Category Theory with Adjunctions and Limits

Eugene W. Stark

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

April 18, 2020

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.

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.

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.

The present (2020) version revises the 2018 version by incorporating 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.

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.

2.1 Partial Magmas

A *partial magma* is a partial binary operation C 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 C, and we use *null* to represent "undefined". We think of the operation C as an operation of "composition", and we regard elements f and g of type $'a$ as *composable* if $C\ g\ f \neq \mathit{null}$.

type-synonym $'a\ \mathit{comp} = {'a} \Rightarrow {'a} \Rightarrow {'a}$

locale *partial-magma* =
fixes $C :: {'a}\ \mathit{comp}$ (**infixr** $\cdot$ *55*)
assumes *ex-un-null*: $\exists!n.\ \forall f.\ n \cdot f = n \wedge f \cdot n = n$
begin

definition $\mathit{null} :: {'a}$
where $\mathit{null} = (\mathit{THE}\ n.\ \forall f.\ n \cdot f = n \wedge f \cdot n = n)$

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

lemma *comp-null* [*simp*]:
shows $\mathit{null} \cdot f = \mathit{null}$ **and** $f \cdot \mathit{null} = \mathit{null}$
using *null-def ex-un-null theI'* $[\mathit{of}\ \lambda n.\ \forall f.\ n \cdot f = n \wedge f \cdot n = n]$
by *auto*

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 $\mathit{ide}\ a \equiv a \cdot a \neq \mathit{null} \wedge$
$(\forall f.\ (f \cdot a \neq \mathit{null} \longrightarrow f \cdot a = f) \wedge (a \cdot f \neq \mathit{null} \longrightarrow a \cdot f = f))$

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*   ($«\text{-} : \text{-} \rightarrow \text{-}»$)
where $«f : a \rightarrow b» \equiv a \in domains\ f \wedge b \in codomains\ f$

abbreviation *hom*
where $hom\ a\ b \equiv \{f.\ «f : a \rightarrow b»\}$

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

lemma *ide-in-hom* [*intro*]:
shows *ide* $a \longleftrightarrow \ll a : a \rightarrow a \gg$
  using *self-domain-iff-ide self-codomain-iff-ide in-hom-def ide-def* **by** *fastforce*

Arrows f g for which the composite $g \cdot f$ is defined are *sequential*.

abbreviation *seq*
where *seq g f* $\equiv$ *arr* $(g \cdot f)$

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

lemma *comp-ide-arr*:
assumes *ide b* **and** *seq b f*
shows $b \cdot f = f$
  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* $\neq$ {} *then* (*SOME a. a* $\in$ *domains f*) *else null*)

definition *cod*
where *cod f* = (*if codomains f* $\neq$ {} *then* (*SOME b. b* $\in$ *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* $\neq$ {}
shows *dom f* $\in$ *domains f*
  using *assms dom-def someI* [*of* $\lambda a.$ *a* $\in$ *domains f*] **by** *auto*

lemma *cod-in-codomains*:
assumes *codomains f* $\neq$ {}
shows *cod f* $\in$ *codomains f*
  using *assms cod-def someI* [*of* $\lambda b.$ *b* $\in$ *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-magma* +
assumes *ext*: $g \cdot f \neq null \Longrightarrow seq\ g\ f$
and *has-domain-iff-has-codomain*: $domains\ f \neq \{\} \longleftrightarrow codomains\ f \neq \{\}$
and *match-1*: $[\![\ seq\ h\ g;\ seq\ (h \cdot g)\ f\ ]\!] \Longrightarrow seq\ g\ f$
and *match-2*: $[\![\ seq\ h\ (g \cdot f);\ seq\ g\ f\ ]\!] \Longrightarrow seq\ h\ g$
and *match-3*: $[\![\ seq\ g\ f;\ seq\ h\ g\ ]\!] \Longrightarrow seq\ (h \cdot g)\ f$
and *comp-assoc′*: $[\![\ seq\ g\ f;\ seq\ h\ g\ ]\!] \Longrightarrow (h \cdot g) \cdot f = h \cdot g \cdot f$
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 $(h \cdot g) \cdot f = h \cdot g \cdot f$
  by (*metis comp-assoc′ ex-un-null ext match-1 match-2*)

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

lemma *domains-comp*:
assumes *seq g f*
shows $domains\ (g \cdot f) = domains\ f$
proof −
  have $domains\ (g \cdot f) = \{a.\ ide\ a \wedge seq\ (g \cdot f)\ a\}$
    using *domains-def ext* **by** *auto*
  also have ... $= \{a.\ ide\ a \wedge seq\ f\ a\}$
    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* $(g \cdot f)$ = *codomains g*
proof –
  have *codomains* $(g \cdot f) = \{b.\ ide\ b \wedge seq\ b\ (g \cdot f)\}$
    using *codomains-def ext* **by** *auto*
  also have ... $= \{b.\ ide\ b \wedge seq\ b\ g\}$
    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* $\neq \{\} \longleftrightarrow$ *arr f*
  by (*simp add*: *arr-def has-domain-iff-has-codomain*)

lemma *has-codomain-iff-arr*:
shows *codomains f* $\neq \{\} \longleftrightarrow$ *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 $a \in$ *domains f* **and** $a' \in$ *domains f*
shows $a = a'$
proof –
  have $ide\ a \wedge seq\ f\ a \wedge ide\ a' \wedge seq\ f\ a'$
    using *assms domains-def ext* **by** *force*
  thus *?thesis*
    using *match-1 ide-def not-arr-null* **by** *metis*
qed

lemma *codomain-unique*:
assumes $b \in$ *codomains f* **and** $b' \in$ *codomains f*
shows $b = b'$
proof –
  have $ide\ b \wedge seq\ b\ f \wedge ide\ b' \wedge seq\ b'\ f$
    using *assms codomains-def ext* **by** *force*
  thus *?thesis*
    using *match-2 ide-def not-arr-null* **by** *metis*
qed

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

lemma *codomains-char*:
assumes *arr f*
shows *codomains f* = {*cod f*}
  using *assms 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*) $\longleftrightarrow$ *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*) $\wedge$ *seq* (*cod f*) *f*
using *assms*(*1*) *has-codomain-iff-arr codomains-def cod-in-codomains ext* **by** *blast*
moreover have *ide* (*cod f*) $\wedge$ *seq g* (*cod f*)
using *assms*(*2*−*3*) *domains-def domains-char 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 $\ll f : a \rightarrow b \gg$ **and** $\ll g : b \rightarrow c \gg$
shows *seq g f*
using *assms* **by** *fastforce*

lemma *compatible-iff-seq*:
shows *domains g* $\cap$ *codomains f* $\neq$ {} $\longleftrightarrow$ *seq g f*
proof
show *domains g* $\cap$ *codomains f* $\neq$ {} $\Longrightarrow$ *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* $\Longrightarrow$ *domains g* $\cap$ *codomains f* $\neq$ {}
proof −
assume *gf*: *seq g f*
have *1*: *cod f* $\in$ *codomains f*
using *gf has-domain-iff-arr domains-comp cod-in-codomains codomains-char* **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*≫
proof
show *1*: *seq g f* **using** *assms compatible-iff-seq* **by** *blast*
show *dom* (*g* · *f*) = *a*
using *assms 1 domains-comp domains-char* **by** *blast*
show *cod* (*g* · *f*) = *c*
using *assms 1 codomains-comp codomains-char* **by** *blast*
qed

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 $b \cdot f = f$
using *assms cod-in-codomains has-codomain-iff-arr ide-def codomains-def* **by** *auto*

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

lemma *dom-dom*:
assumes *arr f*
shows *dom* (*dom f*) = *dom f*
using *assms* **by** *simp*

lemma *cod-cod*:

assumes *arr f*
shows *cod* (*cod f*) = *cod f*
  using *assms* **by** *simp*

lemma *dom-cod*:
assumes *arr f*
shows *dom* (*cod f*) = *cod f*
  using *assms* **by** *simp*

lemma *cod-dom*:
assumes *arr f*
shows *cod* (*dom f*) = *dom f*
  using *assms* **by** *simp*

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

lemma *cod-comp* [*simp*]:
assumes *seq g f*
shows *cod* ($g \cdot f$) = *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 *ide* ($g \cdot f$)
and *seq g f* $\Longrightarrow$ *seq f g* $\Longrightarrow$ $g \cdot f$ = *dom f* $\Longrightarrow$ $g \cdot f$ = *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 canonically 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 $\ll f : a \to b \gg$.

abbreviation *endo*
where *endo* $f \equiv$ *seq f f*

abbreviation *antipar*
where *antipar f g* $\equiv$ *seq g f* $\wedge$ *seq f g*

abbreviation *span*
where *span f g* $\equiv$ *arr f* $\wedge$ *arr g* $\wedge$ *dom* $f =$ *dom g*

abbreviation *cospan*
where *cospan f g* $\equiv$ *arr f* $\wedge$ *arr g* $\wedge$ *cod* $f =$ *cod g*

abbreviation *par*
where *par f g* $\equiv$ *arr f* $\wedge$ *arr g* $\wedge$ *dom* $f =$ *dom g* $\wedge$ *cod* $f =$ *cod g*

end

end

Chapter 3

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 $Hom :: 'o \Rightarrow 'o \Rightarrow 'a\ set$
and $Id :: 'o \Rightarrow 'a$
and $Comp :: 'o \Rightarrow 'o \Rightarrow 'o \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
assumes *Id-in-Hom*: $A \in Obj \Longrightarrow Id\ A \in Hom\ A\ A$
and *Comp-in-Hom*: $\llbracket\ A \in Obj;\ B \in Obj;\ C \in Obj;\ f \in Hom\ A\ B;\ g \in Hom\ B\ C\ \rrbracket$
$\Longrightarrow Comp\ C\ B\ A\ g\ f \in Hom\ A\ C$
and *Comp-Hom-Id*: $\llbracket\ A \in Obj;\ f \in Hom\ A\ B\ \rrbracket \Longrightarrow Comp\ B\ A\ A\ f\ (Id\ A) = f$
and *Comp-Id-Hom*: $\llbracket\ B \in Obj;\ f \in Hom\ A\ B\ \rrbracket \Longrightarrow Comp\ B\ B\ A\ (Id\ B)\ f = f$
and *Comp-assoc*: $\llbracket\ A \in Obj;\ B \in Obj;\ C \in Obj;\ D \in Obj;$
$f \in Hom\ A\ B;\ g \in Hom\ B\ C;\ h \in Hom\ C\ D\ \rrbracket \Longrightarrow$

```
                 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-magma 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
    thus ?thesis by simp
  qed

  lemma ide-char:
  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 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 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 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*:
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 $\llbracket f \neq \mathit{Null}; \mathit{Dom}\ f \in \mathit{Obj}; \mathit{Cod}\ f \in \mathit{Obj}; \mathit{Map}\ f \in \mathit{Hom}\ (\mathit{Dom}\ f)\ (\mathit{Cod}\ f) \rrbracket \Longrightarrow T$
shows T
using *assms arr-char* **by** *simp*

lemma *arr-MkArr* [*simp*]:
assumes $A \in \mathit{Obj}$ **and** $B \in \mathit{Obj}$ **and** $f \in \mathit{Hom}\ A\ B$
shows *arr* (*MkArr A B f*)
using *assms arr-char* **by** *simp*

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

lemma *Arr-comp*:
assumes *arr f* **and** *arr g* **and** $\mathit{Dom}\ g = \mathit{Cod}\ f$
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** $\mathit{Dom}\ g = \mathit{Cod}\ f$
shows $\mathit{Dom}\ (\mathit{COMP}\ g\ f) = \mathit{Dom}\ f$
unfolding *COMP-def*
using *assms arr-char* **by** *simp*

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

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

lemma *seq-char*:
shows $\mathit{seq}\ g\ f \longleftrightarrow \mathit{arr}\ f \wedge \mathit{arr}\ g \wedge \mathit{Dom}\ g = \mathit{Cod}\ f$
using *arr-char not-arr-null null-char COMP-def Arr-comp* **by** *metis*

interpretation *category COMP*
proof
show $\bigwedge g\ f.\ \mathit{COMP}\ g\ f \neq \mathit{null} \Longrightarrow \mathit{seq}\ g\ f$

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* $\in$ *Obj*
using *assms arr-char* **by** *simp*

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

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

lemma *MkArr-in-hom*:
assumes $A \in Obj$ **and** $B \in Obj$ **and** $f \in Hom\ A\ B$
shows *in-hom* (*MkArr A B f*) (*MkIde A*) (*MkIde 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* $\in$ *Obj* $\rightarrow$ *Collect ide*
and *Dom* $\in$ *Collect ide* $\rightarrow$ *Obj*
and *A* $\in$ *Obj* $\Longrightarrow$ *Dom* (*MkIde A*) = *A*
and *a* $\in$ *Collect ide* $\Longrightarrow$ *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 4

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.

4.1 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* $::$ $'arr \Rightarrow 'obj$
and *Cod* $::$ $'arr \Rightarrow 'obj$
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
```

4.2 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 $x \in Arr$
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

4.3 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*)
   apply (*metis empty-iff singletonD*)
  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 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 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

4.4 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 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 $\bigwedge p.\ length\ p = 1 \longleftrightarrow (\exists x.\ p = [x])$
by (*auto simp*: *length-Suc-conv*)
ultimately have $\exists x.\ x \in Arr \wedge Path\ f = [x]$
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 $f \in (\lambda x.\ MkArr\ False\ True\ [x])$ ‘ Arr
using *1* **by** *auto*
qed
thus $f = MkIde\ False \vee f = MkIde\ True \vee f \in (\lambda x.\ MkArr\ False\ True\ [x])$ ‘ Arr
by *auto*
qed

lemma *arr-char*:
shows $arr\ f \longleftrightarrow f = Zero \vee f = One \vee f \in fromArr$ ‘ 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 $\neg\ arr\ f \Longrightarrow ?thesis$
using *dom-char* **by** *simp*
moreover have $arr\ f \Longrightarrow ?thesis$
proof −
assume *f*: *arr f*
have *1*: *dom f* = *MkIde* (*Dom f*)
using *f dom-char* **by** *simp*
have $f = One \Longrightarrow ?thesis$
using *f 1 One-def* **by** (*metis* (*full-types*) *Dom.simps*(*1*))
moreover have $f = Zero \Longrightarrow ?thesis$
using *f 1 Zero-def* **by** (*metis* (*full-types*) *Dom.simps*(*1*))
moreover have $f \in fromArr$ ‘ $Arr \Longrightarrow ?thesis$
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*) ∨

$(a = One \wedge b = One \wedge f = One) \vee$
$(a = Zero \wedge b = One \wedge f \in fromArr \text{ ` } Arr)$

proof −
have *f = Zero* ⟹ *?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* ⟹ *?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* ∈ *fromArr ‘ Arr* ⟹ *?thesis*
proof −
assume *f*: *f* ∈ *fromArr ‘ Arr*
have *1*: *arr f* **using** *f arr-char* **by** *simp*
moreover have *dom f = Zero* ∧ *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-hom f a b* ⟷ $(a = Zero \wedge b = Zero \wedge f = Zero \vee$
$a = One \wedge b = One \wedge f = One \vee$
$a = Zero \wedge b = One \wedge f \in fromArr \text{ ` } Arr)$
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* ≠ *One*
by (*simp add*: *One-def Zero-def*)

lemma *Zero-not-eq-fromArr* [*simp*]:
shows *Zero* ∉ *fromArr ‘ Arr*
using *ide-char not-ide-fromArr*
by (*metis* (*no-types*, *lifting*) *image-iff*)

lemma *One-not-eq-fromArr* [*simp*]:
shows *One* ∉ *fromArr ‘ Arr*
using *ide-char not-ide-fromArr*
by (*metis* (*no-types*, *lifting*) *image-iff*)

lemma *comp-char*:
shows *g · 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* ⟹ *f = Zero* ⟹ *g · 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* $\Longrightarrow$ *g* = *One* $\Longrightarrow$ *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* $\Longrightarrow$ *f* ≠ *Zero* $\Longrightarrow$ *g* ≠ *One* $\Longrightarrow$ *g* · *f* = *null*
using *seq-char Zero-def One-def* **by** *simp*
moreover have ¬*seq g f* $\Longrightarrow$ *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* $\Longrightarrow$ *g* · *f* = *g*
and *g* = *One* $\Longrightarrow$ *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* $\Longrightarrow$ *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

4.5 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 5

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** $\cdot$ *55*)
 where $y \cdot x \equiv$ (*if* $x \in$ *Obj* $\wedge$ $x = y$ *then* x *else* *Null*)

 interpretation *partial-magma 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* $\subseteq$ *category comp*
using *is-category* **by** *auto*

context *discrete-category*
begin

lemma *arr-char* [*iff*]:
shows *arr f* $\longleftrightarrow$ *f* $\in$ *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* $\in$ *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* $\in$ *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* $\in$ *Obj* $\wedge$ *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*

end

end

Chapter 6

DualCategory

theory *DualCategory*
imports *Category*
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** $\cdot$ *55*)
begin

  definition *comp* (**infixr** $\cdot^{op}$ *55*)
  where $g \cdot^{op} f \equiv f \cdot g$

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

  interpretation *partial-magma 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.comp-null*(*2*) *comp-null*(*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*

end

end

Chapter 7

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 $epi\ f = (arr\ f \wedge inj\text{-}on\ (\lambda g.\ g \cdot f)\ \{g.\ seq\ g\ f\})$

definition *mono*
where $mono\ f = (arr\ f \wedge inj\text{-}on\ (\lambda g.\ f \cdot g)\ \{g.\ seq\ f\ g\})$

lemma *epiI* [*intro*]:
assumes $arr\ f$ **and** $\bigwedge g\ g'.\ seq\ g\ f \wedge seq\ g'\ f \wedge g \cdot f = g' \cdot f \Longrightarrow g = g'$
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 *epiE* [*elim*]:
assumes $epi\ f$
and $seq\ g\ f$ **and** $seq\ g'\ f$ **and** $g \cdot f = g' \cdot f$
shows $g = g'$
  using *assms* **unfolding** *epi-def inj-on-def* **by** *blast*

lemma *monoI* [*intro*]:
assumes $arr\ g$ **and** $\bigwedge f\ f'.\ seq\ g\ f \wedge seq\ g\ f' \wedge g \cdot f = g \cdot f' \Longrightarrow f = f'$
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 *monoE* [*elim*]:
assumes *mono g*
and *seq g f* **and** *seq g f'* **and** $g \cdot f = g \cdot f'$
shows $f' = f$
  using *assms* **unfolding** *mono-def inj-on-def* **by** *blast*

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* $f \equiv \exists\, g.\ ide\ (g \cdot f)$

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

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

definition *retraction*
where *retraction* $g \equiv \exists f.\ ide\ (g \cdot f)$

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

lemma *retractionE* [*elim*]:
assumes *retraction* g
obtains f **where** *ide* $(g \cdot f)$
  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* $(h \cdot g)$ **by** *blast*
  have *hg*: *seq* $h\ g$ **using** h **by** *auto*
  fix $f\, f'$
  assume *seq* $g\ f \wedge$ *seq* $g\ f' \wedge g \cdot f = g \cdot f'$
  thus $f = f'$
    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* $(g \cdot f)$ **by** *blast*
  have *gf*: *seq* $g\ f$ **using** f **by** *auto*
  fix $h\ h'$
  assume *seq* $h\ g \wedge$ *seq* $h'\ g \wedge h \cdot g = h' \cdot g$
  thus $h = h'$
    using *gf f ide-compE seqE comp-assoc comp-arr-dom* **by** *metis*

qed

lemma *section-retraction-compose*:
assumes *ide* $(e \cdot m)$ **and** *ide* $(e' \cdot m')$ **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 monoE 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 epiE 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* $\cdot$ *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 monoE 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* $\wedge$ *epi f*
from *f* **obtain** *g* **where** *g*: *ide* $(g \cdot f)$ **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 epiE ide-char'
by *metis*
thus *iso f* **by** *auto*
qed
qed

lemma *iso-iff-section-and-retraction*:
shows *iso f* $\longleftrightarrow$ *section f* $\wedge$ *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* $(f' \cdot f)$
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* $(f' \cdot f)$ $(g \cdot g')$
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

definition *isomorphic*
where *isomorphic a a'* = $(\exists f.\ \ll f : a \to a' \gg \wedge\ iso\ f)$

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** $\ll f : a \to a' \gg \wedge\ iso\ f$
using *assms isomorphic-def* **by** *meson*

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*:
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* · *f*) = *inv f* · *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 iso-inv-iso 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* · *f*) ⟹ *inverse-arrows f g*
and *ide* (*f* · *g*) ⟹ *inverse-arrows f g*
proof −
  show *ide* (*g* · *f*) ⟹ *inverse-arrows f g*
    using *assms*
    by (*metis comp-inv-arr′ epiE ide-compE inv-is-inverse iso-iff-section-and-epi*)
  show *ide* (*f* · *g*) ⟹ *inverse-arrows f g*
    using *assms*
    by (*metis ide-compE comp-arr-inv′ inv-is-inverse iso-iff-mono-and-retraction monoE*)

