ∧ inj-on φ (hom b a))

  interpretation Hom: hom-functor C S.comp S.setp Hom
```

```
proof
  have 1: ⋀a b. Hom (b, a) ∈ hom b a → S.Univ ∧ inj-on (Hom (b, a)) (hom b a)
  proof −
    fix a b
    show Hom (b, a) ∈ hom b a → S.Univ ∧ inj-on (Hom (b, a)) (hom b a)
    proof (cases ide a ∧ ide b)
      show ¬ (ide a ∧ ide b) ⟹ ?thesis
        using inj-on-def by fastforce
      assume ab: ide a ∧ ide b
      show ?thesis
      proof
        have 1: ∃φ. φ ‘ hom b a ∈ range elts ∧ inj-on φ (hom b a)
          using ab locally-small [of a b] small-def [of hom b a] by blast
        let ?φ = SOME φ. φ ‘ hom b a ∈ range elts ∧ inj-on φ (hom b a)
        have φ: ?φ ‘ hom b a ∈ range elts ∧ inj-on ?φ (hom b a)
          using 1 someI-ex [of λφ. φ ‘ hom b a ∈ range elts ∧ inj-on φ (hom b a)]
          by blast
        show Hom (b, a) ∈ hom b a → S.Univ
          unfolding Hom-def
          using φ S.UP-mapsto by auto
        show inj-on (Hom (b, a)) (hom b a)
          unfolding Hom-def
          apply simp
          using ab φ S.inj-UP comp-inj-on injD inj-on-def
          by (metis (no-types, lifting))
      qed
    qed
  qed
  show ⋀f. arr f ⟹ Hom (dom f, cod f) f ∈ S.Univ
    using 1 by blast
  show ⋀b a. ⟦ide b; ide a⟧ ⟹ inj-on (Hom (b, a)) (hom b a)
    using 1 by blast
  show ⋀b a. ⟦ide b; ide a⟧ ⟹ S.setp (Hom (b, a) ‘ hom b a)
    unfolding S.setp-def
    using 1 locally-small S.terminal-char by force
qed

lemma has-ZFC-hom-functor:
shows hom-functor C S.comp S.setp Hom
  ..
```

Using this result, we can now state a more traditional version of the Yoneda Lemma in which the target category of the Yoneda functor is the same for all locally small categories.

```
interpretation Y: yoneda-functor C S.comp S.setp Hom
  ..

theorem ZFC-yoneda-lemma:
assumes ide a and functor Cop.comp S.comp F
```

```
  shows ∃φ. bij-betw φ (S.set (F a)) {τ. natural-transformation Cop.comp S.comp (Y.Y a) F
τ}
  proof −
    interpret F: functor Cop.comp S.comp F
      using assms(2) by blast
    interpret F: set-valued-functor Cop.comp S.comp S.setp F
      ..
    interpret Ya: yoneda-functor-fixed-object C S.comp S.setp Hom a
      using assms(1) by unfold-locales blast
    interpret Ya: yoneda-lemma C S.comp S.setp Hom F a
      ..
    show ?thesis
      using Ya.yoneda-lemma by blast
  qed

 end

end
theory ZFC-SetCat-Interp
imports ZFC-SetCat
begin
```

Here we demonstrate the possibility of making a top-level interpretation of the *ZFC-set-cat* locale

```
 interpretation ZFCClsCat: ZFC-class-cat .
 interpretation ZFCSetCat: ZFC-set-cat .
```

To clarify that the category *ZFCSetCat* is what it is supposed to be, we offer the following summary results.

The set of terminal objects of *ZFCSetCat* is in bijective correspondence with the elements of type *V*.

```
 lemma bij-betw-terminals-and-V:
 shows bij-betw ZFCSetCat.DN ZFCSetCat.Univ (UNIV :: V set)
   using ZFCSetCat.bij-DN
   by (metis (no-types, lifting) Collect-cong ZFCSetCat.terminal-char bij-betw-inv-into
       replete-setcat.bij-arr-of)
```

The set of elements of any object of ZFCSetCat is a small subset of the set of terminal objects.