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*)

end

end

Chapter 8

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 $initial\ a \equiv ide\ a \wedge (\forall\, b.\ ide\ b \longrightarrow (\exists!f.\ \ll f : a \rightarrow b\gg))$

definition *terminal*
where $terminal\ b \equiv ide\ b \wedge (\forall\, a.\ ide\ a \longrightarrow (\exists!f.\ \ll f : a \rightarrow b\gg))$

abbreviation *initial-arr*
where $initial\text{-}arr\ f \equiv arr\ f \wedge initial\ (dom\ f)$

abbreviation *terminal-arr*
where $terminal\text{-}arr\ f \equiv arr\ f \wedge terminal\ (cod\ f)$

abbreviation *point*
where $point\ f \equiv arr\ f \wedge terminal\ (dom\ f)$

lemma *initial-arr-unique*:
assumes $par\ f\ f'$ **and** $initial\text{-}arr\ f$ **and** $initial\text{-}arr\ f'$
shows $f = f'$
using *assms in-homI initial-def ide-cod* **by** *blast*

lemma *initialI* [*intro*]:
assumes $ide\ a$ **and** $\bigwedge b.\ ide\ b \Longrightarrow \exists!f.\ \ll f : a \rightarrow b\gg$
shows *initial a*
using *assms initial-def* **by** *auto*

lemma *initialE* [*elim*]:
assumes *initial a* **and** *ide b*
obtains *f* **where** $\ll f : a \rightarrow b\gg$ **and** $\bigwedge f'. \ll f' : a \rightarrow b\gg \Longrightarrow f' = f$
  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** $\bigwedge a.\ ide\ a \Longrightarrow \exists!f. \ll f : a \rightarrow b\gg$
shows *terminal b*
  using *assms terminal-def* **by** *auto*

lemma *terminalE* [*elim*]:
assumes *terminal b* **and** *ide a*
obtains *f* **where** $\ll f : a \rightarrow b\gg$ **and** $\bigwedge f'. \ll f' : a \rightarrow b\gg \Longrightarrow f' = f$
  using *assms terminal-def terminal-arr-unique* **by** *meson*

theorem *terminal-objs-isomorphic*:
assumes *terminal a* **and** *terminal b*
shows *isomorphic a b*
proof –
  from *assms* **obtain** *f* **where** *f*: $\ll f : a \rightarrow b\gg$
    using *terminal-def* **by** *meson*
  from *assms* **obtain** *g* **where** *g*: $\ll g : b \rightarrow a\gg$
    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

theorem *initial-objs-isomorphic*:
assumes *initial a* **and** *initial b*
shows *isomorphic a b*
proof –
  from *assms* **obtain** *f* **where** *f*: $\ll f : a \rightarrow b\gg$ **using** *initial-def* **by** *auto*
  from *assms* **obtain** *g* **where** *g*: $\ll g : b \rightarrow a\gg$ **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 *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 9

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** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and F :: $'a \Rightarrow 'b$ +
assumes *is-extensional*: $\neg A.arr\ f \Longrightarrow F\ f = B.null$
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* ($\ll$- : - $\to_A$ -$\gg$)
  notation *B.in-hom* ($\ll$- : - $\to_B$ -$\gg$)

  lemma *preserves-hom* [*intro*]:
  assumes $\ll f : a \to_A b\gg$
  shows $\ll F\ f : F\ a \to_B F\ b\gg$
    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 is-extensional 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-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

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

end

locale *embedding-functor* = *functor A B F*
for A :: *′a comp*
and B :: *′b comp*
and F :: $'a \Rightarrow 'b$ +
assumes *is-embedding*: $[\![$ *A.arr f*; *A.arr f′*; $F\,f = F\,f'$ $]\!] \Longrightarrow f = f'$

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 :: $'a \Rightarrow 'b$

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*

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

end

sublocale *identity-functor* ⊆ *functor C C map*
using *is-functor* **by** *auto*

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* ⊆ *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.is-extensional G.is-extensional 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

end

sublocale composite-functor ⊆ functor A C G o F
  using functor-comp F.functor-axioms G.functor-axioms by blast

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 ⋀x. (F o A.map) x = F x
    using A.map-def is-extensional 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 ⋀x. (B.map o F) x = F x
    using B.map-def by (metis comp-apply is-extensional preserves-arr)
qed

locale inverse-functors =
  A: category A +
  B: category B +
  F: functor A B F +
  G: functor B A G
```

for A :: $'a$ *comp* (**infixr** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and F :: $'a \Rightarrow 'b$
and G :: $'b \Rightarrow 'a$ +
assumes *inv*: G *o* F = *identity-functor.map* A
and *inv'*: F *o* G = *identity-functor.map* B

locale *isomorphic-categories* =
  A: *category* A +
  B: *category* B
for A :: $'a$ *comp* (**infixr** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*) +
assumes *iso*: $\exists F\ G.$ *inverse-functors* $A\ B\ F\ G$

sublocale *inverse-functors* $\subseteq$ *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.is-extensional FG'.G.is-extensional 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 +
  F: *functor* $A\ B\ F$
for A :: $'a$ *comp* (**infixr** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)

and F :: $'a \Rightarrow 'b$ +
assumes *invertible*: $\exists\, G.$ *inverse-functors A B F G*
begin

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

definition *inv*
where *inv* $\equiv$ *THE G. inverse-functors A B F G*

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

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

lemma *preserves-terminal*:
assumes *A.terminal a*
shows *B.terminal* (*F a*)
proof
show *0*: *B.ide* (*F a*) **using** *assms F.preserves-ide A.terminal-def* **by** *blast*
fix b :: $'b$
assume *b*: *B.ide b*
show $\exists!g.$ $\ll g : b \rightarrow_B F\ a\gg$
proof
let *?G* = *SOME G. inverse-functors A B F G*
from *invertible* **have** *G*: *inverse-functors A B F ?G*
using *someI-ex* [*of* $\lambda G.$ *inverse-functors A B F G*] **by** *fast*
interpret *inverse-functors A B F ?G* **using** *G* **by** *auto*
let *?P* = $\lambda f.$ $\ll f : ?G\ b \rightarrow_A a\gg$
have *1*: $\exists!f.$ *?P f* **using** *assms b A.terminal-def G.preserves-ide* **by** *simp*
hence *2*: *?P* (*THE f. ?P f*) **by** (*metis* (*no-types*, *lifting*) *theI'*)
thus $\ll F$ (*THE f. ?P f*) : $b \rightarrow_B F\ a\gg$
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*: $\ll$(*THE f. ?P f*) : $?G\ b \rightarrow_A a\gg$
using *assms 2 b G* **by** *simp*
fix g :: $'b$
assume *g*: $\ll g : b \rightarrow_B F\ a\gg$
have *?G* (*F a*) = *a*
using *assms*(*1*) *A.terminal-def inv A.map-simp*
by (*metis 0 F.preserves-reflects-arr B.ideD*(*1*) *comp-apply*)
hence $\ll ?G\ g : ?G\ b \rightarrow_A a\gg$
using *assms*(*1*) *g A.terminal-def inv G.preserves-hom* [*of b F a g*]
by (*elim B.in-homE*, *auto*)
hence *?G g* = (*THE f. ?P f*) **using** *assms 1 3 A.terminal-def* **by** *blast*
thus *g* = *F* (*THE f. ?P f*)

```
      using inv′ g by (metis B.in-homE B.map-simp comp-def)
    qed
  qed

end

sublocale invertible-functor ⊆ inverse-functors A B F inv
  using inv-is-inverse by simp
```

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 ‹λC B A 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 arr f then CC.MkArr (dom f) (cod f) f else CC.null›
                  ‹λF. if CC.arr F then CC.Map F else 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.is-extensional
      by auto
  qed

  interpretation isomorphic-categories C CC.COMP ..

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

```
  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)

    definition map
    where map ≡ F

    lemma map-simp [simp]:
    shows map f = F f
      by (simp add: map-def)

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

  end

  sublocale invertible-functor ⊆ inverse-functors A B F inv
    using inv-is-inverse by simp

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

end
```

Chapter 10

SetCategory

theory *SetCategory*
imports *Category Functor*
begin

This theory defines a locale *set-category* that axiomatizes the notion "category of all $'a$-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 $\mathit{Fun}\ f \in \mathit{set}\ (\mathit{dom}\ f) \rightarrow \mathit{set}\ (\mathit{cod}\ f)$, 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.

10.1 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*

10.2 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.

locale *set-category-data* = *category* S
for S :: *'s comp*   (**infixr** · *55*)
and *img* :: *'s* ⇒ *'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*

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 A* ⊆ *Univ then inv-into* (*Collect ide*) *set A else null*)

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, that there is an object corresponding to each subset of the universe, and for any objects a and b and function F ∈ *hom unity a* → *hom unity b* there is an arrow f ∈ *hom a b* whose action under the composition of S coincides with the function F.

locale *set-category-given-img* = *set-category-data S img*
for *S* :: *'s comp* (**infixr** · *55*)
and *img* :: *'s* ⇒ *'s* +
assumes *nonempty-Univ*: *Univ* ≠ {}
and *img-mapsto*: *ide a* ⟹ *img* ∈ *hom unity a* → *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*: *A* ⊆ *Univ* ⟹ ∃ *a*. *ide a* ∧ *set a* = *A*
and *fun-complete1*: ⟦ *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

Each arrow f ∈ *hom a b* determines a function *Fun f* ∈ *Univ* → *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*:
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*:
assumes *par f f'* **and** *Fun f* = *Fun f'*
shows *f* = *f'*

using *assms comp-arr-point extensional-arr* **by** *metis*

lemma *terminal-unity*:
shows *terminal unity*
using *unity-def nonempty-Univ* **by** (*simp add*: *someI-ex*)

lemma *ide-unity* [*simp*]:
shows *ide unity*
using *terminal-unity terminal-def* **by** *blast*

lemma *set-subset-Univ* [*simp*]:
assumes *ide a*
shows *set a* $\subseteq$ *Univ*
using *assms set-def img-mapsto* **by** *auto*

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

The mapping *mkIde*, which takes subsets of the universe to identities, and *set*, which takes identities to subsets of the universe, are inverses.

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

lemma *set-mkIde* [*simp*]:
assumes *A* $\subseteq$ *Univ*
shows *set* (*mkIde A*) = *A*
using *assms mkIde-def set-complete someI-ex* [*of* $\lambda a.$ *a* $\in$ *Collect ide* $\wedge$ *set a* = *A*]
by (*simp add*: *inv-into-def*)

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

lemma *arr-mkIde* [*iff*]:
shows *arr* (*mkIde A*) $\longleftrightarrow$ *A* $\subseteq$ *Univ*
using *not-arr-null mkIde-def ide-mkIde* **by** *auto*

lemma *dom-mkIde* [*simp*]:
assumes *A* $\subseteq$ *Univ*
shows *dom* (*mkIde A*) = *mkIde A*
using *assms ide-mkIde* **by** *simp*

lemma *cod-mkIde* [*simp*]:
assumes *A* $\subseteq$ *Univ*
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* $\in$ *extensional* (*set* (*dom f*)) $\cap$ (*set* (*dom f*) $\rightarrow$ *set* (*cod f*))
proof
  show *Fun f* $\in$ *extensional* (*set* (*dom f*)) **using** *Fun-def* **by** *fastforce*
  show *Fun f* $\in$ *set* (*dom f*) $\rightarrow$ *set* (*cod f*)
  proof
    fix *t*
    assume *t*: *t* $\in$ *set* (*dom f*)
    have *Fun f t* = *img* (*f* $\cdot$ *inv-into* (*hom unity* (*dom f*)) *img t*)
      using *assms t Fun-def comp-def* **by** *simp*
    moreover have ... $\in$ *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* $\in$ *set* (*cod f*) **by** *auto*
  qed
qed

Identities of S correspond to restrictions of the identity function.

lemma *Fun-ide* [*simp*]:
assumes *ide a*
shows *Fun a* = *restrict* ($\lambda x.\ x$) (*set a*)
  using *assms Fun-def inj-img set-def comp-cod-arr* **by** *fastforce*

lemma *Fun-mkIde* [*simp*]:
assumes *A* $\subseteq$ *Univ*
shows *Fun* (*mkIde A*) = *restrict* ($\lambda x.\ x$) *A*
  using *assms* **by** *simp*

Composition in ($\cdot$) corresponds to extensional function composition.

lemma *Fun-comp* [*simp*]:
assumes *seq g f*
shows *Fun* (*g* $\cdot$ *f*) = *restrict* (*Fun g o Fun f*) (*set* (*dom f*))
proof −
  have *restrict* (*img o S* (*g* $\cdot$ *f*) *o* (*inv-into* (*hom unity* (*dom* (*g* $\cdot$ *f*))) *img*))
              (*set* (*dom* (*g* $\cdot$ *f*)))
        = *restrict* (*Fun g o Fun f*) (*set* (*dom f*))
  proof −
    have *1*: *set* (*dom* (*g* $\cdot$ *f*)) = *set* (*dom f*)
      using *assms* **by** *auto*
    let *?img'* = $\lambda a.\ \lambda t.$ *inv-into* (*hom unity a*) *img t*
    have *2*: $\bigwedge t.$ *t* $\in$ *set* (*dom* (*g* $\cdot$ *f*)) $\Longrightarrow$
              (*img o S* (*g* $\cdot$ *f*) *o ?img'* (*dom* (*g* $\cdot$ *f*))) *t* = (*Fun g o Fun f*) *t*
    proof −
      fix *t*
      assume *t* $\in$ *set* (*dom* (*g* $\cdot$ *f*))
      hence *t*: *t* $\in$ *set* (*dom f*) **by** (*simp add*: *1*)

```
      have 3: ⋀a x. x ∈ hom unity a ⟹ ?img' a (img x) = x
        using assms img-mapsto inj-img ide-cod inv-into-f-eq
        by (metis arrI in-homE mem-Collect-eq)
      have 4: ?img' (dom f) t ∈ hom unity (dom f)
        using assms t inv-into-into [of t img hom unity (dom f)] set-def by simp
      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))
        using assms t 3 Fun-def set-def comp-arr-point by auto
      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
    show ?thesis using 1 2 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 A ⊆ Univ ∧ B ⊆ Univ ∧ 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 img-mapsto inj-img set-def by auto
```

```
    qed
    hence ∃f. «f : a → b» ∧ (∀x. «x : unity → a» ⟶ f · x = ?F' x)
      using assms fun-complete1 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))
        proof −
          have ?img' (dom f) t ∈ hom unity (dom f)
            using t set-def inv-into-into by metis
          thus ?thesis using f by auto
        qed
        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
    by (metis (no-types, lifting) in-homE)
 qed
qed

lemma mkArr-in-hom:
assumes A ⊆ Univ and B ⊆ Univ 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]
       theI' [of λf. f ∈ hom (mkIde A) (mkIde B) ∧ Fun f = restrict F A]
 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 $F \notin A \rightarrow B$. 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 $F \in A \rightarrow B$ 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*) $\longleftrightarrow$ *A* $\subseteq$ *Univ* $\wedge$ *B* $\subseteq$ *Univ* $\wedge$ *F* $\in$ *A* $\rightarrow$ *B*
proof
  show *arr* (*mkArr A B F*) $\Longrightarrow$ *A* $\subseteq$ *Univ* $\wedge$ *B* $\subseteq$ *Univ* $\wedge$ *F* $\in$ *A* $\rightarrow$ *B*
    using *mkArr-def not-arr-null ex-un-null someI-ex* [*of* λf. $\neg$*arr f*] **by** *metis*
  show *A* $\subseteq$ *Univ* $\wedge$ *B* $\subseteq$ *Univ* $\wedge$ *F* $\in$ *A* $\rightarrow$ *B* $\Longrightarrow$ *arr* (*mkArr A B F*)
    using *mkArr-in-hom* **by** *auto*
qed

lemma *Fun-mkArr'*:
assumes *arr* (*mkArr A B F*)
shows «*mkArr A B F* : *mkIde A* $\rightarrow$ *mkIde B*»
and *Fun* (*mkArr A B F*) = *restrict F A*
proof –
  have *1*: *A* $\subseteq$ *Univ* $\wedge$ *B* $\subseteq$ *Univ* $\wedge$ *F* $\in$ *A* $\rightarrow$ *B* **using** *assms* **by** *fast*
  have *2*: *mkArr A B F* $\in$ *hom* (*mkIde A*) (*mkIde B*) $\wedge$
        Fun (*mkArr A B F*) = *restrict F* (*set* (*mkIde A*))
  proof –
    have $\exists!f$. *f* $\in$ *hom* (*mkIde A*) (*mkIde B*) $\wedge$ *Fun f* = *restrict F* (*set* (*mkIde A*))
      using *1 fun-complete* [*of mkIde A mkIde B F*] **by** *simp*
    thus *?thesis* **using** *1 mkArr-def theI'* **by** *simp*
  qed
  show «*mkArr A B F* : *mkIde A* $\rightarrow$ *mkIde B*» **using** *1 2* **by** *auto*
  show *Fun* (*mkArr A B F*) = *restrict F A* **using** *1 2* **by** *auto*
qed

lemma *mkArr-Fun* [*simp*]:
assumes *arr f*
shows *mkArr* (*set* (*dom f*)) (*set* (*cod f*)) (*Fun f*) = *f*
proof –
  have *1*: *set* (*dom f*) $\subseteq$ *Univ* $\wedge$ *set* (*cod f*) $\subseteq$ *Univ* $\wedge$ *ide* (*dom f*) $\wedge$ *ide* (*cod f*) $\wedge$
      Fun f $\in$ *extensional* (*set* (*dom f*)) $\cap$ (*set* (*dom f*) $\rightarrow$ *set* (*cod f*))
    using *assms Fun-mapsto* **by** *force*
  hence $\exists!f'$. $f' \in$ *hom* (*dom f*) (*cod f*) $\wedge$ *Fun* f' = *restrict* (*Fun f*) (*set* (*dom f*))
    using *fun-complete* **by** *force*
  moreover have *f* $\in$ *hom* (*dom f*) (*cod f*) $\wedge$ *Fun f* = *restrict* (*Fun f*) (*set* (*dom f*))
    using *assms 1 extensional-restrict* **by** *force*
  ultimately have *f* = (*THE* f'. $f' \in$ *hom* (*dom f*) (*cod f*) $\wedge$
              Fun f' = *restrict* (*Fun f*) (*set* (*dom f*)))

using *theI′* [*of* $\lambda f'.\ f' \in$ *hom* (*dom f*) (*cod f*) $\land$ *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* 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 arr-mkArr Fun-mkArr*
by (*intro arr-eqI*, *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* [*simp*]:
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 $A \subseteq Univ$
shows $mkArr\ A\ A\ (\lambda x.\ x) = mkIde\ A$
  using *assms* **by** (*intro arr-eqI*, *auto*)

lemma *comp-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*) **and** *arr* (*mkArr B C G*)
shows $mkArr\ B\ C\ G \cdot mkArr\ A\ B\ F = mkArr\ A\ C\ (G \circ F)$
proof (*intro arr-eqI*)
  have *1*: *seq* (*mkArr B C G*) (*mkArr A B F*) **using** *assms* **by** *force*
  have *2*: $G\ o\ F \in A \rightarrow C$ **using** *assms* **by** *auto*
  show $par\ (mkArr\ B\ C\ G \cdot mkArr\ A\ B\ F)\ (mkArr\ A\ C\ (G \circ F))$
    using *1 2* **by** *auto*
  show $Fun\ (mkArr\ B\ C\ G \cdot mkArr\ A\ B\ F) = Fun\ (mkArr\ A\ C\ (G \circ F))$
    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 $terminal\ t \longleftrightarrow ide\ t \wedge (\exists!x.\ x \in set\ t)$
proof −
  have $terminal\ t \Longrightarrow ide\ t \wedge (\exists!x.\ x \in set\ t)$
  proof −
    assume *t*: *terminal t*
    have *ide t* **using** *t terminal-def* **by** *auto*
    moreover have $\exists!x.\ x \in set\ t$
    proof −
      have $\exists!x.\ x \in hom\ unity\ t$
        using *t terminal-unity terminal-def* **by** *auto*
      thus *?thesis* **using** *set-def* **by** *auto*
    qed
    ultimately show $ide\ t \wedge (\exists!x.\ x \in set\ t)$ **by** *auto*
  qed
  moreover have $ide\ t \wedge (\exists!x.\ x \in set\ t) \Longrightarrow terminal\ t$
  proof −
    assume *t*: $ide\ t \wedge (\exists!x.\ x \in set\ t)$
    from *this* **obtain** t' **where** $set\ t = \{t'\}$ **by** *blast*
    hence t': $set\ t = \{t'\} \wedge \{t'\} \subseteq Univ \wedge t = mkIde\ \{t'\}$
      using *t set-subset-Univ mkIde-set* **by** *metis*
    show *terminal t*
    proof
      show *ide t* **using** *t* **by** *simp*
      show $\bigwedge a.\ ide\ a \Longrightarrow \exists!f.\ \ll f : a \rightarrow t\gg$
      proof −
        fix *a*
        assume *a*: *ide a*

show $\exists!f.\ \ll f : a \rightarrow t\gg$
proof
show *1*: $\ll mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t') : a \rightarrow t\gg$
using *a t t' mkArr-in-hom*
by (*metis Pi-I' mkIde-set set-subset-Univ singletonD*)
show $\bigwedge f.\ \ll f : a \rightarrow t\gg \Longrightarrow f = mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t')$
proof −
fix *f*
assume *f*: $\ll f : a \rightarrow t\gg$
show $f = mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t')$
proof (*intro arr-eqI*)
show *1*: *par f* $(mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t'))$ **using** *1 f in-homE* **by** *metis*
show $Fun\ f = Fun\ (mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t'))$
proof −
have $Fun\ (mkArr\ (set\ a)\ \{t'\}\ (\lambda x.\ t')) = (\lambda x{\in}set\ a.\ t')$
using *1 Fun-mkArr* **by** *simp*
also have ... $= Fun\ f$
proof −
have $\bigwedge x.\ x \in set\ a \Longrightarrow Fun\ f\ x = t'$
using *f t' Fun-def mkArr-Fun arr-mkArr*
by (*metis PiE in-homE singletonD*)
moreover have $\bigwedge x.\ x \notin set\ a \Longrightarrow Fun\ f\ x = undefined$
using *f Fun-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *force*
qed
qed
qed
qed
qed
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 $terminal\ t \longleftrightarrow ide\ t \wedge set\ t = \{t\}$
proof
assume *t*: *terminal t*
show $ide\ t \wedge set\ t = \{t\}$
proof
show *ide t* **using** *t terminal-char1* **by** *auto*
show $set\ t = \{t\}$
proof −
have $\exists!x.\ x \in hom\ unity\ t$ **using** *t terminal-def terminal-unity* **by** *force*
moreover have $t \in img\ `\ hom\ unity\ t$ **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) +
assumes ex-img: ∃ img. set-category-given-img S img
begin

  notation in-hom («- : - → -»)

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

end

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

context set-category
begin
```

The arbitrary choice of *img* induces a system of inclusions, which are 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 $(\exists f.\ \ll f : a \rightarrow b\gg \wedge\ incl\ f) \longleftrightarrow set\ a \subseteq set\ b$
proof
  show $\exists f.\ \ll f : a \rightarrow b\gg \wedge\ incl\ f \Longrightarrow set\ a \subseteq set\ b$
    using *incl-def* **by** *auto*
  show $set\ a \subseteq set\ b \Longrightarrow \exists f.\ \ll f : a \rightarrow b\gg \wedge\ incl\ f$
  proof
    assume *1*: $set\ a \subseteq set\ b$
    show $\ll mkArr\ (set\ a)\ (set\ b)\ (\lambda x.\ x) : a \rightarrow b\gg \wedge\ incl\ (mkArr\ (set\ a)\ (set\ b)\ (\lambda x.\ x))$
    proof
      show $\ll mkArr\ (set\ a)\ (set\ b)\ (\lambda x.\ x) : a \rightarrow b\gg$
      proof −
        have $(\lambda x.\ x) \in set\ a \rightarrow set\ b$ **using** *1* **by** *auto*
        thus *?thesis*
          using *assms mkArr-in-hom set-subset-Univ in-homI* **by** *auto*
      qed
      thus $incl\ (mkArr\ (set\ a)\ (set\ b)\ (\lambda x.\ x))$
        using *1 incl-def* **by** *force*
    qed
  qed
qed

end

10.3 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 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 +
  S': *set-category* S'
for S :: *'s comp* (**infixr** $\cdot$ *55*)
and S' :: *'t comp* (**infixr** $\cdot'$ *55*)
and φ :: $'s \Rightarrow 't$ +
assumes *bij-φ*: *bij-betw* φ $S.Univ$ $S'.Univ$
begin

  notation *S.in-hom* $(\ll\text{-} : \text{-} \rightarrow \text{-}\gg)$
  notation *S'.in-hom* $(\ll\text{-} : \text{-} \rightarrow'' \text{-}\gg)$

  abbreviation ψ
  where $\psi \equiv inv\text{-}into\ S.Univ\ \varphi$

  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 ψ ' φ ' $A = A$
  using *assms bij-*φ **by** (*simp add*: *bij-betw-def*)

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

The object map Φo of a functor from S to S'.

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

lemma *set-*Φo:
assumes $S.ide\ a$
shows $S'.set\ (\Phi o\ a) = \varphi$ ' $S.set\ a$
proof −
  from *assms* **have** $S.set\ a \subseteq S.Univ$ **by** *simp*
  then show *?thesis*
  using *S'.set-mkIde* Φ*o-def assms bij-*φ *bij-betw-def image-mono mem-Collect-eq restrict-def*
  by (*metis* (*no-types*, *lifting*))
qed

lemma Φ*o-preserves-ide*:
assumes $S.ide\ a$
shows $S'.ide\ (\Phi o\ a)$
  using *assms S'.ide-mkIde S.set-subset-Univ bij-*φ *bij-betw-def image-mono restrict-apply'*
  unfolding Φ*o-def*
  by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

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

fix x
assume x: $x \in \varphi$ ‘ $S.set\ (S.dom\ f)$
have $\psi\ x \in S.set\ (S.dom\ f)$
using *assms x* ψ*-img-*φ*-img* [*of* $S.set\ (S.dom\ f)$] *S.set-subset-Univ* **by** *auto*
hence $S.Fun\ f\ (\psi\ x) \in S.set\ (S.cod\ f)$ **using** *assms S.Fun-mapsto* **by** *auto*
hence $\varphi\ (S.Fun\ f\ (\psi\ x)) \in \varphi$ ‘ $S.set\ (S.cod\ f)$ **by** *simp*
thus $\Phi a\ f\ x \in \varphi$ ‘ $S.set\ (S.cod\ f)$ **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 $\Phi a\ (g \cdot f) = restrict\ (\Phi a\ g\ o\ \Phi a\ f)\ (S'.set\ (\Phi o\ (S.dom\ f)))$
proof −
have $\Phi a\ (g \cdot f) = (\lambda x' \in \varphi$ ‘ $S.set\ (S.dom\ f).\ \varphi\ (S.Fun\ (S\ g\ f)\ (\psi\ x')))$
using *gf* Φa*-def* **by** *auto*
also have ... $= (\lambda x' \in \varphi$ ‘ $S.set\ (S.dom\ f).$
$\varphi\ (restrict\ (S.Fun\ g\ o\ S.Fun\ f)\ (S.set\ (S.dom\ f))\ (\psi\ x')))$
using *gf set-*Φo *S.Fun-comp* **by** *simp*
also have ... $= restrict\ (\Phi a\ g\ o\ \Phi a\ f)\ (S'.set\ (\Phi o\ (S.dom\ f)))$
proof −
have $\bigwedge x'.\ x' \in \varphi$ ‘ $S.set\ (S.dom\ f)$
$\Longrightarrow \varphi\ (restrict\ (S.Fun\ g\ o\ S.Fun\ f)\ (S.set\ (S.dom\ f))\ (\psi\ x')) = \Phi a\ g\ (\Phi a\ f\ x')$
proof −
fix x'
assume X': $x' \in \varphi$ ‘ $S.set\ (S.dom\ f)$
hence *1*: $\psi\ x' \in S.set\ (S.dom\ f)$
using *gf* ψ*-img-*φ*-img* [*of* $S.set\ (S.dom\ f)$] *S.set-subset-Univ S.ide-dom* **by** *blast*
hence $\varphi\ (restrict\ (S.Fun\ g\ o\ S.Fun\ f)\ (S.set\ (S.dom\ f))\ (\psi\ x'))$
$= \varphi\ (S.Fun\ g\ (S.Fun\ f\ (\psi\ x')))$
using *restrict-apply* **by** *auto*
also have ... $= \varphi\ (S.Fun\ g\ (\psi\ (\varphi\ (S.Fun\ f\ (\psi\ x')))))$
proof −
have $S.Fun\ f\ (\psi\ x') \in S.set\ (S.cod\ f)$
using *gf 1 S.Fun-mapsto* **by** *fast*
hence $\psi\ (\varphi\ (S.Fun\ f\ (\psi\ x'))) = S.Fun\ f\ (\psi\ x')$
using *assms bij-*φ *S.set-subset-Univ bij-betw-def inv-into-f-f subsetCE S.ide-cod*
by (*metis S.seqE*)
thus *?thesis* **by** *auto*
qed
also have ... $= \Phi a\ g\ (\Phi a\ f\ x')$
proof −
have $\Phi a\ f\ x' \in \varphi$ ‘ $S.set\ (S.cod\ f)$
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 $\in$ *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)) ($\lambda x'$. 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* ($\lambda x'$. x') (*S'.set* (Φo a))
proof –
have Φa a = ($\lambda x' \in \varphi$ ‘ *S.set a*. φ (*S.Fun a* (ψ x')))
using *assms* Φa*-def restrict-apply* **by** *auto*
also have ... = ($\lambda x' \in$ *S'.set* (Φo a). φ (ψ x'))
proof –
have *S.Fun a* = ($\lambda x \in$ *S.set a*. x) **using** *assms S.Fun-ide* **by** *simp*
moreover have $\bigwedge x'$. $x' \in \varphi$ ‘ *S.set a* $\Longrightarrow$ ψ $x' \in$ *S.set a*
using *assms bij-*φ *S.set-subset-Univ image-iff* **by** (*metis* ψ*-img-*φ*-img*)
ultimately show *?thesis*
using *assms set-*Φo **by** *auto*
qed
also have ... = *restrict* ($\lambda x'$. x') (*S'.set* (Φo a))
using *assms S'.set-subset-Univ* Φo*-preserves-ide* φ*-*ψ

by (*meson restr-eqI subsetCE*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
using *assms* Φ-*def* Φ*o-preserves-ide* S'.*mkArr-restrict-eq*$'$
by (*metis* S'.*arrI* S.*ide-char*)
qed
thus *?thesis*
using *assms* S'.*mkIde-as-mkArr* Φ*o-preserves-ide* Φ-*in-hom* **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*-Φ*o* **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* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod g*))) (Φ*a* (S *g f*))
using Φ-*def assms* **by** *auto*
also have ... = S'.*mkArr* (S'.*set* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod g*)))
(*restrict* (Φ*a g o* Φ*a f*) (S'.*set* (Φ*o* (S.*dom f*))))
using *assms* Φ*a-comp set*-Φ*o* **by** *force*
also have ... = S'.*mkArr* (S'.*set* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod g*))) (Φ*a g o* Φ*a f*)
proof −
have S'.*arr* (S'.*mkArr* (S'.*set* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod g*))) (Φ*a g o* Φ*a f*))
using *assms* Φ*a-mapsto* [*of f*] Φ*a-mapsto* [*of g*] Φ*o-preserves-ide* S'.*arr-mkArr*
by (*elim* S.*seqE*, *auto*)
thus *?thesis*
using *assms* S'.*mkArr-restrict-eq* **by** *auto*
qed
also have ... = S' (S'.*mkArr* (S'.*set* (Φ*o* (S.*dom g*))) (S'.*set* (Φ*o* (S.*cod g*))) (Φ*a g*))
(S'.*mkArr* (S'.*set* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod f*))) (Φ*a f*))
proof −
have S'.*arr* (S'.*mkArr* (S'.*set* (Φ*o* (S.*dom f*))) (S'.*set* (Φ*o* (S.*cod f*))) (Φ*a f*))
using *assms* Φ*a-mapsto set*-Φ*o* S.*ide-dom* S.*ide-cod* Φ*o-preserves-ide*
S'.*arr-mkArr* S'.*set-subset-Univ* S.*seqE*
by *metis*
moreover have S'.*arr* (S'.*mkArr* (S'.*set* (Φ*o* (S.*dom g*))) (S'.*set* (Φ*o* (S.*cod g*)))
(Φ*a g*))
using *assms* Φ*a-mapsto set*-Φ*o* S.*ide-dom* S.*ide-cod* Φ*o-preserves-ide* S'.*arr-mkArr*
S'.*set-subset-Univ* S.*seqE*
by *metis*
ultimately show *?thesis* **using** *assms* S'.*comp-mkArr* **by** *force*
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 $S'.Fun\ (\Phi\ f)\ (\varphi\ x) = \Phi a\ f\ (\varphi\ x)$
  using *assms* Φ-*def* Φ.*preserves-arr set*-Φ*o* **by** *auto*

lemma Φ-*acts-elementwise*:
assumes *S.ide a*
shows $S'.set\ (\Phi\ a) = \Phi\ `\ S.set\ a$
proof
  have *0*: $S'.set\ (\Phi\ a) = \varphi\ `\ S.set\ a$
    using *assms* Φ-*ide set*-Φ*o* **by** *simp*
  have *1*: $\bigwedge x.\ x \in S.set\ a \Longrightarrow \Phi\ x = \varphi\ x$
  proof −
    fix *x*
    assume *x*: $x \in S.set\ a$
    have *1*: *S.terminal x* **using** *assms x S.set-subset-Univ* **by** *blast*
    hence *2*: $S'.terminal\ (\varphi\ x)$
      by (*metis CollectD CollectI bij*-φ *bij-betw-def image-iff*)
    have $\Phi\ x = \Phi o\ x$
      using *assms x 1* Φ-*ide S.terminal-def* **by** *auto*
    also have ... $= \varphi\ x$
    proof −
      have $\Phi o\ x = S'.mkIde\ (\varphi\ `\ S.set\ x)$
        using *assms 1 x* Φ*o-def S.terminal-def* **by** *auto*
      moreover have $S'.mkIde\ (\varphi\ `\ S.set\ x) = \varphi\ x$
        using *assms x 1 2 S.terminal-char2 S′.terminal-char2 S′.mkIde-set bij*-φ
        by (*metis image-empty image-insert*)
      ultimately show *?thesis* **by** *auto*
    qed
    finally show $\Phi\ x = \varphi\ x$ **by** *auto*
  qed
  show $S'.set\ (\Phi\ a) \subseteq \Phi\ `\ S.set\ a$ **using** *0 1* **by** *force*
  show $\Phi\ `\ S.set\ a \subseteq S'.set\ (\Phi\ a)$ **using** *0 1* **by** *force*
qed

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

shows $S'.incl\ (\Phi\ m)$
proof $-$
have *1*: $S.arr\ m \wedge S.set\ (S.dom\ m) \subseteq S.set\ (S.cod\ m) \wedge$
$m = S.mkArr\ (S.set\ (S.dom\ m))\ (S.set\ (S.cod\ m))\ (\lambda x.\ x)$
using *assms* $S.incl\text{-}def$ **by** *blast*
have $S'.arr\ (\Phi\ m)$ **using** *1* **by** *auto*
moreover have *2*: $S'.set\ (S'.dom\ (\Phi\ m)) \subseteq S'.set\ (S'.cod\ (\Phi\ m))$
using *1* $\Phi.preserves\text{-}dom$ $\Phi.preserves\text{-}cod$ $\Phi\text{-}acts\text{-}elementwise$
by (*metis* (*full-types*) $S.ide\text{-}cod$ $S.ide\text{-}dom$ *image-mono*)
moreover have $\Phi\ m =$
$S'.mkArr\ (S'.set\ (S'.dom\ (\Phi\ m)))\ (S'.set\ (S'.cod\ (\Phi\ m)))\ (\lambda x'.\ x')$
proof $-$
have $\Phi\ m = S'.mkArr\ (S'.set\ (\Phi o\ (S.dom\ m)))\ (S'.set\ (\Phi o\ (S.cod\ m)))\ (\Phi a\ m)$
using *1* $\Phi\text{-}def$ **by** *simp*
also have ... $= S'.mkArr\ (S'.set\ (S'.dom\ (\Phi\ m)))\ (S'.set\ (S'.cod\ (\Phi\ m)))\ (\Phi a\ m)$
using *1* $\Phi\text{-}ide$ **by** *auto*
finally have *3*: $\Phi\ m =$
$S'.mkArr\ (S'.set\ (S'.dom\ (\Phi\ m)))\ (S'.set\ (S'.cod\ (\Phi\ m)))\ (\Phi a\ m)$
by *auto*
also have ... $= S'.mkArr\ (S'.set\ (S'.dom\ (\Phi\ m)))\ (S'.set\ (S'.cod\ (\Phi\ m)))\ (\lambda x'.\ x')$
proof $-$
have *4*: $S.Fun\ m = restrict\ (\lambda x.\ x)\ (S.set\ (S.dom\ m))$
using *assms* $S.incl\text{-}def$ **by** (*metis* (*full-types*) $S.Fun\text{-}mkArr$)
hence $\Phi a\ m = restrict\ (\lambda x'.\ x')\ (\varphi\ `\ (S.set\ (S.dom\ m)))$
proof $-$
have *5*: $\bigwedge x'.\ x' \in \varphi\ `\ S.set\ (S.dom\ m) \Longrightarrow \varphi\ (\psi\ x') = x'$
using *1* $bij\text{-}\varphi$ *bij-betw-def* $S'.set\text{-}subset\text{-}Univ$ $S.ide\text{-}dom$ $\Phi o\text{-}preserves\text{-}ide$
f-inv-into-f $set\text{-}\Phi o$
by (*metis subsetCE*)
have $\Phi a\ m = restrict\ (\lambda x'.\ \varphi\ (S.Fun\ m\ (\psi\ x')))\ (\varphi\ `\ S.set\ (S.dom\ m))$
using $\Phi a\text{-}def$ **by** *simp*
also have ... $= restrict\ (\lambda x'.\ x')\ (\varphi\ `\ S.set\ (S.dom\ m))$
proof $-$
have $\bigwedge x.\ x \in \varphi\ `\ (S.set\ (S.dom\ m)) \Longrightarrow \varphi\ (S.Fun\ m\ (\psi\ x)) = x$
proof $-$
fix x
assume x: $x \in \varphi\ `\ (S.set\ (S.dom\ m))$
hence $\psi\ x \in S.set\ (S.dom\ m)$
using *1* $S.ide\text{-}dom$ $S.set\text{-}subset\text{-}Univ$ $\psi\text{-}img\text{-}\varphi\text{-}img$ *image-eqI* **by** *metis*
thus $\varphi\ (S.Fun\ m\ (\psi\ x)) = x$ **using** *1 4 5 x* **by** *simp*
qed
thus *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed
hence $\Phi a\ m = restrict\ (\lambda x'.\ x')\ (S'.set\ (S'.dom\ (\Phi\ m)))$
using *1* $set\text{-}dom\text{-}\Phi$ **by** *auto*
thus *?thesis*
using *2 3* $\langle S'.arr\ (\Phi\ m)\rangle$ $S'.mkArr\text{-}restrict\text{-}eq$ $S'.ide\text{-}cod$ $S'.ide\text{-}dom$ $S'.incl\text{-}def$

    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

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' S ψ
 apply *unfold-locales* **by** (*simp add*: *bij-*φ *bij-betw-inv-into*)

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 $S.Fun\ (\Psi\ f')\ (\psi\ x') = \Psi a\ f'\ (\psi\ x')$
 using *assms INV.Fun-*Φ **by** *blast*

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

lemma $\Phi\Psi$:
assumes $S.arr\ f$
shows $\Psi\ (\Phi\ f) = f$
proof (*intro S.arr-eqI*)
 show *par*: $S.par\ (\Psi\ (\Phi\ f))\ f$
  using *assms* Φo*-preserves-ide* Ψo-Φo **by** *auto*
 show $S.Fun\ (\Psi\ (\Phi\ f)) = S.Fun\ f$
 proof −
  have $S.arr\ (\Psi\ (\Phi\ f))$ **using** *assms* **by** *auto*
  moreover have $\Psi\ (\Phi\ f) = S.mkArr\ (S.set\ (S.dom\ f))\ (S.set\ (S.cod\ f))\ (\Psi a\ (\Phi\ f))$
   using *assms INV*.Φ*-def* Φ*-in-hom* Ψo-Φo **by** *auto*
  moreover have $\Psi a\ (\Phi\ f) = (\lambda x \in S.set\ (S.dom\ f).\ \psi\ (S'.Fun\ (\Phi\ f)\ (\varphi\ x)))$
  proof −
   have $\Psi a\ (\Phi\ f) = (\lambda x \in \psi\ `\ S'.set\ (S'.dom\ (\Phi\ f)).\ \psi\ (S'.Fun\ (\Phi\ f)\ (\varphi\ x)))$
   proof −

have $\bigwedge x.\ x \in \psi$ ‘ $S'.set$ $(S'.dom$ $(\Phi\ f)) \Longrightarrow INV.\psi\ x = \varphi\ x$
  using *assms S.ide-dom S.set-subset-Univ* Ψ*.preserves-reflects-arr par bij-*φ
    inv-into-inv-into-eq subsetCE INV.set-dom-Φ
  by *metis*
thus *?thesis*
  using *INV.*Φ*a-def* **by** *auto*
qed
moreover have ψ ‘ $S'.set$ $(S'.dom$ $(\Phi\ f)) = S.set$ $(S.dom\ f)$
  using *assms* **by** (*metis par* Ψ*.preserves-reflects-arr INV.set-dom-*Φ)
ultimately show *?thesis* **by** *auto*
qed
ultimately have *1*: $S.Fun$ $(\Psi$ $(\Phi\ f)) = (\lambda x \in S.set$ $(S.dom\ f).\ \psi$ $(S'.Fun$ $(\Phi\ f)$ $(\varphi\ x)))$
  using *S'.Fun-mkArr* **by** *simp*
show *?thesis*
proof
  fix x
  have $x \notin S.set$ $(S.dom\ f) \Longrightarrow S.Fun$ $(\Psi$ $(\Phi\ f))$ $x = S.Fun\ f\ x$
    using *1 assms extensional-def S.Fun-mapsto S.Fun-def* **by** *auto*
  moreover have $x \in S.set$ $(S.dom\ f) \Longrightarrow S.Fun$ $(\Psi$ $(\Phi\ f))$ $x = S.Fun\ f\ x$
  proof –
    assume x: $x \in S.set$ $(S.dom\ f)$
    have $S.Fun$ $(\Psi$ $(\Phi\ f))$ $x = \psi$ $(\varphi$ $(S.Fun\ f$ $(\psi$ $(\varphi\ x))))$
      using *assms x 1 Fun-*Φ *bij-*φ Φ*a-def* **by** *auto*
    also have ... $= S.Fun\ f\ x$
    proof –
      have *2*: $\bigwedge x.\ x \in S.Univ \Longrightarrow \psi$ $(\varphi\ x) = x$
        using *bij-*φ *bij-betw-inv-into-left* **by** *fast*
      have $S.Fun\ f$ $(\psi$ $(\varphi\ x)) = S.Fun\ f\ x$
        using *assms x 2*
        by (*metis S.ide-dom S.set-subset-Univ subsetCE*)
      moreover have $S.Fun\ f\ x \in S.Univ$
        using *x assms S.Fun-mapsto S.set-subset-Univ S.ide-cod* **by** *blast*
      ultimately show *?thesis* **using** *2* **by** *auto*
    qed
    finally show *?thesis* **by** *auto*
  qed
  ultimately show $S.Fun$ $(\Psi$ $(\Phi\ f))$ $x = S.Fun\ f\ x$ **by** *auto*
qed
qed
qed

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

lemma $\Psi\Phi$:
assumes $S'.arr\ f'$