```
 lemma ide-implies-small-set:
 assumes ZFCSetCat.ide a
 shows small (ZFCSetCat.set a) and ZFCSetCat.set a ⊆ ZFCSetCat.Univ
   using assms ZFCSetCat.ide-char ZFCSetCat.setp-set-ide ZFCSetCat.setp-def
    apply blast
   using assms ZFCSetCat.setp-imp-subset-Univ ZFCSetCat.setp-set-ide
   by blast
```

Every small set (at an arbitrary type) is in bijective correspondence with the set of elements of some object of *ZFCSetCat*.

```
lemma small-implies-bij-to-set:
assumes small A
shows ∃ a φ. ZFCSetCat.ide a ∧ bij-betw φ A (ZFCSetCat.set a)
proof −
  obtain v ψ where v: bij-betw ψ A (ZFC-in-HOL.elts v)
    by (meson assms bij-betw-the-inv-into eqpoll-def small-eqpoll)
  let ?a = ZFCSetCat.mkIde (ZFCSetCat.UP ‘ ZFC-in-HOL.elts v)
  have a: ZFCSetCat.ide ?a
    using ZFCSetCat.setp-def
    by (metis (no-types, lifting) UNIV-I ZFCSetCat.ide-mkIde bij-betw-imp-surj-on
        image-eqI image-subset-iff replacement replete-setcat.bij-arr-of small-elts)
  have bij-betw (ZFCSetCat.UP ∘ ψ) A (ZFCSetCat.set ?a)
  proof −
    have bij-betw ZFCSetCat.UP (ZFC-in-HOL.elts v) (ZFCSetCat.set ?a)
    proof −
    have ZFCSetCat.UP ‘ ZFCSetCat.DN ‘ (ZFCSetCat.set ?a) = ZFCSetCat.UP ‘ ZFC-in-HOL.elts
v
          using a ZFCSetCat.set-mkIde ZFCSetCat.DN-UP ZFCSetCat.UP-mapsto ZFCSet-
Cat.setp-def
      by (metis (no-types, lifting) ZFCSetCat.arr-mkIde ZFCSetCat.ideD(1) bij-betw-imp-surj-on
          image-inv-into-cancel replete-setcat.bij-arr-of)
      hence ZFCSetCat.set ?a = ZFCSetCat.UP ‘ ZFC-in-HOL.elts v
        using ZFCSetCat.arr-mkIde ZFCSetCat.ide-char' ZFCSetCat.set-mkIde a
        by presburger
      thus ?thesis
        using ZFCSetCat.inj-UP
            bij-betw-def [of ZFCSetCat.UP ZFC-in-HOL.elts v ZFCSetCat.set ?a]
        by (simp add: inj-on-def)
    qed
    thus ?thesis
      using bij-betw-trans v by blast
  qed
  thus ?thesis
    using a by blast
qed
```

For objects a and b of ZFCSetCat, the arrows from a to b are in bijective correspondence with the extensional functions between the underlying sets of terminal objects.

```
lemma bij-betw-hom-and-ext-funcset:
assumes ZFCSetCat.ide a and ZFCSetCat.ide b
shows bij-betw ZFCSetCat.Fun (ZFCSetCat.hom a b) (ZFCSetCat.set a →E ZFCSetCat.set
b)
proof (unfold bij-betw-def, intro conjI)
  have 1: ZFCSetCat.Dom a = ZFCSetCat.set a ∧ ZFCSetCat.Dom b = ZFCSetCat.set b
    using assms ZFCSetCat.ideD(2) by presburger
  show inj-on ZFCSetCat.Fun (ZFCSetCat.hom a b)
    apply (intro inj-onI)
    using ZFCSetCat.arr-eqI_SC by blast
  show ZFCSetCat.Fun ‘ ZFCSetCat.hom a b = ZFCSetCat.set a →E ZFCSetCat.set b
```