```
shows Φ (Ψ f′) = f′
proof (intro S′.arr-eqI)
  show par: S′.par (Φ (Ψ f′)) f′
    using assms Φ.preserves-ide Ψ.preserves-ide Φ-ide INV.Φ-ide Φo-Ψo by auto
  show S′.Fun (Φ (Ψ f′)) = S′.Fun f′
  proof −
    have S′.arr (Φ (Ψ f′)) using assms by blast
    moreover have Φ (Ψ f′) =
                S′.mkArr (S′.set (S′.dom f′)) (S′.set (S′.cod f′)) (Φa (Ψ f′))
      using assms Φ-def INV.Φ-in-hom Φo-Ψo by simp
    moreover have Φa (Ψ f′) = (λx′ ∈ S′.set (S′.dom f′). φ (S.Fun (Ψ f′) (ψ x′)))
      unfolding Φa-def
      using assms par Ψ.preserves-arr set-dom-Φ by metis
    ultimately have 1: S′.Fun (Φ (Ψ f′)) =
                      (λx′ ∈ S′.set (S′.dom f′). φ (S.Fun (Ψ f′) (ψ x′)))
      using S′.Fun-mkArr by simp
    show ?thesis
    proof
      fix x′
      have x′ ∉ S′.set (S′.dom f′) ⟹ S′.Fun (Φ (Ψ f′)) x′ = S′.Fun f′ x′
        using 1 assms S′.Fun-mapsto extensional-def by (simp add: S′.Fun-def)
      moreover have x′ ∈ S′.set (S′.dom f′) ⟹ S′.Fun (Φ (Ψ f′)) x′ = S′.Fun f′ x′
      proof −
        assume x′: x′ ∈ S′.set (S′.dom f′)
        have S′.Fun (Φ (Ψ f′)) x′ = φ (S.Fun (Ψ f′) (ψ x′))
          using x′ 1 by auto
        also have ... = φ (Ψa f′ (ψ x′))
          using Fun-Ψ x′ assms S′.set-subset-Univ bij-φ by metis
        also have ... = φ (ψ (S′.Fun f′ (φ (ψ x′))))
        proof −
          have φ (Ψa f′ (ψ x′)) = φ (ψ (S′.Fun f′ x′))
          proof −
            have x′ ∈ S′.Univ
              by (meson S′.ide-dom S′.set-subset-Univ assms subsetCE x′)
            thus ?thesis
              by (simp add: INV.Φa-def INV.ψ-φ x′)
          qed
          also have ... = φ (ψ (S′.Fun f′ (φ (ψ x′))))
            using assms x′ φ-ψ S′.set-subset-Univ 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′.set-subset-Univ S′.ide-dom by (metis subsetCE)
          moreover have S′.Fun f′ x′ ∈ S′.Univ
            using x′ assms S′.Fun-mapsto S′.set-subset-Univ S′.ide-cod by blast
```

ultimately show *?thesis* **using** *2* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed
ultimately show $S'.Fun\ (\Phi\ (\Psi\ f'))\ x' = S'.Fun\ f'\ x'$ **by** *auto*
qed
qed
qed

lemma *inverse-functors-*Φ*-*Ψ:
shows *inverse-functors* $S\ S'\ \Phi\ \Psi$
proof −
interpret $\Phi\Psi$: *composite-functor* $S\ S'\ S\ \Phi\ \Psi$ **..**
have *inv*: $\Psi\ o\ \Phi = S.map$
using $\Phi\Psi$ *S.map-def* $\Phi\Psi$*.is-extensional* **by** *auto*

interpret $\Psi\Phi$: *composite-functor* $S'\ S\ S'\ \Psi\ \Phi$ **..**
have *inv'*: $\Phi\ o\ \Psi = S'.map$
using $\Psi\Phi$ *S'.map-def* $\Psi\Phi$*.is-extensional* **by** *auto*

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

lemma *are-isomorphic*:
shows $\exists\,\Phi.$ *invertible-functor* $S\ S'\ \Phi \wedge (\forall\,m.\ S.incl\ m \longrightarrow S'.incl\ (\Phi\ m))$
proof −
interpret *inverse-functors* $S\ S'\ \Phi\ \Psi$
using *inverse-functors-*Φ*-*Ψ **by** *auto*
have *1*: *inverse-functors* $S\ S'\ \Phi\ \Psi$ **..**
interpret *invertible-functor* $S\ S'\ \Phi$
apply *unfold-locales* **using** *1* **by** *auto*
have *invertible-functor* $S\ S'\ \Phi$ **..**
thus *?thesis* **using** Φ*-preserves-incl* **by** *auto*
qed

end

theorem *set-category-is-categorical*:
assumes *set-category* S **and** *set-category* S'
and *bij-betw* φ (*set-category-data.Univ* S) (*set-category-data.Univ* S')
shows $\exists\,\Phi.$ *invertible-functor* $S\ S'\ \Phi\ \wedge$
$(\forall\,m.\ set\text{-}category.incl\ S\ m \longrightarrow set\text{-}category.incl\ S'\ (\Phi\ m))$
proof −
interpret S: *set-category* S **using** *assms*(*1*) **by** *auto*
interpret S': *set-category* S' **using** *assms*(*2*) **by** *auto*
interpret *two-set-categories-bij-betw-Univ* $S\ S'\ \varphi$
apply (*unfold-locales*) **using** *assms*(*3*) **by** *auto*

```
    show ?thesis using are-isomorphic by auto
  qed
```

10.4 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)
```

10.4.1 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 auto
      show ⋀b. ide b ⟹ ∃!f. «f : empty → b»
      proof −
        fix b
        assume b: ide b
        show ∃!f. «f : empty → b»
        proof
          show 1: «mkArr {} (set b) (λx. x) : empty → b»
            using b empty-def mkArr-in-hom mkIde-set set-subset-Univ
            by (metis 0 Pi-empty UNIV-I arr-mkIde)
          show ⋀f. «f : empty → b» ⟹ f = mkArr {} (set b) (λx. x)
          proof −
            fix f
            assume f: «f : empty → b»
            show f = mkArr {} (set b) (λx. x)
            proof (intro arr-eqI)
              show 1: par f (mkArr {} (set b) (λx. x))
                using 1 f by force
              show Fun f = Fun (mkArr {} (set b) (λx. x))
                using empty-def 1 f Fun-mapsto by fastforce
            qed
          qed
```

    qed
  qed
qed

10.4.2 Identity Arrows

Identity arrows correspond to restrictions of the identity function.

lemma *ide-char*:
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* **by** *blast*
qed

10.4.3 Inclusions

lemma *ide-implies-incl*:
assumes *ide a*
shows *incl a*
proof −
  have *arr a* $\wedge$ *Dom a* $\subseteq$ *Cod a* **using** *assms* **by** *auto*
  moreover have $a =$ *mkArr* (*Dom a*) (*Cod a*) $(\lambda x.\ x)$
    using *assms* **by** *simp*
  ultimately show *?thesis* **using** *incl-def* **by** *simp*
qed

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

abbreviation *incl-of*
where *incl-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 set-subset-Univ terminal-char2*
    by (*metis mem-Collect-eq subsetCE*)
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 $set\ b \subseteq Univ$ **by** *simp*
  hence $a \in Univ \wedge set\ a \subseteq set\ b$
    using *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 set-subset-Univ*)
  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 $Dom\ (g \cdot f) \subseteq Cod\ (g \cdot f)$
    using *assms 1 incl-def* **by** *auto*
  moreover have $g \cdot f = mkArr\ (Dom\ f)\ (Cod\ g)\ (restrict\ (\lambda x.\ x)\ (Dom\ f))$
    using *assms 1 Fun-comp incl-def Fun-mkArr mkArr-Fun Fun-ide comp-cod-arr*
        ide-dom dom-comp cod-comp
    by *metis*
  ultimately show *?thesis* **using** *incl-def* **by** *force*
qed

10.4.4 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 *Cod f* ⊆ *Univ* **using** *assms* **by** *simp*
  ultimately show *?thesis* **using** *img-def* **by** *simp*
qed

lemma *set-img* [*simp*]:
assumes *arr f*
shows *set* (*img f*) = *Img f*
proof –
  have *Fun f* ‘ *set* (*dom f*) ⊆ *set* (*cod f*) ∧ *set* (*cod f*) ⊆ *Univ*
    using *assms Fun-mapsto* **by** *auto*
  hence *Fun f* ‘ *set* (*dom f*) ⊆ *Univ* **by** *auto*
  thus *?thesis* **using** *assms img-def set-mkIde* **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 img-def terminal-unity terminal-char2*
        image-empty image-insert mem-Collect-eq set-img
    by *force*
  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* ∧ *Cod f* ⊆ *Univ*
    using *assms Fun-mapsto* **by** *auto*
  hence *2*: *ide* (*mkIde* (*Img f*)) **by** *fastforce*
  moreover have *ide* (*cod f*) **using** *assms* **by** *auto*
  moreover have *set* (*mkIde* (*Img f*)) ⊆ *Cod f*
    using *1 2* **by** *force*
  ultimately show *incl-in* (*mkIde* (*Img f*)) (*cod f*)
    using *incl-in-def* **by** *blast*
qed

lemma *img-point-elem-set*:
assumes $\ll x : unity \rightarrow a \gg$
shows *img x* $\in$ *set a*
proof $-$
  have *incl-in* (*img x*) *a*
    using *assms incl-in-img-cod* **by** *auto*
  hence *set* (*img x*) $\subseteq$ *set a*
    using *incl-in-def* **by** *blast*
  moreover have *img x* $\in$ *set* (*img x*)
    using *assms img-point-in-Univ terminal-char2* **by** *simp*
  ultimately show *?thesis* **by** *auto*
qed

The corestriction of an arrow f is the arrow *corestr f* $\in$ *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 $\ll$*corestr f* : *dom f* $\rightarrow$ *img f*$\gg$
proof $-$
  have *Fun f* $\in$ *Dom f* $\rightarrow$ *Fun f* ‘ *Dom f* $\wedge$ *Dom f* $\subseteq$ *Univ*
    using *assms* **by** *auto*
  moreover have *Fun f* ‘ *Dom f* $\subseteq$ *Univ*
  proof $-$
    have *Fun f* ‘ *Dom f* $\subseteq$ *Cod f* $\wedge$ *Cod f* $\subseteq$ *Univ*
      using *assms Fun-mapsto* **by** *auto*
    thus *?thesis* **by** *blast*
  qed
  ultimately have *mkArr* (*Dom f*) (*Fun f* ‘ *Dom f*) (*Fun f*) $\in$ *hom* (*dom f*) (*img f*)
    using *assms img-def mkArr-in-hom* [*of Dom f Fun f* ‘ *Dom f Fun f*] **by** *simp*
  thus *?thesis* **using** *corestr-def* **by** *fastforce*
qed

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*)
  have *1*: $\ll$*corestr f* : *dom f* $\rightarrow$ *img f*$\gg$
    using *assms corestr-in-hom* **by** *blast*
  moreover have *2*: $\ll$*incl-of* (*img f*) (*cod f*) : *img f* $\rightarrow$ *cod f*$\gg$
    using *assms incl-in-img-cod incl-incl-of* **by** *fast*
  ultimately show *P*: *par* (*incl-of* (*img f*) (*cod f*) $\cdot$ *corestr f*) *f*
    using *assms in-homE* **by** *blast*
  show *Fun* (*incl-of* (*img f*) (*cod f*) $\cdot$ *corestr f*) = *Fun f*
  proof $-$

have *Fun* (*incl-of* (*img f*) (*cod f*) · *corestr f*)
= *restrict* (*Fun* (*incl-of* (*img f*) (*cod f*)) *o Fun* (*corestr f*)) (*Dom f*)
using *Fun-comp 1 2 P* **by** *auto*
also have
... = *restrict* (*restrict* (λ*x*. *x*) (*Img f*) *o restrict* (*Fun f*) (*Dom f*)) (*Dom f*)
proof –
have *Fun* (*corestr f*) = *restrict* (*Fun f*) (*Dom f*)
using *assms corestr-def Fun-mkArr corestr-in-hom* **by** *force*
moreover have *Fun* (*incl-of* (*img f*) (*cod f*)) = *restrict* (λ*x*. *x*) (*Img f*)
proof –
have *arr* (*incl-of* (*img f*) (*cod f*)) **using** *incl-incl-of P* **by** *blast*
moreover have *incl-of* (*img f*) (*cod f*) = *mkArr* (*Img f*) (*Cod f*) (λ*x*. *x*)
using *assms* **by** *fastforce*
ultimately show *?thesis* **using** *assms img-def Fun-mkArr* **by** *metis*
qed
ultimately show *?thesis* **by** *argo*
qed
also have ... = *Fun f*
proof
fix *x*
show *restrict* (*restrict* (λ*x*. *x*) (*Img f*) *o restrict* (*Fun f*) (*Dom f*)) (*Dom f*) *x* = *Fun f x*
using *assms extensional-restrict Fun-mapsto extensional-arb* [*of Fun f Dom f x*]
by (*cases x* ∈ *Dom f*, *auto*)
qed
finally show *?thesis* **by** *auto*
qed
qed

lemma *Fun-corestr*:
assumes *arr f*
shows *Fun* (*corestr f*) = *Fun f*
proof –
have *1*: *f* = *incl-of* (*img f*) (*cod f*) · *corestr f*
using *assms img-fact* **by** *auto*
hence *2*: *Fun f* = *restrict* (*Fun* (*incl-of* (*img f*) (*cod f*)) *o Fun* (*corestr f*)) (*Dom f*)
using *assms* **by** (*metis Fun-comp dom-comp*)
also have ... = *restrict* (*Fun* (*corestr f*)) (*Dom f*)
using *assms* **by** (*metis 1 2 Fun-mkArr seqE mkArr-Fun corestr-def*)
also have ... = *Fun* (*corestr f*)
using *assms 1* **by** (*metis Fun-def dom-comp extensional-restrict restrict-extensional*)
finally show *?thesis* **by** *auto*
qed

10.4.5 Points and Terminal Objects

To each element t of *set a* is associated a point *mkPoint a t* ∈ *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* ≡ *mkArr* {*unity*} (*set a*) (λ-. *t*)

lemma *mkPoint-in-hom*:
assumes *ide a* **and** $t \in$ *set a*
shows «*mkPoint a t* : *unity* → *a*»
  using *assms mkArr-in-hom*
  by (*metis Pi-I mkIde-set set-subset-Univ terminal-char2 terminal-unity mkPoint-def*)

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

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* ∈ *hom unity a* → *set a*
and ⋀*x*. «*x* : *unity* → *a*» ⟹ *mkPoint a* (*img x*) = *x*
proof −
  show *img* ∈ *hom unity a* → *set a*
    using *img-point-elem-set* **by** *simp*
  show ⋀*x*. «*x* : *unity* → *a*» ⟹ *mkPoint a* (*img x*) = *x*
  proof −
    fix *x*
    assume *x*: «*x* : *unity* → *a*»
    show *mkPoint a* (*img x*) = *x*
    proof (*intro arr-eqI*)
      have *0*: *img x* ∈ *set a*
        using *x img-point-elem-set* **by** *metis*
      hence *1*: *mkPoint a* (*img x*) ∈ *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- ∈ {*unity*}. *img x*)
        using *1 mkPoint-def* **by** *auto*
      also have ... = *Fun x*
      proof
        fix *z*
        have *z* ≠ *unity* ⟹ (λ- ∈ {*unity*}. *img x*) *z* = *Fun x z*
          using *x Fun-mapsto Fun-def restrict-apply singletonD terminal-char2 terminal-unity*
          by *auto*
        moreover have (λ- ∈ {*unity*}. *img x*) *unity* = *Fun x unity*
          using *x 0 elem-set-implies-set-eq-singleton set-img terminal-char2 terminal-unity*
          by (*metis 2 image-insert in-homE restrict-apply singletonI singleton-insert-inj-eq*)
        ultimately show (λ- ∈ {*unity*}. *img x*) *z* = *Fun x z* **by** *auto*
      qed
      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 ⋀t. t ∈ set a ⟹ img (mkPoint a t) = t
proof -
  show mkPoint a ∈ set a → hom unity a
    using assms(1) mkPoint-in-hom by simp
  show ⋀t. t ∈ set a ⟹ 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 1 t elem-set-implies-incl-in elem-set-implies-set-eq-singleton img-def
                 incl-in-def dom-mkArr mkIde-set terminal-char2 terminal-unity mkPoint-def)
    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 ⋀x. x ∈ hom unity a ⟹ mkPoint a (img x) = x
    using assms mkPoint-img by auto
  show ⋀t. t ∈ set a ⟹ img (mkPoint a t) = t
    using assms img-mkPoint by auto
qed
```

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 → a»
shows Fun x = (λ- ∈ {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*)
  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* $\rightarrow$ *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* $\rightarrow$ *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* **by** *auto*
    also have ... = (λ- $\in$ {*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*:
assumes *arr f* **and** «*x* : *unity* $\rightarrow$ *dom f*»
shows $f \cdot x =$ *mkPoint* (*cod f*) (*Fun f* (*img x*))
proof −
  have $x =$ *mkPoint* (*dom f*) (*img x*) **using** *assms mkPoint-img* **by** *simp*
  thus *?thesis* **using** *assms comp-arr-mkPoint* [*of f img x*]
    by (*simp add*: *img-point-elem-set*)
qed

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*: $f \cdot$ *mkPoint* (*dom f*) $t =$ *mkPoint* (*cod f*) (*Fun f t*)

using *assms t comp-arr-mkPoint* **by** *simp*
hence *img* $(f \cdot mkPoint\ (dom\ f)\ t) = img\ (mkPoint\ (cod\ f)\ (Fun\ f\ t))$ **by** *simp*
thus *?thesis*
proof −
have $Fun\ f\ t \in Cod\ f$ **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'*:
assumes $par\ f\ f'$ **and** $\bigwedge x.\ \ll x : unity \to dom\ f\gg \Longrightarrow f \cdot x = f' \cdot x$
shows $f = f'$
using *assms Fun-in-terms-of-comp mkPoint-in-hom* **by** (*intro arr-eqI*, *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** $F \in hom\ unity\ a \to hom\ unity\ b$
shows $\ll mkArr'\ a\ b\ F : a \to b\gg$
proof −
have $img\ o\ F\ o\ mkPoint\ a \in set\ a \to set\ b$
proof
fix *t*
assume *t*: $t \in set\ a$
thus $(img\ o\ F\ o\ mkPoint\ a)\ t \in set\ b$
using *assms mkPoint-in-hom img-point-elem-set* [*of F* (*mkPoint a t*) *b*]
by *auto*
qed
thus *?thesis*
using *assms mkArr'-def mkArr-in-hom* [*of set a set b*] **by** *simp*
qed

lemma *comp-point-mkArr'*:
assumes *ide a* **and** *ide b* **and** $F \in hom\ unity\ a \to hom\ unity\ b$
shows $\bigwedge x.\ \ll x : unity \to a\gg \Longrightarrow mkArr'\ a\ b\ F \cdot x = F\ x$
proof −
fix *x*
assume *x*: $\ll x : unity \to a\gg$
have $Fun\ (mkArr'\ a\ b\ F)\ (img\ x) = img\ (F\ x)$
unfolding *mkArr'-def*
using *assms x Fun-mkArr arr-mkArr img-point-elem-set mkPoint-img mkPoint-in-hom*
by (*simp add*: *Pi-iff*)

hence $mkArr'\ a\ b\ F \cdot x = mkPoint\ b\ (img\ (F\ x))$
using *assms x mkArr'-in-hom [of a b F] comp-arr-point* **by** *auto*
thus $mkArr'\ a\ b\ F \cdot x = F\ x$
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 $\exists!x.\ \ll x : unity \to a\gg$
shows *terminal a*
proof −
have *a*: *ide a*
using *assms ide-cod mem-Collect-eq* **by** *blast*
hence *1*: *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 $hom\ unity\ a = \{THE\ x.\ x \in hom\ unity\ a\}$
using *assms theI' [of* $\lambda x.\ x \in hom\ unity\ a$*]* **by** *auto*
ultimately have $set\ a = \{img\ (THE\ x.\ x \in hom\ unity\ a)\}$
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 $\exists!f.\ \ll f : a \to b\gg \wedge (\forall x.\ \ll x : unity \to a\gg \longrightarrow f \cdot x = F\ x)$
proof
have *1*: $\ll mkArr'\ a\ b\ F : a \to b\gg$ **using** *assms mkArr'-in-hom* **by** *auto*
moreover have *2*: $\bigwedge x.\ \ll x : unity \to a\gg \Longrightarrow mkArr'\ a\ b\ F \cdot x = F\ x$
using *assms comp-point-mkArr'* **by** *auto*
ultimately show $\ll mkArr'\ a\ b\ F : a \to b\gg \wedge$
$(\forall x.\ \ll x : unity \to a\gg \longrightarrow mkArr'\ a\ b\ F \cdot x = F\ x)$ **by** *blast*
fix *f*
assume *f*: $\ll f : a \to b\gg \wedge (\forall x.\ \ll x : unity \to a\gg \longrightarrow f \cdot x = F\ x)$
show $f = mkArr'\ a\ b\ F$
using *f 1 2* **by** (*intro arr-eqI' [of f mkArr' a b F], fastforce, auto*)
qed

10.4.6 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'* $\longleftrightarrow$
  ($\exists$ *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* $\cup$ *Cod f'*)
  let *?m = incl-of* (*cod f*) *?b*
  let *?m' = incl-of* (*cod f'*) *?b*
  have *incl ?m*
    using *assms incl-incl-of* [*of cod f ?b*] *incl-in-def* **by** *simp*
  have *incl ?m'*
    using *assms incl-incl-of* [*of cod f' ?b*] *incl-in-def* **by** *simp*
  have *m*: *?m = mkArr* (*Cod f*) (*Cod f* $\cup$ *Cod f'*) (λx. *x*)
    by (*simp add*: *assms*)
  have *m'*: *?m' = mkArr* (*Cod f'*) (*Cod f* $\cup$ *Cod f'*) (λx. *x*)
    by (*simp add*: *assms*)
  have *seq*: *seq ?m f* $\wedge$ *seq ?m' f'*
    using *assms m m'* **by** *simp*
  have *?m* $\cdot$ *f* = *?m'* $\cdot$ *f'*
  proof (*intro arr-eqI*)
    show *par*: *par* (*?m* $\cdot$ *f*) (*?m'* $\cdot$ *f'*)
      using *assms m m'* **by** *simp*
    show *Fun* (*?m* $\cdot$ *f*) = *Fun* (*?m'* $\cdot$ *f'*)
      using *assms seq par ff' Fun-mapsto Fun-comp seqE*
      by (*metis Fun-ide Fun-mkArr comp-cod-arr ide-cod*)
  qed
  hence *incl ?m* $\wedge$ *incl ?m'* $\wedge$ *seq ?m f* $\wedge$ *seq ?m' f'* $\wedge$ *?m* $\cdot$ *f* = *?m'* $\cdot$ *f'*
    using *seq* ‹*incl ?m*› ‹*incl ?m'*› **by** *simp*
  thus $\exists$ *m m'*. *incl m* $\wedge$ *incl m'* $\wedge$ *seq m f* $\wedge$ *seq m' f'* $\wedge$ *m* $\cdot$ *f* = *m'* $\cdot$ *f'* **by** *auto*
  next
  assume *ff'*: $\exists$ *m m'*. *incl m* $\wedge$ *incl m'* $\wedge$ *seq m f* $\wedge$ *seq m' f'* $\wedge$ *m* $\cdot$ *f* = *m'* $\cdot$ *f'*
  show *Fun f = Fun f'*
  proof –
    from *ff'* **obtain** *m m'*
    where *mm'*: *incl m* $\wedge$ *incl m'* $\wedge$ *seq m f* $\wedge$ *seq m' f'* $\wedge$ *m* $\cdot$ *f* = *m'* $\cdot$ *f'*
      by *blast*
    show *?thesis*
      using *ff' mm' Fun-incl seqE*
      by (*metis Fun-comp Fun-ide comp-cod-arr ide-cod*)
  qed
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'* $\longleftrightarrow$ *corestr f = corestr f'*
  using *assms corestr-def Fun-corestr* **by** *metis*

10.4.7 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
  proof
    have Cod g ⊆ Univ ∧ Dom g ⊆ Univ using assms by auto
    moreover have ?F ∈ Cod g → Dom g
    proof
      fix y
      assume y: y ∈ Cod g
      let ?P = λx. x ∈ Dom g ∧ Fun g x = y
      have ∃ x. ?P x using y assms by force
      hence ?P (SOME x. ?P x) using someI-ex [of ?P] by fast
      hence ?P (?F y) using Hilbert-Choice.inv-into-def by metis
      thus ?F y ∈ Dom g by auto
    qed
    ultimately show Cod g ⊆ Univ ∧ Dom g ⊆ Univ ∧ ?F ∈ Cod g → Dom g by auto
  qed
  show ide (g · ?f)
  proof –
    have g = mkArr (Dom g) (Cod g) (Fun g) using assms 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 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 *restrict* (*Fun g o Fun f*) (*Cod g*) = *restrict* (λ*x*. *x*) (*Cod g*)
using *f Fun-comp Fun-ide ide-compE* **by** *metis*
hence *Fun g* ‘ *Fun f* ‘ *Cod g* = *Cod g*
by (*metis image-comp image-ident image-restrict-eq*)
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* **by** *fastforce*

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*: *Cod f* ⊆ *Univ* **using** *assms* **by** *simp*
have *2*: *Dom f* ⊆ *Univ* **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 λx. 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 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) (λx. x) = ...
    proof (intro mkArr-eqI')
      show arr (mkArr (Dom f) (Dom f) (λx. 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 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 $\mathit{Cod}\ f \neq \{\} \longrightarrow \mathit{Dom}\ f \neq \{\}$
proof
assume $\mathit{Cod}\ f \neq \{\}$
from *this* **obtain** y **where** $y \in \mathit{Cod}\ f$ **by** *blast*
hence $\mathit{Fun}\ g\ y \in \mathit{Dom}\ f$
using *g Fun-mapsto*
by (*metis seqE ide-compE image-eqI retractionI retraction-char*)
thus $\mathit{Dom}\ f \neq \{\}$ **by** *blast*
qed
thus *?thesis* **by** *auto*
qed
moreover have *inj-on* (*Fun f*) (*Dom f*)
proof –
have *restrict* (*Fun g o Fun f*) (*Dom f*) = *Fun* $(g \cdot f)$
using *g Fun-comp* **by** (*metis Fun-comp ide-compE*)
also have ... = *restrict* $(\lambda x.\ x)$ (*Dom f*)
using *g Fun-ide* **by** *auto*
finally have *restrict* (*Fun g o Fun f*) (*Dom f*) = *restrict* $(\lambda x.\ x)$ (*Dom f*) **by** *auto*
thus *?thesis* **using** *inj-onI inj-on-imageI2 inj-on-restrict-eq* **by** *metis*
qed
ultimately show *?thesis* **by** *auto*
qed
next
assume F: $\mathit{arr}\ f \wedge (\mathit{Dom}\ f = \{\} \longrightarrow \mathit{Cod}\ f = \{\}) \wedge$ *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 $\mathit{ide}\ (g \cdot f) \longleftrightarrow \mathit{antipar}\ f\ g \wedge \mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f) = (\lambda x \in \mathit{Dom}\ f.\ x)$
proof
show $\mathit{ide}\ (g \cdot f) \Longrightarrow \mathit{antipar}\ f\ g \wedge \mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f) = (\lambda x \in \mathit{Dom}\ f.\ x)$
proof –
assume *fg*: $\mathit{ide}\ (g \cdot f)$
have *1*: *antipar f g* **using** *fg* **by** *force*
moreover have $\mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f) = (\lambda x \in \mathit{Dom}\ f.\ x)$
proof
fix x
have $x \notin \mathit{Dom}\ f \Longrightarrow \mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f)\ x = (\lambda x \in \mathit{Dom}\ f.\ x)\ x$
by (*simp add*: *compose-def*)
moreover have $x \in \mathit{Dom}\ f \Longrightarrow$
$\mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f)\ x = (\lambda x \in \mathit{Dom}\ f.\ x)\ x$
using *fg 1 Fun-comp* **by** (*metis Fun-comp Fun-ide compose-eq′ ide-compE*)
ultimately show $\mathit{compose}\ (\mathit{Dom}\ f)\ (\mathit{Fun}\ g)\ (\mathit{Fun}\ f)\ x = (\lambda x \in \mathit{Dom}\ f.\ x)\ x$ **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed

show *antipar f g* $\wedge$ *compose* (*Dom f*) (*Fun g*) (*Fun f*) = ($\lambda x \in$ *Dom f*. *x*) $\Longrightarrow$ *ide* (*g* $\cdot$ *f*)
proof –
assume *fg*: *antipar f g* $\wedge$ *compose* (*Dom f*) (*Fun g*) (*Fun f*) = ($\lambda x \in$ *Dom f*. *x*)
show *ide* (*g* $\cdot$ *f*)
proof –
have *1*: *arr* (*g* $\cdot$ *f*) **using** *fg* **by** *auto*
moreover have *Dom* (*g* $\cdot$ *f*) = *Cod* (*S g f*)
using *fg 1* **by** *force*
moreover have *Fun* (*g* $\cdot$ *f*) = ($\lambda x \in$ *Dom* (*g* $\cdot$ *f*). *x*)
using *fg 1* **by** *force*
ultimately show *?thesis* **using** *1 ide-char* **by** *blast*
qed
qed
qed

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*)
proof –
have *iso f* $\longleftrightarrow$ *section f* $\wedge$ *retraction f*
using *iso-iff-section-and-retraction* **by** *auto*
also have ... $\longleftrightarrow$ *arr f* $\wedge$ *inj-on* (*Fun f*) (*Dom f*) $\wedge$ *Img f* = *Cod f*
using *section-char retraction-char* **by** *force*
also have ... $\longleftrightarrow$ *arr f* $\wedge$ *bij-betw* (*Fun f*) (*Dom f*) (*Cod f*)
using *inj-on-def bij-betw-def* [*of Fun f Dom f Cod f*] **by** *meson*
finally show *?thesis* **by** *auto*
qed

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*)
(λy. *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 $\bigwedge y.\ y \in Cod\ f \Longrightarrow (if\ y \in Img\ f\ then\ SOME\ x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
$else\ SOME\ x.\ x \in Dom\ f)$
$= inv\text{-}into\ (Dom\ f)\ (Fun\ f)\ y$
proof −
fix *y*
assume $y \in Cod\ f$
hence $y \in Img\ f$ **using** *assms iso-is-retraction retraction-char* **by** *metis*
thus $(if\ y \in Img\ f\ then\ SOME\ x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
$else\ SOME\ x.\ x \in Dom\ f)$
$= inv\text{-}into\ (Dom\ f)\ (Fun\ f)\ y$
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 $Fun\ (inv\ f) = restrict\ (inv\text{-}into\ (Dom\ f)\ (Fun\ f))\ (Cod\ f)$
using *assms inv-in-hom inv-char iso-inv-iso iso-is-arr Fun-mkArr* **by** *metis*