```
proof
  show ZFCSetCat.Fun ‘ ZFCSetCat.hom a b ⊆ ZFCSetCat.set a →E ZFCSetCat.set b
  proof −
    have ZFCSetCat.Fun ‘ ZFCSetCat.hom a b ⊆ ZFCSetCat.Dom a →E ZFCSetCat.Dom b
    proof −
      have ⋀f. f ∈ ZFCSetCat.hom a b ⟹
                ZFCSetCat.Fun f ∈ ZFCSetCat.Dom a →E ZFCSetCat.Dom b
      proof −
        have ⋀f. f ∈ ZFCSetCat.hom a b ⟹
                  ZFCSetCat.Fun f ∈
                   extensional (ZFCSetCat.Dom a) ∩ (ZFCSetCat.Dom a → ZFCSetCat.Dom
b)
        proof −
          fix f
          assume f: f ∈ ZFCSetCat.hom a b
          have ZFCSetCat.in-hom f a b
            using f by blast
          thus ZFCSetCat.Fun f ∈
                extensional (ZFCSetCat.Dom a) ∩ (ZFCSetCat.Dom a → ZFCSetCat.Dom b)
            using f 1 Int-iff Pi-iff
            apply (elim ZFCSetCat.in-homE [of f a b])
            using ZFCSetCat.Fun-mapsto [of f] by presburger
        qed
        thus ⋀f. f ∈ ZFCSetCat.hom a b ⟹
                   ZFCSetCat.Fun f ∈ ZFCSetCat.Dom a →E ZFCSetCat.Dom b
          by (simp add: PiE-def)
      qed
      thus ?thesis by blast
    qed
    thus ZFCSetCat.Fun ‘ ZFCSetCat.hom a b ⊆ ZFCSetCat.set a →E ZFCSetCat.set b
      using 1 by blast
  qed
  show ZFCSetCat.set a →E ZFCSetCat.set b ⊆ ZFCSetCat.Fun ‘ ZFCSetCat.hom a b
  proof −
    have ZFCSetCat.Dom a →E ZFCSetCat.Dom b ⊆ ZFCSetCat.Fun ‘ ZFCSetCat.hom a b
    proof
      fix F
      assume F: F ∈ ZFCSetCat.Dom a →E ZFCSetCat.Dom b
      have 2: ⋀F. F ∈ ZFCSetCat.Dom a →E ZFCSetCat.Dom b ⟹
                ∃f. ZFCSetCat.in-hom f a b ∧ ZFCSetCat.Fun f = F
      proof −
        fix F
        have 3: ZFCSetCat.set a = ZFCSetCat.Dom a ∧ ZFCSetCat.set b = ZFCSetCat.Dom
b
          using assms ZFCSetCat.ideD(2) by presburger
        assume F: F ∈ ZFCSetCat.Dom a →E ZFCSetCat.Dom b
        hence 4: F ∈ ZFCSetCat.set a →E ZFCSetCat.set b
          using 3 by blast
        hence F ∈ ZFCSetCat.set a → ZFCSetCat.set b
```

```
        by blast
      hence ∃!f. ZFCSetCat.in-hom f a b ∧ ZFCSetCat.Fun f = restrict F (ZFCSetCat.set
a)
        using assms F ZFCSetCat.fun-complete [of a b] by presburger
      moreover have restrict F (ZFCSetCat.set a) = F
        using F 4 PiE-restrict by blast
      ultimately show ∃f. ZFCSetCat.in-hom f a b ∧ ZFCSetCat.Fun f = F
        by auto
    qed
    obtain f where f: f ∈ ZFCSetCat.hom a b ∧ ZFCSetCat.Fun f = F
      using F 2 by blast
    thus F ∈ ZFCSetCat.Fun ‘ ZFCSetCat.hom a b
      by blast
  qed
  thus ?thesis
    using 1 by simp
 qed
qed

end
```

Bibliography

[1] J. Adamek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. (online edition), 2004. http://katmat.math.uni-bremen.de/acc.

[2] A. Katovsky. Category theory. *Archive of Formal Proofs*, June 2010. http://isa-afp.org/entries/Category2.shtml, Formal proof development.

[3] A. Katovsky. Category theory in Isabelle/HOL. http://apk32.user.srcf.net/Isabelle/Category/Cat.pdf, June 2010.

[4] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[5] G. O'Keefe. Category theory to Yoneda's lemma. *Archive of Formal Proofs*, Apr. 2005. http://isa-afp.org/entries/Category.shtml, Formal proof development.

[6] E. W. Stark. Bicategories. *Archive of Formal Proofs*, Jan. 2020. http://isa-afp.org/entries/Bicategory.shtml, Formal proof development.

[7] Wikipedia. Adjoint functors — Wikipedia, the free encyclopedia, 2016. http://en.wikipedia.org/w/index.php?title=Adjoint_functors&oldid=709540944, [Online; accessed 23-June-2016].