10.4.8 Monomorphisms and Epimorphisms

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

lemma *mono-char*:
shows $mono\ f \longleftrightarrow arr\ f \wedge inj\text{-}on\ (Fun\ f)\ (Dom\ f)$
proof
assume *f*: *mono f*
hence *arr f* **using** *mono-def* **by** *auto*
moreover have $inj\text{-}on\ (Fun\ f)\ (Dom\ f)$
proof (*intro inj-onI*)
have *0*: $inj\text{-}on\ (S\ f)\ (hom\ unity\ (dom\ f))$
proof −
have $hom\ unity\ (dom\ f) \subseteq \{g.\ seq\ f\ g\}$
using *f mono-def arrI* **by** *auto*
hence $\exists A.\ hom\ unity\ (dom\ f) \subseteq A \wedge inj\text{-}on\ (S\ f)\ A$
using *f mono-def* **by** *auto*
thus *?thesis*
by (*meson subset-inj-on*)
qed
fix *x x′*
assume *x*: $x \in Dom\ f$ **and** *x′*: $x' \in Dom\ f$ **and** *xx′*: $Fun\ f\ x = Fun\ f\ x'$

```
      have 1: 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
      have f · mkPoint (dom f) x = f · mkPoint (dom f) x'
        using ‹arr f› x x' xx' comp-arr-mkPoint by simp
      hence mkPoint (dom f) x = mkPoint (dom f) x'
        using 0 1 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 ∧ seq f g' ∧ f · g = f · g' ⟹ g = g'
      proof −
        fix g g'
        assume gg': seq f g ∧ seq f g' ∧ f · g = f · g'
        show g = g'
        proof (intro arr-eqI)
          show par: par g g'
            using gg' dom-comp by (metis seqE)
          show Fun g = Fun g'
          proof
            fix x
            have x ∉ Dom g ⟹ Fun g x = Fun g' x
              using gg' by (simp add: par Fun-def)
            moreover have x ∈ Dom g ⟹ Fun g x = Fun g' x
            proof −
              assume x: x ∈ Dom g
              have Fun f (Fun g x) = Fun (f · g) x
                using gg' x Fun-comp [of f g] by auto
              also have ... = Fun f (Fun g' x)
                using par f gg' x monoE by simp
              finally have Fun f (Fun g x) = Fun f (Fun g' x) by auto
              moreover have Fun g x ∈ Dom f ∧ Fun g' x ∈ Dom f
                using par gg' x Fun-mapsto by fastforce
              ultimately show Fun g x = Fun g' x
                using f gg' inj-onD [of Fun f Dom f Fun g x Fun g' x]
                by simp
            qed
            ultimately show Fun g x = Fun g' x by auto
          qed
        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** $Dom\ f = \{\} \longrightarrow Cod\ f = \{\}$
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 $epi\ f \longleftrightarrow arr\ f \wedge (Img\ f = Cod\ f \vee (\forall t\ t'.\ t \in Univ \wedge t' \in Univ \longrightarrow t = t'))$
proof
  assume *epi*: *epi f*
  show $arr\ f \wedge (Img\ f = Cod\ f \vee (\forall t\ t'.\ t \in Univ \wedge t' \in Univ \longrightarrow t = t'))$
  proof −
    have *f*: *arr f* **using** *epi epi-implies-arr* **by** *auto*
    moreover have $\neg(\forall t\ t'.\ t \in Univ \wedge t' \in Univ \longrightarrow t = t') \Longrightarrow Img\ f = Cod\ f$
    proof −
      assume $\neg(\forall t\ t'.\ t \in Univ \wedge t' \in Univ \longrightarrow t = t')$
      from *this* **obtain** *tt* **and** *ff*
        where *B*: $tt \in Univ \wedge ff \in Univ \wedge tt \neq ff$ **by** *blast*
      show $Img\ f = Cod\ f$
      proof
        show $Img\ f \subseteq Cod\ f$ **using** *f Fun-mapsto* **by** *auto*
        show $Cod\ f \subseteq Img\ f$
        proof
          let $?g = mkArr\ (Cod\ f)\ \{ff,\ tt\}\ (\lambda y.\ tt)$
          let $?g' = mkArr\ (Cod\ f)\ \{ff,\ tt\}\ (\lambda y.\ if\ \exists x.\ x \in Dom\ f \wedge Fun\ f\ x = y$
                                    $then\ tt\ else\ ff)$
          let $?b = mkIde\ \{ff,\ tt\}$
          have *g*: $\ll ?g : cod\ f \rightarrow ?b \gg \wedge Fun\ ?g = (\lambda y \in Cod\ f.\ tt)$
            using *f B in-homI* [*of ?g*] **by** *simp*
          have *g'*: $?g' \in hom\ (cod\ f)\ ?b\ \wedge$
                $Fun\ ?g' = (\lambda y \in Cod\ f.\ if\ \exists x.\ x \in Dom\ f \wedge Fun\ f\ x = y\ then\ tt\ else\ ff)$
            using *f B in-homI* [*of ?g'*] **by** *simp*
          have $?g \cdot f = ?g' \cdot f$
          proof (*intro arr-eqI*)
            show $par\ (?g \cdot f)\ (?g' \cdot f)$
              using *f g g'* **by** *auto*
            show $Fun\ (?g \cdot f) = Fun\ (?g' \cdot f)$
              using *f g g' Fun-comp comp-mkArr* **by** *force*
          qed
          hence *gg'*: $?g = ?g'$

using $\mathit{epi}\ f\ g\ g'\ \mathit{epiE}\ [\mathit{of}\ f\ ?g\ ?g']$ **by** *fastforce*
fix y
assume y: $y \in \mathit{Cod}\ f$
have $\mathit{Fun}\ ?g'\ y = \mathit{tt}$ **using** $\mathit{gg}'\ g\ y$ **by** *simp*
hence $(\mathit{if}\ \exists x.\ x \in \mathit{Dom}\ f \wedge \mathit{Fun}\ f\ x = y\ \mathit{then}\ \mathit{tt}\ \mathit{else}\ \mathit{ff}) = \mathit{tt}$
using $g'\ y$ **by** *simp*
hence $\exists x.\ x \in \mathit{Dom}\ f \wedge \mathit{Fun}\ f\ x = y$
using B **by** *argo*
thus $y \in \mathit{Img}\ f$ **by** *blast*
qed
qed
qed
ultimately show $\mathit{arr}\ f \wedge (\mathit{Img}\ f = \mathit{Cod}\ f \vee (\forall t\ t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \longrightarrow t = t'))$
by *fast*
qed
next
show $\mathit{arr}\ f \wedge (\mathit{Img}\ f = \mathit{Cod}\ f \vee (\forall t\ t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \longrightarrow t = t')) \Longrightarrow \mathit{epi}\ f$
proof −
have $\mathit{arr}\ f \wedge \mathit{Img}\ f = \mathit{Cod}\ f \Longrightarrow \mathit{epi}\ f$
proof −
assume f: $\mathit{arr}\ f \wedge \mathit{Img}\ f = \mathit{Cod}\ f$
show $\mathit{epi}\ f$
using f *arr-eqI′ epiE retractionI retraction-if-Img-eq-Cod retraction-is-epi*
by *meson*
qed
moreover have $\mathit{arr}\ f \wedge (\forall t\ t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \longrightarrow t = t') \Longrightarrow \mathit{epi}\ f$
proof −
assume f: $\mathit{arr}\ f \wedge (\forall t\ t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \longrightarrow t = t')$
have $\bigwedge f\ f'.\ \mathit{par}\ f\ f' \Longrightarrow f = f'$
proof −
fix $f\ f'$
assume ff': $\mathit{par}\ f\ f'$
show $f = f'$
proof (*intro arr-eqI*)
show $\mathit{par}\ f\ f'$ **using** ff' **by** *simp*
have $\bigwedge t\ t'.\ t \in \mathit{Cod}\ f \wedge t' \in \mathit{Cod}\ f \Longrightarrow t = t'$
using $f\ \mathit{ff}'$ *set-subset-Univ ide-cod subsetD* **by** *blast*
thus $\mathit{Fun}\ f = \mathit{Fun}\ f'$
using ff' *Fun-mapsto* $[\mathit{of}\ f]$ *Fun-mapsto* $[\mathit{of}\ f']$
extensional-arb $[\mathit{of}\ \mathit{Fun}\ f\ \mathit{Dom}\ f]$ *extensional-arb* $[\mathit{of}\ \mathit{Fun}\ f'\ \mathit{Dom}\ f]$
by *fastforce*
qed
qed
moreover have $\bigwedge g\ g'.\ \mathit{par}\ (g \cdot f)\ (g' \cdot f) \Longrightarrow \mathit{par}\ g\ g'$
by *force*
ultimately show $\mathit{epi}\ f$
using f **by** (*intro epiI*; *metis*)
qed
ultimately show $\mathit{arr}\ f \wedge (\mathit{Img}\ f = \mathit{Cod}\ f \vee (\forall t\ t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \longrightarrow t = t'))$

$\Longrightarrow$ *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** $\exists\, t\, t'.\ t \in \mathit{Univ} \wedge t' \in \mathit{Univ} \wedge t \neq t'$
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** $m \cdot e = m' \cdot e'$
and *incl m* **and** *incl m′* **and** *retraction e* **and** *retraction e′*
shows $m = m'$ **and** $e = e'$
proof –
  have *1*: $\mathit{cod}\ m = \mathit{cod}\ m' \wedge \mathit{dom}\ e = \mathit{dom}\ e'$
    using *assms*(*1*–*3*) **by** (*metis dom-comp cod-comp*)
  hence *2*: *span e e′* **using** *assms*(*1*–*2*) **by** *blast*
  hence *3*: $\mathit{Fun}\ e = \mathit{Fun}\ e'$
    using *assms eq-Fun-iff-incl-joinable* **by** *meson*
  hence $\mathit{img}\ e = \mathit{img}\ e'$ **using** *assms 1 img-def* **by** *auto*
  moreover have $\mathit{img}\ e = \mathit{cod}\ e \wedge \mathit{img}\ e' = \mathit{cod}\ e'$
    using *assms*(*6*–*7*) *retraction-char img-def* **by** *simp*
  ultimately have *par e e′* **using** *2* **by** *simp*
  thus $e = e'$ **using** *3 arr-eqI* **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

10.5 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*
  for *S* :: *′s comp*   (**infixr** $\cdot_S$ *55*)
  and *U* :: *′a set*

and ι :: $'a \Rightarrow 's$ +
assumes *ι-mapsto*: $\iota \in U \rightarrow Univ$
and *inj-ι*: *inj-on* ι U
begin

abbreviation o
where o $\equiv$ *inv-into* U ι

lemma o-*mapsto*:
shows o $\in \iota$ ‘ $U \rightarrow U$
by (*simp add*: *inv-into-into*)

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

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

end

end

Chapter 11

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.

locale *setcat*
begin

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

interpretation *concrete-category* ‹*UNIV* :: *'a set set*› ‹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* (**infixr** · *55*)
where *Comp* ≡ *COMP*
notation *in-hom* («- : - → -»)

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

    by *simp*
  show $Cod\ f = Cod\ (MkArr\ (Dom\ f)\ (Cod\ f)\ (\lambda x \in Dom\ f.\ Map\ f\ x))$
    by *simp*
  show $Map\ f = Map\ (MkArr\ (Dom\ f)\ (Cod\ f)\ (\lambda x \in Dom\ f.\ Map\ f\ x))$
    using *assms arr-char*
    by (*metis* (*mono-tags*, *lifting*) *Int-iff MkArr-Map extensional-restrict*)
qed

lemma *arr-char*:
shows $arr\ f \longleftrightarrow f \neq Null \wedge Map\ f \in extensional\ (Dom\ f) \cap (Dom\ f \to Cod\ f)$
  using *arr-char* **by** *auto*

lemma *terminal-char*:
shows $terminal\ a \longleftrightarrow (\exists x.\ a = MkIde\ \{x\})$
proof
  show $\exists x.\ a = MkIde\ \{x\} \Longrightarrow terminal\ a$
  proof −
    assume a: $\exists x.\ a = MkIde\ \{x\}$
    from *this* **obtain** x **where** x: $a = MkIde\ \{x\}$ **by** *blast*
    have $terminal\ (MkIde\ \{x\})$
    proof
      show $ide\ (MkIde\ \{x\})$
        using *ide-MkIde* **by** *auto*
      show $\bigwedge a.\ ide\ a \Longrightarrow \exists!f.\ \ll f : a \to MkIde\ \{x\}\gg$
      proof
        fix $a :: 'a\ setcat.arr$
        assume a: $ide\ a$
        show $\ll MkArr\ (Dom\ a)\ \{x\}\ (\lambda\text{-}\in Dom\ a.\ x) : a \to MkIde\ \{x\}\gg$
          using *a MkArr-in-hom*
          by (*metis* (*mono-tags*, *lifting*) *IntI MkIde-Dom′ restrictI restrict-extensional*
            singletonI UNIV-I)
        fix $f :: 'a\ setcat.arr$
        assume f: $\ll f : a \to MkIde\ \{x\}\gg$
        show $f = MkArr\ (Dom\ a)\ \{x\}\ (\lambda\text{-} \in Dom\ a.\ x)$
        proof −
          have *1*: $Dom\ f = Dom\ a \wedge Cod\ f = \{x\}$
            using *a f* **by** (*metis* (*mono-tags*, *lifting*) *Dom.simps*(*1*) *in-hom-char*)
          moreover have $Map\ f = (\lambda\text{-} \in Dom\ a.\ x)$
          proof
            fix z
            have $z \notin Dom\ a \Longrightarrow Map\ f\ z = (\lambda\text{-} \in Dom\ a.\ x)\ z$
              using *f 1 MkArr-expansion*
              by (*metis* (*mono-tags*, *lifting*) *Map.simps*(*1*) *in-homE restrict-apply*)
            moreover have $z \in Dom\ a \Longrightarrow Map\ f\ z = (\lambda\text{-} \in Dom\ a.\ x)\ z$
              using *f 1 arr-char* [*of f*] **by** *fastforce*
            ultimately show $Map\ f\ z = (\lambda\text{-} \in Dom\ a.\ x)\ z$ **by** *auto*
          qed
          ultimately show *?thesis*
           using *f MkArr-expansion* [*of f*] **by** *fastforce*

qed
qed
qed
thus *terminal a* **using** *x* **by** *simp*
qed
show *terminal a* $\Longrightarrow \exists\, x.\ a = \mathit{MkIde}\ \{x\}$
proof −
assume *a*: *terminal a*
hence *ide a* **using** *terminal-def* **by** *auto*
have *1*: $\exists!x.\ x \in \mathit{Dom}\ a$
proof −
have $\mathit{Dom}\ a = \{\} \Longrightarrow \neg\mathit{terminal}\ a$
proof −
assume $\mathit{Dom}\ a = \{\}$
hence *1*: $a = \mathit{MkIde}\ \{\}$ **using** ‹*ide a*› *MkIde-Dom′* **by** *force*
have $\bigwedge f.\ f \in \mathit{hom}\ (\mathit{MkIde}\ \{\mathit{undefined}\})\ (\mathit{MkIde}\ (\{\} :: {'a\ set}))$
$\Longrightarrow \mathit{Map}\ f \in \{\mathit{undefined}\} \to \{\}$
proof −
fix *f*
assume *f*: $f \in \mathit{hom}\ (\mathit{MkIde}\ \{\mathit{undefined}\})\ (\mathit{MkIde}\ (\{\} :: {'a\ set}))$
show $\mathit{Map}\ f \in \{\mathit{undefined}\} \to \{\}$
using *f MkArr-expansion arr-char* [*of f*] *in-hom-char* **by** *auto*
qed
hence $\mathit{hom}\ (\mathit{MkIde}\ \{\mathit{undefined}\})\ a = \{\}$ **using** *1* **by** *auto*
moreover have *ide* (*MkIde* {*undefined*}) **using** *ide-MkIde* **by** *auto*
ultimately show ¬*terminal a* **by** *blast*
qed
moreover have $\bigwedge x\ x'.\ x \in \mathit{Dom}\ a \wedge x' \in \mathit{Dom}\ a \wedge x \neq x' \Longrightarrow \neg\mathit{terminal}\ a$
proof −
fix $x\ x'$
assume *1*: $x \in \mathit{Dom}\ a \wedge x' \in \mathit{Dom}\ a \wedge x \neq x'$
have $\ll\mathit{MkArr}\ \{\mathit{undefined}\}\ (\mathit{Dom}\ a)\ (\lambda\text{-} \in \{\mathit{undefined}\}.\ x) : \mathit{MkIde}\ \{\mathit{undefined}\} \to a\gg$
using *1*
by (*metis* (*mono-tags*, *lifting*) *IntI MkIde-Dom′* ‹*ide a*› *restrictI*
restrict-extensional MkArr-in-hom UNIV-I)
moreover have
$\ll\mathit{MkArr}\ \{\mathit{undefined}\}\ (\mathit{Dom}\ a)\ (\lambda\text{-} \in \{\mathit{undefined}\}.\ x') : \mathit{MkIde}\ \{\mathit{undefined}\} \to a\gg$
using *1*
by (*metis* (*mono-tags*, *lifting*) *IntI MkIde-Dom′* ‹*ide a*› *restrictI*
restrict-extensional MkArr-in-hom UNIV-I)
moreover have $\mathit{MkArr}\ \{\mathit{undefined}\}\ (\mathit{Dom}\ a)\ (\lambda\text{-} \in \{\mathit{undefined}\}.\ x) \neq$
$\mathit{MkArr}\ \{\mathit{undefined}\}\ (\mathit{Dom}\ a)\ (\lambda\text{-} \in \{\mathit{undefined}\}.\ x')$
using *1* **by** (*metis arr.inject restrict-apply′ singletonI*)
ultimately show ¬*terminal a*
using *terminal-arr-unique*
by (*metis* (*mono-tags*, *lifting*) *in-homE*)
qed
ultimately show *?thesis*
using *a* **by** *auto*

qed
hence *Dom a* = {*THE x. x* ∈ *Dom a*}
using *theI* [*of* λ*x. x* ∈ *Dom a*] **by** *auto*
hence *a* = *MkIde* {*THE x. x* ∈ *Dom a*}
using *a terminal-def* **by** (*metis* (*mono-tags*, *lifting*) *MkIde-Dom′*)
thus ∃ *x. a* = *MkIde* {*x*}
by *auto*
qed
qed

definition *Img* :: *′a setcat.arr* ⇒ *′a setcat.arr*
where *Img f* = *MkIde* (*Map f* ‘ *Dom f*)

interpretation *set-category-data Comp Img* **..**

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

The inverse maps *UP* and *DOWN* 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 *UP* :: *′a* ⇒ *′a setcat.arr*
where *UP x* ≡ *MkIde* {*x*}

definition *DOWN* :: *′a setcat.arr* ⇒ *′a*
where *DOWN t* ≡ *the-elem* (*Dom t*)

abbreviation *U*
where *U* ≡ *DOWN unity*

lemma *UP-mapsto*:
shows *UP* ∈ *UNIV* → *Univ*
using *terminal-char UP-def* **by** *fast*

lemma *DOWN-mapsto*:
shows *DOWN* ∈ *Univ* → *UNIV*
by *auto*

lemma *DOWN-UP* [*simp*]:
shows *DOWN* (*UP x*) = *x*
by (*simp add*: *DOWN-def UP-def*)

lemma *UP-DOWN* [*simp*]:
assumes *t* ∈ *Univ*
shows *UP* (*DOWN t*) = *t*
using *assms terminal-char UP-def DOWN-def*

by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq DOWN-UP*)

lemma *inj-UP*:
shows *inj UP*
by (*metis DOWN-UP injI*)

lemma *bij-UP*:
shows *bij-betw UP UNIV Univ*
proof (*intro bij-betwI*)
interpret *category Comp* **using** *is-category* **by** *auto*
show *DOWN-UP*: $\bigwedge x :: 'a.\ DOWN\ (UP\ x) = x$ **by** *simp*
show *UP-DOWN*: $\bigwedge t.\ t \in Univ \Longrightarrow UP\ (DOWN\ t) = t$ **by** *simp*
show $UP \in UNIV \rightarrow Univ$ **using** *UP-mapsto* **by** *auto*
show $DOWN \in Collect\ terminal \rightarrow UNIV$ **by** *auto*
qed

lemma *Dom-terminal*:
assumes *terminal t*
shows $Dom\ t = \{DOWN\ t\}$
using *assms UP-def*
by (*metis* (*mono-tags*, *lifting*) *Dom.simps*(*1*) *DOWN-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 $(UP \circ Fun\ p \circ DOWN)\ unity$.

lemma *Img-point*:
assumes $\ll p : unity \rightarrow a\gg$
shows $Img \in hom\ unity\ a \rightarrow Univ$
and $Img\ p = (UP\ o\ Map\ p\ o\ DOWN)\ unity$
proof −
show $Img \in hom\ unity\ a \rightarrow Univ$
proof
fix *f*
assume *f*: $f \in hom\ unity\ a$
have *terminal* (*MkIde* (*Map f* ‘ *Dom unity*))
proof −
obtain $u :: 'a$ **where** *u*: $unity = MkIde\ \{u\}$
using *terminal-unity terminal-char*
by (*metis* (*mono-tags*, *lifting*))
have $Map\ f\ ‘\ Dom\ unity = \{Map\ f\ u\}$
using *u* **by** *simp*
thus *?thesis*
using *terminal-char* **by** *auto*
qed
hence $MkIde\ (Map\ f\ ‘\ Dom\ unity) \in Univ$ **by** *simp*
moreover have $MkIde\ (Map\ f\ ‘\ Dom\ unity) = Img\ f$
using *f dom-char Img-def in-homE*
by (*metis* (*mono-tags*, *lifting*) *Dom.simps*(*1*) *mem-Collect-eq*)
ultimately show $Img\ f \in Univ$ **by** *auto*
qed

  have *Img p* = *MkIde* (*Map p* ‘ *Dom p*) **using** *Img-def* **by** *blast*
  also have ... = *MkIde* (*Map p* ‘ {*U*})
    using *assms in-hom-char terminal-unity Dom-terminal*
    by (*metis* (*mono-tags*, *lifting*))
  also have ... = (*UP o Map p o DOWN*) *unity* **by** (*simp add*: *UP-def*)
  finally show *Img p* = (*UP o Map p o DOWN*) *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* :: *'a setcat.arr => 'a setcat.arr => 'a setcat.arr*
where *MkElem t a* ≡ *MkArr* {*U*} (*Dom a*) (λ- ∈ {*U*}. *DOWN t*)

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

lemma *MkElem-Img*:
assumes *p* ∈ *hom unity a*
shows *MkElem* (*Img p*) *a* = *p*
proof −
  have *0*: *Img p* = *UP* (*Map p U*)
    using *assms Img-point*(*2*) **by** *auto*
  have *1*: *Dom p* = {*U*}
    using *assms terminal-unity Dom-terminal*
    by (*metis* (*mono-tags*, *lifting*) *in-hom-char mem-Collect-eq*)
  moreover have *Cod p* = *Dom a*
    using *assms*
    by (*metis* (*mono-tags*, *lifting*) *in-hom-char mem-Collect-eq*)
  moreover have *Map p* = (λ- ∈ {*U*}. *DOWN* (*Img p*))
  proof
    fix *e*
    show *Map p e* = (λ- ∈ {*U*}. *DOWN* (*Img p*)) *e*
    proof −
      have *Map p e* = (λ*x* ∈ *Dom p*. *Map p x*) *e*
        using *assms MkArr-expansion* [*of p*]
        by (*metis* (*mono-tags*, *lifting*) *CollectD Map.simps*(*1*) *in-homE*)

    also have ... = (λ- ∈ {*U*}. *DOWN* (*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 MkArr-Map CollectD*
  by (*metis* (*mono-tags*, *lifting*) *in-homE mem-Collect-eq*)
qed

lemma *inj-Img*:
assumes *ide a*
shows *inj-on Img* (*hom unity a*)
proof (*intro inj-onI*)
  fix *x y*
  assume *x*: *x* ∈ *hom unity a*
  assume *y*: *y* ∈ *hom unity a*
  assume *eq*: *Img x* = *Img y*
  show *x* = *y*
  proof (*intro arr-eqI*)
    show *arr x* **using** *x* **by** *blast*
    show *arr y* **using** *y* **by** *blast*
    show *Dom x* = *Dom y*
      using *x y in-hom-char* **by** (*metis* (*mono-tags*, *lifting*) *CollectD*)
    show *Cod x* = *Cod y*
      using *x y in-hom-char* **by** (*metis* (*mono-tags*, *lifting*) *CollectD*)
    show *Map x* = *Map y*
    proof −
      have *Map x* = (λ*z* ∈ {*U*}. *Map x z*) ∧ *Map y* = (λ*z* ∈ {*U*}. *Map y z*)
        using *x y* ‹*arr x*› ‹*arr y*› *Dom-terminal terminal-unity MkArr-expansion*
        by (*metis* (*mono-tags*, *lifting*) *CollectD Map.simps*(*1*) *in-hom-char*)
      moreover have *Map x U* = *Map y U*
        using *x y eq*
        by (*metis* (*mono-tags*, *lifting*) *CollectD Img-point*(*2*) *o-apply setcat.DOWN-UP*)
      ultimately show *?thesis*
        by (*metis* (*mono-tags*, *lifting*) *restrict-ext singletonD*)
    qed
  qed
qed

lemma *set-char*:
assumes *ide a*
shows *set a* = *UP* ‘ *Dom a*
proof
  show *set a* ⊆ *UP* ‘ *Dom a*
  proof
    fix *t*
    assume *t* ∈ *set a*
    from *this* **obtain** *p* **where** *p*: ≪*p* : *unity* → *a*≫ ∧ *t* = *Img p*

using *set-def* **by** *blast*
have $t = (UP\ o\ Map\ p\ o\ DOWN)\ unity$
using *p Img-point(2)* **by** *blast*
moreover have $(Map\ p\ o\ DOWN)\ unity \in Dom\ a$
using *p arr-char in-hom-char Dom-terminal terminal-unity*
by (*metis* (*mono-tags*, *lifting*) *IntD2 Pi-split-insert-domain o-apply*)
ultimately show $t \in UP\ `\ Dom\ a$ **by** *simp*
qed
show $UP\ `\ Dom\ a \subseteq set\ a$
proof
fix t
assume $t \in UP\ `\ Dom\ a$
from *this* **obtain** x **where** x: $x \in Dom\ a \wedge t = UP\ x$ **by** *blast*
let $?p = MkElem\ (UP\ x)\ a$
have p: $?p \in hom\ unity\ a$
using *assms x MkElem-in-hom* [*of dom a*] *ideD*(*1−2*) **by** *force*
moreover have $Img\ ?p = t$
using *p x DOWN-UP*
by (*metis* (*no-types*, *lifting*) *Dom.simps*(*1*) *Map.simps*(*1*) *image-empty*
image-insert image-restrict-eq setcat.Img-def UP-def)
ultimately show $t \in set\ a$ **using** *set-def* **by** *blast*
qed
qed

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

qed

lemma *arr-eqI′*:
assumes *par f f′* **and** $\bigwedge t.\ \ll t : unity \to dom\ f\gg \Longrightarrow f \cdot t = f' \cdot t$
shows $f = f'$
proof (*intro arr-eqI*)
  show *arr f* **using** *assms* **by** *simp*
  show *arr f′* **using** *assms* **by** *simp*
  show $Dom\ f = Dom\ f'$
    using *assms* **by** (*metis* (*mono-tags*, *lifting*) *Dom-dom*)
  show $Cod\ f = Cod\ f'$
    using *assms* **by** (*metis* (*mono-tags*, *lifting*) *Cod-cod*)
  show $Map\ f = Map\ f'$
  proof
    have *1*: $\bigwedge x.\ x \in Dom\ f \Longrightarrow \ll MkElem\ (UP\ x)\ (dom\ f) : unity \to dom\ f\gg$
      using *MkElem-in-hom* **by** (*metis* (*mono-tags*, *lifting*) *assms*(*1*))
    fix *x*
    show $Map\ f\ x = Map\ f'\ x$
      using *assms 1* ‹$Dom\ f = Dom\ f'$› **by** (*simp add*: *Map-via-comp*)
  qed
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-category Comp*
proof
  show $\exists img :: 'a\ setcat.arr \Rightarrow 'a\ setcat.arr.$ *set-category-given-img Comp img*
  proof
    show *set-category-given-img* (*Comp* :: *′a setcat.arr comp*) *Img*
    proof
      show $Univ \neq \{\}$ **using** *terminal-char* **by** *blast*
      fix *a* :: *′a setcat.arr*
      assume *a*: *ide a*
      show $Img \in hom\ unity\ a \to Univ$ **using** *Img-point terminal-unity* **by** *blast*
      show *inj-on Img* (*hom unity a*) **using** *a inj-Img terminal-unity* **by** *blast*
      next
      fix *t* :: *′a setcat.arr*
      assume *t*: *terminal t*
      show $t \in Img$ ‘ *hom unity t*
      proof −
        have $t \in set\ t$
          using *t set-char* [*of t*]
          by (*metis* (*mono-tags*, *lifting*) *Dom.simps*(*1*) *image-insert insertI1 UP-def*
            terminal-char terminal-def)
        thus *?thesis*
          using *t set-def* [*of t*] **by** *simp*
      qed
      next

fix A :: $'a$ *setcat.arr set*
assume A: $A \subseteq$ *Univ*
show $\exists\, a.$ *ide* $a \wedge$ *set* $a = A$
proof
  let $?a$ = *MkArr* (*DOWN* ‘ A) (*DOWN* ‘ A) ($\lambda x \in$ (*DOWN* ‘ A). x)
  show *ide* $?a \wedge$ *set* $?a = A$
  proof
    show *1*: *ide* $?a$
      using *ide-char* [*of* $?a$] **by** *simp*
    show *set* $?a = A$
    proof −
      have *2*: $\bigwedge x.\ x \in A \Longrightarrow x =$ *UP* (*DOWN* x)
        using A *UP-DOWN* **by** *force*
      hence *UP* ‘ *DOWN* ‘ $A = A$
        using A *UP-DOWN* **by** *auto*
      thus *?thesis*
        using *1* A *set-char* [*of* $?a$] **by** *simp*
    qed
  qed
qed
next
fix a b :: $'a$ *setcat.arr*
assume a: *ide* a **and** b: *ide* b **and** ab: *set* a = *set* b
show $a = b$
  using a b ab *set-char inj-UP inj-image-eq-iff dom-char in-homE ide-in-hom*
  by (*metis* (*mono-tags*, *lifting*))
next
fix $f\, f'$ :: $'a$ *setcat.arr*
assume *par*: *par* $f\, f'$ **and** *ff'*: $\bigwedge x.$ «x : *unity* → *dom* f» $\Longrightarrow f \cdot x = f' \cdot x$
show $f = f'$ **using** *par ff' arr-eqI'* **by** *blast*
next
fix a b :: $'a$ *setcat.arr* **and** F :: $'a$ *setcat.arr* ⇒ $'a$ *setcat.arr*
assume a: *ide* a **and** b: *ide* b **and** F: $F \in$ *hom unity* a → *hom unity* b
show $\exists f.$ «f : $a \rightarrow b$» $\wedge$ ($\forall x.$ «x : *unity* → *dom* f» $\longrightarrow f \cdot x = F\, x$)
proof
  let $?f$ = *MkArr* (*Dom* a) (*Dom* b) ($\lambda x \in$ *Dom* a. *Map* (F (*MkElem* (*UP* x) a)) U)
  have *1*: «$?f$: $a \rightarrow b$»
  proof −
    have ($\lambda x \in$ *Dom* a. *Map* (F (*MkElem* (*UP* x) a)) U)
            $\in$ *extensional* (*Dom* a) $\cap$ (*Dom* a → *Dom* b)
    proof
      show ($\lambda x \in$ *Dom* a. *Map* (F (*MkElem* (*UP* x) a)) U) $\in$ *extensional* (*Dom* a)
        using a F **by** *simp*
      show ($\lambda x \in$ *Dom* a. *Map* (F (*MkElem* (*UP* x) a)) U) $\in$ *Dom* a → *Dom* b
      proof
        fix x
        assume x: $x \in$ *Dom* a
        have *MkElem* (*UP* x) $a \in$ *hom unity* a
          using x a *MkElem-in-hom* [*of* a x] *ide-char ideD*(*1*−*2*) **by** *force*

```
      hence 1: F (MkElem (UP x) a) ∈ hom unity b
        using F by auto
      moreover have Dom (F (MkElem (UP x) a)) = {U}
        using 1 MkElem-Img
        by (metis (mono-tags, lifting) Dom.simps(1))
      moreover have Cod (F (MkElem (UP x) a)) = Dom b
        using 1 by (metis (mono-tags, lifting) CollectD in-hom-char)
      ultimately have Map (F (MkElem (UP x) a)) ∈ {U} → Dom b
        using arr-char [of F (MkElem (UP x) a)] by blast
      thus Map (F (MkElem (UP x) a)) U ∈ Dom b by blast
    qed
  qed
  hence «?f : MkIde (Dom a) → MkIde (Dom b)»
    using a b MkArr-in-hom by blast
  thus ?thesis
    using a b by simp
qed
moreover have ⋀x. «x : unity → dom ?f» ⟹ ?f · x = F x
proof -
  fix x
  assume x: «x : unity → dom ?f»
  have 2: x = MkElem (Img x) a
    using a x 1 MkElem-Img [of x a]
    by (metis (mono-tags, lifting) in-homE mem-Collect-eq)
  moreover have 5: Dom x = {U} ∧ Cod x = Dom a ∧
                  Map x = (λ- ∈ {U}. DOWN (Img x))
    using x 2
    by (metis (no-types, lifting) Cod.simps(1) Dom.simps(1) Map.simps(1))
  moreover have Cod ?f = Dom b using 1 by simp
  ultimately have
      3: ?f · x =
           MkArr {U} (Dom b) (compose {U} (Map ?f) (λ- ∈ {U}. DOWN (Img x)))
    using 1 x comp-char [of ?f MkElem (Img x) a]
    by (metis (mono-tags, lifting) in-homE seqI)
  have 4: compose {U} (Map ?f) (λ- ∈ {U}. DOWN (Img x)) = Map (F x)
  proof
    fix y
    have y ∉ {U} ⟹
            compose {U} (Map ?f) (λ- ∈ {U}. DOWN (Img x)) y = Map (F x) y
    proof -
      assume y: y ∉ {U}
      have compose {U} (Map ?f) (λ- ∈ {U}. DOWN (Img x)) y = undefined
        using y compose-def extensional-arb by simp
      also have ... = Map (F x) y
      proof -
        have 5: F x ∈ hom unity b using x F 1 by fastforce
        hence Dom (F x) = {U}
          by (metis (mono-tags, lifting) 2 CollectD Dom.simps(1) in-hom-char x)
        thus ?thesis
```

using *x y F 5 arr-char* [*of F x*] *extensional-arb* [*of Map* (*F x*) {*U*} *y*]
by (*metis* (*mono-tags*, *lifting*) *CollectD Int-iff in-hom-char*)
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have
$y \in \{U\} \Longrightarrow$
compose $\{U\}$ (*Map ?f*) (λ- $\in \{U\}$. *DOWN* (*Img x*)) y = *Map* (*F x*) y
proof −
assume y: $y \in \{U\}$
have *compose* $\{U\}$ (*Map ?f*) (λ- $\in \{U\}$. *DOWN* (*Img x*)) y =
Map ?f (*DOWN* (*Img x*))
using *y* **by** (*simp add*: *compose-eq restrict-apply′*)
also have ... = (λx. *Map* (*F* (*MkElem* (*UP x*) *a*)) *U*) (*DOWN* (*Img x*))
proof −
have *DOWN* (*Img x*) $\in$ *Dom a*
using *x y a 5 arr-char in-homE restrict-apply*
by (*metis* (*mono-tags*, *lifting*) *IntD2 PiE*)
thus *?thesis*
using *restrict-apply* **by** *simp*
qed
also have ... = *Map* (*F x*) *y*
using *x y 1 2 MkElem-Img* [*of x a*] **by** *simp*
finally show
compose $\{U\}$ (*Map ?f*) (λ- $\in \{U\}$. *DOWN* (*Img x*)) y = *Map* (*F x*) y
by *auto*
qed
ultimately show
compose $\{U\}$ (*Map ?f*) (λ- $\in \{U\}$. *DOWN* (*Img x*)) y = *Map* (*F x*) y
by *auto*
qed
show *?f* $\cdot$ *x* = *F x*
proof (*intro arr-eqI*)
have *5*: *?f* $\cdot$ *x* $\in$ *hom unity b* **using** *1 x* **by** *blast*
have *6*: *F x* $\in$ *hom unity b*
using *x F 1*
by (*metis* (*mono-tags*, *lifting*) *PiE in-homE mem-Collect-eq*)
show *arr* (*Comp ?f x*) **using** *5* **by** *blast*
show *arr* (*F x*) **using** *6* **by** *blast*
show *Dom* (*Comp ?f x*) = *Dom* (*F x*)
using *5 6* **by** (*metis* (*mono-tags*, *lifting*) *CollectD in-hom-char*)
show *Cod* (*Comp ?f x*) = *Cod* (*F x*)
using *5 6* **by** (*metis* (*mono-tags*, *lifting*) *CollectD in-hom-char*)
show *Map* (*Comp ?f x*) = *Map* (*F x*)
using *3 4* **by** *simp*
qed
qed
thus «*?f* : *a* → *b*» ∧ (∀ *x*. «*x* : *unity* → *dom ?f*» ⟶ *Comp ?f x* = *F x*)
using *1* **by** *blast*

      qed
    qed
  qed
qed

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

corollary *is-concrete-set-category*:
shows *concrete-set-category Comp Univ UP*
proof −
  interpret *S*: *set-category Comp* **using** *is-set-category* **by** *auto*
  show *?thesis*
  proof
    show *1*: $UP \in Univ \rightarrow S.Univ$
      using *UP-def terminal-char* **by** *force*
    show *inj-on UP Univ*
      by (*metis* (*mono-tags*, *lifting*) *injD inj-UP inj-onI*)
  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** $\varphi :: 's \Rightarrow 'a$
assumes *set-category S*
and *bij-betw* φ (*Collect* (*category.terminal S*)) *UNIV*
shows $\exists\,\Phi.$ *invertible-functor S* (*Comp* :: *'a setcat.arr comp*) Φ
      $\wedge$ ($\forall\, m.$ *set-category.incl S m* $\longrightarrow$ *set-category.incl Comp* (Φ *m*))
proof −
  interpret *S*: *set-category S* **using** *assms* **by** *auto*
  let $?\psi$ = *inv-into S.Univ* φ
  have *bij-betw* (*UP o* φ) *S.Univ* (*Collect terminal*)
  proof (*intro bij-betwI*)
    show *UP o* $\varphi \in S.Univ \rightarrow$ *Collect terminal*
      using *assms*(*2*) *UP-mapsto* **by** *auto*
    show $?\psi$ *o DOWN* $\in$ *Collect terminal* $\rightarrow S.Univ$
    proof
      fix x :: *'a setcat.arr*
      assume x: $x \in Univ$
      show (*inv-into S.Univ* $\varphi \circ$ *DOWN*) $x \in S.Univ$
        using *x assms*(*2*) *bij-betw-def comp-apply inv-into-into*
        by (*metis UNIV-I*)
    qed
    fix t
    assume $t \in S.Univ$
    thus ($?\psi$ *o DOWN*) ((*UP o* φ) t) = t
      using *assms*(*2*) *bij-betw-inv-into-left*

by (*metis comp-apply DOWN-UP*)
next
fix t' :: $'a$ *setcat.arr*
assume $t' \in$ *Collect terminal*
thus (*UP o* φ) ((*?*ψ *o DOWN*) t') $= t'$
using *assms*(*2*) **by** (*simp add*: *bij-betw-def f-inv-into-f*)
qed
thus *?thesis*
using *assms*(*1*) *set-category-is-categorical* [*of S Comp UP o* φ] *is-set-category*
by *auto*
qed

end

The following context defines the entities that are intended to be exported from this theory. The idea is to avoid exposing as little detail about the construction used in the *setcat* locale as possible, so that proofs using the result of that construction will depend only on facts proved from axioms in the *set-category* locale and not on concrete details from the construction of the interpretation.

context
begin

interpretation *S*: *setcat* **.**

definition *comp*
where *comp* $\equiv$ *S.Comp*

interpretation *set-category comp*
unfolding *comp-def* **using** *S.is-set-category* **by** *simp*

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

definition *DOWN*
where *DOWN* $=$ *S.DOWN*

definition *UP*
where *UP* $=$ *S.UP*

lemma *UP-mapsto*:
shows *UP* $\in$ *UNIV* $\rightarrow$ *Univ*
using *S.UP-mapsto*
by (*simp add*: *UP-def comp-def*)

lemma *DOWN-mapsto*:
shows *DOWN* $\in$ *Univ* $\rightarrow$ *UNIV*
by *auto*

lemma *DOWN-UP* [*simp*]:
shows *DOWN* (*UP* *x*) = *x*
  by (*simp add*: *DOWN-def UP-def*)

lemma *UP-DOWN* [*simp*]:
assumes $t \in$ *Univ*
shows *UP* (*DOWN* *t*) = *t*
  using *assms DOWN-def UP-def*
  by (*simp add*: *DOWN-def UP-def comp-def*)

lemma *inj-UP*:
shows *inj UP*
  by (*metis DOWN-UP injI*)

lemma *bij-UP*:
shows *bij-betw UP UNIV Univ*
  by (*metis S.bij-UP UP-def comp-def*)

end

end

Chapter 12

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* $\equiv$ (*if Arr f then* (*C1.cod* (*fst f*), *C2.cod* (*snd f*)) *else Null*)

definition *comp* :: ('*a1*, '*a2*) *arr* $\Rightarrow$ ('*a1*, '*a2*) *arr* $\Rightarrow$ ('*a1*, '*a2*) *arr*
where *comp g f* = (*if Arr f* $\wedge$ *Arr g* $\wedge$ *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-magma comp*
proof
show $\exists!n. \forall f. n \cdot f = n \wedge f \cdot n = n$
proof
let *?P* = $\lambda n. \forall f. n \cdot f = n \wedge f \cdot n = n$
show *1*: *?P Null* **using** *comp-def not-Arr-Null* **by** *metis*
thus $\bigwedge n. \forall f. n \cdot f = n \wedge f \cdot n = n \Longrightarrow n = Null$ **by** *metis*
qed
qed

notation *in-hom* («- : - → -»)

lemma *null-char* [*simp*]:
shows *null* = *Null*
proof –
let *?P* = $\lambda n. \forall f. n \cdot f = n \wedge f \cdot n = n$
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* $\neq$ {} $\longleftrightarrow$ *Arr f*
proof
show *domains f* $\neq$ {} $\Longrightarrow$ *Arr f*
unfolding *domains-def comp-def null-char* **by** (*auto*; *metis*)
assume *f*: *Arr f*
show *domains f* $\neq$ {}

  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* [*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* [*intro*]:
assumes *C1.seq g1 f1* ∧ *C2.seq g2 f2*
shows *seq* (*g1*, *g2*) (*f1*, *f2*)
  using *assms comp-def* **by** *auto*

lemma *seqE* [*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* $\longleftrightarrow$ *C1.seq* (*fst g*) (*fst f*) $\wedge$ *C2.seq* (*snd g*) (*snd f*)
  using *comp-def* **by** *auto*

lemma *Dom-comp*:
assumes *seq g f*
shows *Dom* (*g* $\cdot$ *f*) = *Dom f*
  using *assms comp-def*
  apply (*cases C1.arr* (*fst g*); *cases C1.arr* (*fst f*);
        cases C2.arr (*snd f*); *cases C2.arr* (*snd g*); *simp-all*)
  by *auto*

lemma *Cod-comp*:
assumes *seq g f*
shows *Cod* (*g* $\cdot$ *f*) = *Cod g*
  using *assms comp-def*
  apply (*cases C1.arr* (*fst f*); *cases C2.arr* (*snd f*);
        cases C1.arr (*fst g*); *cases C2.arr* (*snd g*); *simp-all*)
  by *auto*

theorem *is-category*:
shows *category comp*
proof
  fix *f*
  show (*domains f* $\neq$ {}) = (*codomains f* $\neq$ {})
    using *has-domain-char has-codomain-char* **by** *simp*
  fix *g*
  show *g* $\cdot$ *f* $\neq$ *null* $\Longrightarrow$ *seq g f*
    using *comp-def seq-char* **by** (*metis C1.seqI C2.seqI Pair-inject null-char*)
  fix *h*
  show *seq h g* $\Longrightarrow$ *seq* (*h* $\cdot$ *g*) *f* $\Longrightarrow$ *seq g f*
    using *comp-def null-char seq-char* **by** (*elim seqE C1.seqE C2.seqE*, *simp*)
  show *seq h* (*g* $\cdot$ *f*) $\Longrightarrow$ *seq g f* $\Longrightarrow$ *seq h g*
    using *comp-def null-char seq-char* **by** (*elim seqE C1.seqE C2.seqE*, *simp*)
  show *seq g f* $\Longrightarrow$ *seq h g* $\Longrightarrow$ *seq* (*h* $\cdot$ *g*) *f*
    using *comp-def null-char seq-char* **by** (*elim seqE C1.seqE C2.seqE*, *simp*)
  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 C1.seqE C2.seqE*, *simp*)
qed

end

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

context *product-category*
begin

lemma *dom-char*:
shows *dom f* = *Dom f*
proof (*cases Arr f*)
  show $\neg$*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 $\neg$*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* [*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* [*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* [*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* · *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 $C1.inverse\text{-}arrows\ (fst\ f)\ (fst\ g) \wedge C2.inverse\text{-}arrows\ (snd\ f)\ (snd\ g)$
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*: $C1.inverse\text{-}arrows\ (fst\ f)\ g1$ **using** *f* **by** *blast*
obtain *g2* **where** *g2*: $C2.inverse\text{-}arrows\ (snd\ f)\ g2$ **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* **by** *force*
hence $inverse\text{-}arrows\ f\ (g1,\ g2)$
using *f g1 g2 ide-char comp-char* **by** (*intro inverse-arrowsI*, *auto*)
thus *iso f* **by** *auto*
qed

lemma *isoI* [*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 comp-char* **by** *auto*
    show *ide* (*f* · (*C1.inv* (*fst f*), *C2.inv* (*snd f*)))
      using *1 2 ide-char comp-char* **by** *auto*
  qed
  thus *?thesis* **using** *inverse-unique* **by** *auto*
qed

end

end

Chapter 13

NaturalTransformation

theory *NaturalTransformation*
imports *Functor*
begin

13.1 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** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and F :: $'a \Rightarrow {'b}$
and G :: $'a \Rightarrow {'b}$
and τ :: $'a \Rightarrow {'b}$ +
assumes *is-extensional*: $\neg A.arr\ f \Longrightarrow \tau\ f = B.null$

and *preserves-dom* [*iff*]: $A.arr\, f \Longrightarrow B.dom\ (\tau\ f) = F\ (A.dom\, f)$
and *preserves-cod* [*iff*]: $A.arr\, f \Longrightarrow B.cod\ (\tau\ f) = G\ (A.cod\, f)$
and *is-natural-1* [*iff*]: $A.arr\, f \Longrightarrow G\, f \cdot_B \tau\ (A.dom\, f) = \tau\ f$
and *is-natural-2* [*iff*]: $A.arr\, f \Longrightarrow \tau\ (A.cod\, f) \cdot_B F\, f = \tau\ f$
begin

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 is-natural-1 is-natural-2* **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 *is-extensional A.arr-cod-iff-arr B.arr-cod-iff-arr preserves-cod* **by** *force*

lemma *preserves-hom* [*intro*]:
assumes $\ll f : a \rightarrow_A b\gg$
shows $\ll \tau\ f : F\ a \rightarrow_B G\ b\gg$
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 is-natural-1*)

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 is-natural-2*)

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* $A\ B\ \tau$
shows $F = \tau$
proof
interpret τ: *functor* $A\ B\ \tau$ **using** *assms* **by** *auto*
fix f
show $F\, f = \tau\ f$
using *assms*
by (*metis A.dom-cod B.comp-cod-arr F.is-extensional F.preserves-arr F.preserves-cod* τ*.preserves-dom is-extensional is-natural-2 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 F.preserves-arr G.is-extensional G.preserves-arr G.preserves-dom B.cod-dom functor-implies-equals-dom is-extensional is-natural-1 preserves-cod preserves-dom*)
qed

end

13.2 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 *eqI*:
assumes *natural-transformation A B F G* σ **and** *natural-transformation A B F G* σ'
and $\bigwedge a.$ *partial-magma.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*.is-natural-2* σ'*.is-natural-2* σ*.is-extensional* σ'*.is-extensional 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 *is-natural-1* or equivalently by *is-natural-2*.

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** $\cdot_A$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *55*)
and *F* :: *'a* ⇒ *'b*
and *G* :: *'a* ⇒ *'b*
and *t* :: *'a* ⇒ *'b* +

assumes *maps-ide-in-hom* [*intro*]: *A.ide a* $\Longrightarrow$ «*t a* : *F a* $\rightarrow_B$ *G a*»
and *is-natural*: *A.arr f* $\Longrightarrow$ *t* (*A.cod f*) $\cdot_B$ *F f* = *G f* $\cdot_B$ *t* (*A.dom f*)
begin

definition *map*
where *map f* = (*if A.arr f then t* (*A.cod f*) $\cdot_B$ *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.dom-comp B.seqI*
            G.preserves-arr G.preserves-dom B.in-homE maps-ide-in-hom)
   apply (*metis A.ide-dom B.arrI B.cod-comp B.in-homE B.seqI*
            G.preserves-arr G.preserves-cod G.preserves-dom 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* $\tau = \tau$
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 eqI* **by** *blast*
qed

13.3 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 *is-extensional B.comp-arr-dom B.comp-cod-arr*
  by (*unfold-locales*, *simp-all*)
qed

sublocale *functor* ⊆ *natural-transformation A B F F F*
  by (*simp add*: *functor-axioms*)

13.4 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** $\cdot_A$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *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* ⊆ *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

13.5 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 τ *o* σ, 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** $\cdot_A$ *55*)
and *B* :: *'b comp*   (**infixr** $\cdot_B$ *55*)
and *F* :: *'a* ⇒ *'b*
and *G* :: *'a* ⇒ *'b*
and *H* :: *'a* ⇒ *'b*
and σ :: *'a* ⇒ *'b*
and τ :: *'a* ⇒ *'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*.is-natural-2* σ*.naturality* τ*.is-natural-1* 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*.is-natural-1*)

end

sublocale *vertical-composite* ⊆ *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
using *assms* **apply** (*intro eqI*)
apply *auto*[*2*]
apply (*meson functor-is-transformation natural-transformation-def vertical-composite.intro vertical-composite.is-natural-transformation*)
proof −
fix *a* :: *'a*
have *vertical-composite A B F F G F* τ
by (*meson assms functor-is-transformation natural-transformation.axioms*(*1*−*4*) *vertical-composite.intro*)
thus *vertical-composite.map A B F* τ *a* = τ *a*
using *assms natural-transformation.is-extensional natural-transformation.is-natural-2 vertical-composite.map-def*
by *fastforce*
qed

lemma *vcomp-ide-cod* [*simp*]:
assumes *natural-transformation A B F G* τ
shows *vertical-composite.map A B* τ *G* = τ

using *assms* **apply** (*intro eqI*)
apply *auto*[*2*]
apply (*meson functor-is-transformation natural-transformation-def vertical-composite.intro vertical-composite.is-natural-transformation*)
proof −
fix *a* :: *'a*
assume *a*: *partial-magma.ide A a*
interpret *Goτ*: *vertical-composite A B F G G τ G*
by (*meson assms functor-is-transformation natural-transformation.axioms*(*1−4*) *vertical-composite.intro*)
show *vertical-composite.map A B τ G a* = *τ a*
using *assms a natural-transformation.is-extensional natural-transformation.is-natural-1 Goτ.map-simp-ide Goτ.B.comp-cod-arr*
by *simp*
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 eqI*, *auto*)
qed

13.6 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** $\cdot_A$ *55*)

and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and F :: $'a \Rightarrow 'b$
and G :: $'a \Rightarrow 'b$
and τ :: $'a \Rightarrow 'b$ +
assumes *components-are-iso* [*simp*]: $A.ide\ a \Longrightarrow B.iso\ (\tau\ a)$
begin

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 is-natural-2 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* $\subseteq$ *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* $A\ B\ F\ G\ \varphi$
using *assms naturally-isomorphic-def* **by** *blast*
interpret φ: *natural-isomorphism* $A\ B\ F\ G\ \varphi$
using φ **by** *auto*
interpret $\varphi.F$: *full-functor* $A\ B\ F$
using *assms* **by** *auto*
write A (**infixr** $\cdot_A$ *55*)
write B (**infixr** $\cdot_B$ *55*)
write $\varphi.A.in\text{-}hom$ ($\ll$- : - $\rightarrow_A$ -$\gg$)
write $\varphi.B.in\text{-}hom$ ($\ll$- : - $\rightarrow_B$ -$\gg$)
show *full-functor* $A\ B\ G$
proof
fix $a\ a'\ g$

assume a': $\varphi.A.ide\ a'$ **and** a: $\varphi.A.ide\ a$
and g: $\ll g : G\ a' \rightarrow_B G\ a\gg$
show $\exists f. \ll f : a' \rightarrow_A a\gg \wedge G\ f = g$
proof −
let $?g' = \varphi.B.inv\ (\varphi\ a) \cdot_B g \cdot_B \varphi\ a'$
have g': $\ll ?g' : F\ a' \rightarrow_B F\ a\gg$
using *a a' g φ.preserves-hom φ.components-are-iso φ.B.inv-in-hom* **by** *force*
obtain f' **where** f': $\ll f' : a' \rightarrow_A a\gg \wedge F\ f' = ?g'$
using *a a' g' φ.F.is-full [of a a' ?g']* **by** *blast*
moreover have $G\ f' = g$
proof −
have $G\ f' = \varphi\ a \cdot_B ?g' \cdot_B \varphi.B.inv\ (\varphi\ a')$
using *a a' f' φ.naturality [of f'] φ.components-are-iso φ.is-natural-2*
by (*metis φ.A.in-homE φ.B.comp-assoc φ.B.invert-side-of-triangle*(*2*)
φ.preserves-reflects-arr)
also have ... $= (\varphi\ a \cdot_B \varphi.B.inv\ (\varphi\ a)) \cdot_B g \cdot_B \varphi\ a' \cdot_B \varphi.B.inv\ (\varphi\ a')$
using *φ.B.comp-assoc* **by** *auto*
also have ... $= g$
using *a a' g φ.B.comp-arr-dom φ.B.comp-cod-arr φ.B.comp-arr-inv*
φ.B.inv-is-inverse
by *auto*
finally show *?thesis* **by** *blast*
qed
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 $\varphi.F$: *faithful-functor A B F*
using *assms* **by** *auto*
show *faithful-functor A B G*
using *φ.naturality φ.components-are-iso φ.B.iso-is-section φ.B.section-is-mono*
φ.B.monoE φ.F.is-faithful φ.is-natural-1 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** $\cdot_A$ *55*)
and B :: *'b comp* (**infixr** $\cdot_B$ *55*)
and F :: $'a \Rightarrow 'b$
and G :: $'a \Rightarrow 'b$
and τ :: $'a \Rightarrow 'b$
begin

interpretation τ': *transformation-by-components A B G F* ‹λa. *B.inv* (τ *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*.is-natural-2* 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*: *B.iso-inv-iso 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 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** $\cdot_A$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *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.natural-transformation-axioms*
        σσ′.map-simp-ide inv.B.comp-inv-arr inv.inv
    by (*intro 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.natural-transformation-axioms*
        σ′σ.map-simp-ide inv′.inv inv.B.comp-inv-arr
    by (*intro 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.natural-transformation-axioms τ′.inverts-components*
        τ.B.comp-inv-arr ττ′.map-simp-ide
    by (*intro 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.natural-transformation-axioms* τ'*.inverts-components* $\tau'\tau$*.map-simp-ide* τ*.B.comp-arr-inv*
    by (*intro 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

13.7 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*.is-extensional* τ*.is-extensional*
    apply (*unfold-locales*, *auto*)
     apply (*metis* σ*.is-natural-1* σ*.preserves-reflects-arr* τ*.preserves-comp-1*)
    by (*metis* σ*.is-natural-2* 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.is-extensional* **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.is-extensional* **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.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.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.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.is-extensional* **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 $K \circ$ (*vertical-composite.map* $A\ B\ \tau\ \tau'$) = *vertical-composite.map* $A\ C$ ($K \circ \tau$) ($K \circ \tau'$)
proof −
interpret K: *functor* $B\ C\ K$ **using** *assms*(*1*) **by** *auto*
interpret τ: *natural-transformation* $A\ B\ F\ G\ \tau$ **using** *assms*(*2*) **by** *auto*
interpret τ': *natural-transformation* $A\ B\ G\ H\ \tau'$ **using** *assms*(*3*) **by** *auto*
interpret $\tau'\tau$: *vertical-composite* $A\ B\ F\ G\ H\ \tau\ \tau'$ **..**
interpret $K o\tau$: *natural-transformation* $A\ C$ ‹K o F› ‹K o G› ‹K o τ›
using τ.*natural-transformation-axioms* K.*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.*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.*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* $K o\tau'\tau$.*is-extensional* $K o\tau'$-$K o\tau$.*map-simp-1* $\tau'\tau$.*map-simp-1*
by *auto*
qed

The interchange law for horizontal and vertical composition.

lemma *interchange*:
assumes *natural-transformation* $B\ C\ F\ G\ \tau$ **and** *natural-transformation* $B\ C\ G\ H\ \nu$
and *natural-transformation* $C\ D\ K\ L\ \sigma$ **and** *natural-transformation* $C\ D\ L\ M\ \mu$
shows *vertical-composite.map* $C\ D\ \sigma\ \mu \circ$ *vertical-composite.map* $B\ C\ \tau\ \nu$ =
vertical-composite.map $B\ D$ ($\sigma \circ \tau$) ($\mu \circ \nu$)
proof −
interpret τ: *natural-transformation* $B\ C\ F\ G\ \tau$
using *assms*(*1*) **by** *auto*
interpret ν: *natural-transformation* $B\ C\ G\ H\ \nu$
using *assms*(*2*) **by** *auto*
interpret σ: *natural-transformation* $C\ D\ K\ L\ \sigma$
using *assms*(*3*) **by** *auto*
interpret μ: *natural-transformation* $C\ D\ L\ M\ \mu$
using *assms*(*4*) **by** *auto*
interpret $\nu\tau$: *vertical-composite* $B\ C\ F\ G\ H\ \tau\ \nu$ **..**
interpret $\mu\sigma$: *vertical-composite* $C\ D\ K\ L\ M\ \sigma\ \mu$ **..**
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 μσoντ: natural-transformation B D ‹K o F› ‹M o H› ‹μσ.map o ντ.map›
  using μσ.natural-transformation-axioms ντ.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 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.is-natural-1 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 14

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** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp*   (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *'b comp*   (**infixr** $\cdot_B$ *55*)
and *F* :: *'a1* $*$ *'a2* $\Rightarrow$ *'b*
begin

  notation *A1.in-hom*   («- : - $\rightarrow_{A1}$ -»)
  notation *A2.in-hom*   («- : - $\rightarrow_{A2}$ -»)

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** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp*   (**infixr** $\cdot_{A2}$ *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 B1xB2.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
    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 ..
```

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** $\cdot_{A2xA1}$ *55*)
notation *A1xA2.in-hom* («- : - $\rightarrow_{A1xA2}$ -»)
notation *A2xA1.in-hom* («- : - $\rightarrow_{A2xA1}$ -»)

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* ≡ (λ*f*. *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* = (λ*f*. *F* (*snd f*, *fst f*))
    using *is-extensional SF.is-extensional 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 *is-extensional*
apply *auto*[*4*]
by (*metis A1.ideD*(*1*) *A1.comp-ide-self A1xA2.comp-simp A1xA2.seq-char fst-conv preserves-comp-2 snd-conv*)

lemma *fixing-ide-gives-functor-2*:
assumes *A2.ide a2*
shows *functor A1 B* (*λf1. F* (*f1, a2*))
using *assms*
apply (*unfold-locales*)
using *is-extensional*
apply *auto*[*4*]
by (*metis A1xA2.comp-simp A1xA2.seq-char A2.ideD*(*1*) *A2.comp-ide-self fst-conv 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* (*λf2. F* (*A1.dom f1, f2*)) (*λf2. F* (*A1.cod f1, f2*)) (*λf2. F* (*f1, f2*))
proof −
let *?Fdom* = *λf2. F* (*A1.dom f1, f2*)
interpret *Fdom*: *functor A2 B ?Fdom* **using** *assms fixing-ide-gives-functor-1* **by** *auto*
let *?Fcod* = *λf2. F* (*A1.cod f1, f2*)
interpret *Fcod*: *functor A2 B ?Fcod* **using** *assms fixing-ide-gives-functor-1* **by** *auto*
let *?τ* = *λf2. F* (*f1, f2*)
show *natural-transformation A2 B ?Fdom ?Fcod ?τ*
using *assms*
apply *unfold-locales*
using *is-extensional*
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* (*λf1. F* (*f1, A2.dom f2*)) (*λf1. F* (*f1, A2.cod f2*)) (*λf1. 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* (*λf1. sym* (*A2.dom f2, f1*)) (*λf1. sym* (*A2.cod f2, f1*)) (*λf1. 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′* $\cdot_{A1}$ *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′* $\cdot_{A1}$ *f1*, *f2*)) = τ′oτ.*map*
proof
fix *f2*
have ¬*A2.arr f2* ⟹ *F* (*f1′* $\cdot_{A1}$ *f1*, *f2*) = τ′oτ.*map f2*
using τ′oτ.*is-extensional is-extensional* **by** *simp*
moreover have *A2.arr f2* ⟹ *F* (*f1′* $\cdot_{A1}$ *f1*, *f2*) = τ′oτ.*map f2*
proof −
assume *f2*: *A2.arr f2*
have *F* (*f1′* $\cdot_{A1}$ *f1*, *f2*) = *B* (*F* (*f1′*, *f2*)) (*F* (*f1*, *A2.dom f2*))
using *assms f2 preserves-comp A1xA2.arr-char A1xA2.comp-char A2.comp-arr-dom*
by (*metis fst-conv snd-conv*)
also have ... = τ′oτ.*map f2*
using *f2* τ′oτ.*map-simp-2* **by** *simp*
finally show *F* (*f1′* $\cdot_{A1}$ *f1*, *f2*) = τ′oτ.*map f2* **by** *auto*
qed
ultimately show *F* (*f1′* $\cdot_{A1}$ *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′* $\cdot_{A2}$ *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′* $\cdot_{A2}$ *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 is-extensional is-natural-1 is-natural-2
      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* ‹λ*f1*. *G* (*f1*, *a2*)›
  using *assms G.fixing-ide-gives-functor-2* **by** *simp*
show *?thesis*
  using *assms is-extensional is-natural-1 is-natural-2*
  apply (*unfold-locales*, *auto*)
   apply (*metis A2.ide-char*)
  by (*metis A2.ide-char*)
qed

end

end

Chapter 15

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.

15.1 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** $\cdot_A$ *55*)
and *B* :: *'b comp*   (**infixr** $\cdot_B$ *55*)
begin

  notation *A.in-hom*   («- : - $\rightarrow_A$ -»)
  notation *B.in-hom*   («- : - $\rightarrow_B$ -»)

  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

abbreviation *comp* (**infixr** $\cdot$ *55*)
where *comp* $\equiv$ *COMP*
notation *in-hom* («- : - → -»)

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* $\Longrightarrow$ *natural-transformation* A B (*Dom* f) (*Cod* f) (*Map* f) $\Longrightarrow$ T
shows T
using *assms arr-char null-char* **by** *simp*

lemma *arr-MkArr* [*iff*]:
shows *arr* (*MkArr* F G τ) $\longleftrightarrow$ *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 null-char* **by** *fastforce*

end

15.2 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
    proof −
      have Map (t′ · t) = t′ot.map
        using assms(1) seq-char t′ot.natural-transformation-axioms by simp
      thus ?thesis
        using assms(2) t′ot.map-simp-2 t′.preserves-comp-2 B.comp-assoc by auto
    qed
  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 [intro]:
  assumes arr (MkArr F G τ)
  and F = F′ and G = G′ and τ = τ′
  shows MkArr F G τ = MkArr F′ G′ τ′
    using assms arr-eqI by simp

  lemma MkArr-eqI′ [intro]:
  assumes arr (MkArr F G τ) and τ = τ′
  shows MkArr F G τ = MkArr F G τ′
    using assms arr-eqI by simp

  lemma iso-char [iff]:
  shows iso t ⟷ t ≠ null ∧ 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 $t \neq null$ **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 $\tau'o\tau$: *vertical-composite A B* ‹*Dom t*› ‹*Cod t*› ‹*Dom t*› ‹*Map t*› ‹*Map t'*› **..**
  interpret $\tau o\tau'$: *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*: $\tau'o\tau.map = Dom\ t \wedge \tau o\tau'.map = Cod\ t$
        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* $\tau o\tau'$*.map-simp-ide* $\tau'o\tau$*.map-simp-ide* τ*.F.preserves-ide* τ*.G.preserves-ide*
        by *auto*
    qed
  qed
qed
next
assume *t*: $t \neq null \wedge$ *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* $\wedge$
        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* $\cdot$ *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* $\cdot$ *MkArr* (*Cod t*) (*Dom t*) τ'*.map*)
      using *t 1 2*
      by (*metis Map.simps*(*1*) τ'*.natural-transformation-axioms arr-MkArr comp-char*
        dom-MkArr dom-comp ide-char' ide-compE)

```
      qed
    qed
  qed

end
```

15.3 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*)
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* ($Fg' \cdot_{[A,B]xA} Fg$) = *map* $Fg' \cdot_B$ *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* **..**

15.4 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** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp* (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_{B}$ *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** $\cdot_{A1xA2}$ *55*)
notation *A2-B.comp* (**infixr** $\cdot_{[A2,B]}$ *55*)
notation *A2-BxA2.comp* (**infixr** $\cdot_{[A2,B]xA2}$ *55*)
notation *A1xA2.in-hom* ($\ll$- : - $\rightarrow_{A1xA2}$ -$\gg$)
notation *A2-B.in-hom* ($\ll$- : - $\rightarrow_{[A2,B]}$ -$\gg$)
notation *A2-BxA2.in-hom* ($\ll$- : - $\rightarrow_{[A2,B]xA2}$ -$\gg$)

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 (*λf2. F* (*A1.dom f1*, *f2*)) (*λf2. G* (*A1.cod f1*, *f2*))
(*λf2. 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* (*λf2. F* (*A1.dom f1*, *f2*)) (*λf2. G* (*A1.cod f1*, *f2*))
(*λf2. 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 $\ll$*curry F G τ f1* : *curry F F F* (*A1.dom f1*) $\rightarrow_{[A2,B]}$ *curry G G G* (*A1.cod f1*)$\gg$
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 ‹λf2. F (A1.dom f1, f2)›
      using f1 τ.F.is-extensional apply (unfold-locales, simp-all)
      by (metis A1xA2.comp-char A1.arr-dom-iff-arr A1.comp-arr-dom A1.dom-dom
               A1xA2.seqI τ.F.preserves-comp-2 fst-conv snd-conv)
    interpret G-cod-f1: functor A2 B ‹λf2. G (A1.cod f1, f2)›
      using f1 τ.G.is-extensional A1.arr-cod-iff-arr
      apply (unfold-locales, simp-all)
      using A1xA2.comp-char A1.arr-cod-iff-arr A1.comp-cod-arr
      by (metis A1.cod-cod A1xA2.seqI τ.G.preserves-comp-2 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.is-extensional 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.dom (curry F G τ f1) = curry F F F (A1.dom f1)
    using assms f1 curry-in-hom by blast
  show A2-B.cod (curry F G τ f1) = curry G G G (A1.cod 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 A2-B.arr (A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2))
                          G-f1oτ-dom-f1.map)
        using G-f1oτ-dom-f1.natural-transformation-axioms by blast
      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.is-extensional τ.is-extensional 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* ‹λ*f2*. *F* (*A1*.*dom f1*, *f2*)›
                  ‹λ*f2*. *G* (*A1*.*cod f1*, *f2*)› ‹λ*f2*. 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* = (λ*f2*. τ (*f1*, *f2*)) *f2*
        using *f1 f2 G-f1oτ-dom-f1*.*map-simp-2 B*.*comp-assoc* τ.*is-natural-1*
        by *fastforce*
    qed
    ultimately show *G-f1oτ-dom-f1*.*map f2* = (λ*f2*. 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*) $\cdot_{[A2,B]}$ *curry F F F f1* = *curry F G* τ *f1*
proof −
 interpret *τ-cod-f1*: *natural-transformation A2 B* ‹λ*f2*. *F* (*A1*.*cod f1*, *f2*)›
                  ‹λ*f2*. *G* (*A1*.*cod f1*, *f2*)› ‹λ*f2*. 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*
                  ‹λ*f2*. *F* (*A1*.*dom f1*, *f2*)› ‹λ*f2*. *F* (*A1*.*cod f1*, *f2*)› ‹λ*f2*. *F* (*f1*, *f2*)›
   using *f1* τ.*F*.*fixing-arr-gives-natural-transformation-1* **by** *simp*
 interpret *τ-cod-f1oF-f1*: *vertical-composite A2 B*
                  ‹λ*f2*. *F* (*A1*.*dom f1*, *f2*)› ‹λ*f2*. *F* (*A1*.*cod f1*, *f2*)›
                  ‹λ*f2*. *G* (*A1*.*cod f1*, *f2*)›
                  ‹λ*f2*. *F* (*f1*, *f2*)› ‹λ*f2*. τ (*A1*.*cod f1*, *f2*)› **..**
 have *curry F G* τ (*A1*.*cod f1*) $\cdot_{[A2,B]}$ *curry F F F f1*
    = *A2-B*.*MkArr* (λ*f2*. *F* (*A1*.*dom f1*, *f2*)) (λ*f2*. *G* (*A1*.*cod f1*, *f2*)) *τ-cod-f1oF-f1*.*map*
 proof −
  have
     curry F F F f1 =
       A2-B.*MkArr* (λ*f2*. *F* (*A1*.*dom f1*, *f2*)) (λ*f2*. *F* (*A1*.*cod f1*, *f2*))
             (λ*f2*. *F* (*f1*, *f2*)) ∧
      «*curry F F F f1* : *curry F F F* (*A1*.*dom f1*) $\rightarrow_{[A2,B]}$ *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* (λ*f2*. *F* (*A1*.*dom f1*, *f2*)) (λ*f2*. *G* (*A1*.*dom f1*, *f2*))
             (λ*f2*. τ (*A1*.*dom f1*, *f2*)) ∧
       «*curry F G* τ (*A1*.*cod f1*) :
         curry F F F (*A1*.*cod f1*) $\rightarrow_{[A2,B]}$ *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* (λ*f2*. *F* (*A1*.*dom f1*, *f2*)) (λ*f2*. *G* (*A1*.*cod f1*, *f2*))
                  (λ*f2*. 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 A2-B.arr (A2-B.MkArr (λf2. F (A1.dom f1, f2)) (λf2. G (A1.cod f1, f2))
                             τ-cod-f1oF-f1.map)
          using τ-cod-f1oF-f1.natural-transformation-axioms by blast
        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.is-extensional τ-cod-f1oF-f1.is-extensional)
          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.is-natural-2
              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
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.dom (uncurry τ f) = uncurry F (A1xA2.dom f)
      using f uncurry-def by simp
    show B.cod (uncurry τ f) = uncurry G (A1xA2.cod 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.is-natural-1 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*) $\cdot_B$ *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*.is-natural-2*)

    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*.is-extensional* **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.is-extensional* **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.is-extensional* **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.is-extensional* **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.is-extensional 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.is-extensional
          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* $(\ll\text{-} : \text{-} \rightarrow_{[A2,B]} \text{-}\gg)$

definition *map*
where *map* ≡ *curry F F F*

lemma *map-simp* [*simp*]:
assumes *A1.arr f1*
shows *map f1* =
    A2-B.MkArr (λ*f2*. *F* (*A1.dom f1*, *f2*)) (λ*f2*. *F* (*A1.cod f1*, *f2*)) (λ*f2*. *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** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp* (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_{B}$ *55*)
and *F* :: *'a1* ∗ *'a2* ⇒ *'b*
begin

notation *A1xA2.comp* (**infixr** $\cdot_{A1xA2}$ *55*)
notation *A1-B.comp* (**infixr** $\cdot_{[A1,B]}$ *55*)
notation *A1xA2.in-hom* $(\ll\text{-} : \text{-} \rightarrow_{A1xA2} \text{-}\gg)$
notation *A1-B.in-hom* $(\ll\text{-} : \text{-} \rightarrow_{[A1,B]} \text{-}\gg)$

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 16

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 φ $(b,\ a)$ 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*.

16.1 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 +

Cop: *dual-category C* +
CopxC: *product-category Cop.comp C* +
S: *set-category S*
for C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$ +
assumes *maps-arr-to-Univ*: $C.arr\ f \Longrightarrow \varphi\ (C.dom\ f,\ C.cod\ f)\ f \in S.Univ$
and *local-inj*: $[\![\ C.ide\ b;\ C.ide\ a\ ]\!] \Longrightarrow$ *inj-on* $(\varphi\ (b,\ a))\ (C.hom\ b\ a)$
begin

notation *S.in-hom* («- : - $\rightarrow_S$ -»)
notation *CopxC.comp* (**infixr** $\odot$ *55*)
notation *CopxC.in-hom* («- : - $\rightleftharpoons$ -»)

definition *set*
where *set ba* $\equiv \varphi$ (*fst ba*, *snd ba*) ‘ *C.hom* (*fst ba*) (*snd ba*)

lemma *set-subset-Univ*:
assumes *C.ide b* **and** *C.ide a*
shows *set* $(b,\ a) \subseteq$ *S.Univ*
using *assms set-def maps-arr-to-Univ CopxC.ide-char* **by** *auto*

definition ψ :: $'c * 'c \Rightarrow 's \Rightarrow 'c$
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 $\varphi\ (b,\ a) \in C.hom\ b\ a \rightarrow set\ (b,\ a)$
using *assms set-def maps-arr-to-Univ* **by** *auto*

lemma ψ*-mapsto*:
assumes *C.ide b* **and** *C.ide a*
shows $\psi\ (b,\ a) \in set\ (b,\ a) \rightarrow C.hom\ b\ a$
using *assms set-def* ψ*-def local-inj* **by** *auto*

lemma ψ-φ [*simp*]:
assumes «f : $b \rightarrow a$»
shows $\psi\ (b,\ a)\ (\varphi\ (b,\ a)\ f) = f$
using *assms local-inj* [*of b a*] ψ*-def* **by** *fastforce*

lemma φ-ψ [*simp*]:
assumes *C.ide b* **and** *C.ide a*
and $x \in set\ (b,\ a)$
shows $\varphi\ (b,\ a)\ (\psi\ (b,\ a)\ x) = x$
using *assms set-def local-inj* ψ*-def* **by** *auto*

lemma ψ*-img-set*:
assumes *C.ide b* **and** *C.ide a*
shows $\psi\ (b,\ a)$ ‘ *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* (λ*h. 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* (λ*h. 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* **by** *auto*
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* (λ*h. a* · *h* · *b*) *o* ψ (*b*, *a*))
    using *assms map-def* **by** *auto*
  also have ... = *S.mkArr* (*set* (*b*, *a*)) (*set* (*b*, *a*)) (λ*h. h*)
  proof −
    have *S.mkArr* (*set* (*b*, *a*)) (*set* (*b*, *a*)) (λ*h. h*) = ...
      using *assms S.arr-mkArr set-subset-Univ set-def C.comp-arr-dom C.comp-cod-arr*
      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* **by** *simp*
  finally show *?thesis* **by** *auto*
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 S.set-mkIde set-subset-Univ* **by** *simp*

The definition does in fact yield a functor.

interpretation *functor CopxC.comp S map*
proof
  fix *gf*
  assume $\neg$*CopxC.arr gf*
  thus *map gf* = *S.null* **using** *map-def* **by** *auto*
  next
  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*)
  proof −
    have *S.dom* (*map gf*) = *S.mkArr* (*set* (*CopxC.dom gf*)) (*set* (*CopxC.dom gf*)) (λ*x. x*)
      using *gf arr-map map-def* **by** *simp*
    also have ... = *S.mkArr* (*set* (*CopxC.dom gf*)) (*set* (*CopxC.dom gf*))
                        (φ (*CopxC.dom gf*) *o*
                        (λ*h. snd* (*CopxC.dom gf*) · *h* · *fst* (*CopxC.dom gf*)) *o*
                        ψ (*CopxC.dom gf*))
      using *gf set-subset-Univ* ψ*-mapsto map-def set-def*
      apply (*intro S.mkArr-eqI'*, *auto*)
      by (*metis C.comp-arr-dom C.comp-cod-arr C.in-homE*)
    also have ... = *map* (*CopxC.dom gf*)
      using *gf map-def C.arr-dom-iff-arr C.arr-cod-iff-arr* **by** *simp*
    finally show *?thesis* **by** *auto*
  qed
  show *S.cod* (*map gf*) = *map* (*CopxC.cod gf*)
  proof −
    have *S.cod* (*map gf*) = *S.mkArr* (*set* (*CopxC.cod gf*)) (*set* (*CopxC.cod gf*)) (λ*x. x*)
      using *gf map-def arr-map* **by** *simp*
    also have ... = *S.mkArr* (*set* (*CopxC.cod gf*)) (*set* (*CopxC.cod gf*))
                        (φ (*CopxC.cod gf*) *o*
                        (λ*h. snd* (*CopxC.cod gf*) · *h* · *fst* (*CopxC.cod gf*)) *o*
                        ψ (*CopxC.cod gf*))
      using *gf set-subset-Univ* ψ*-mapsto map-def set-def*
      apply (*intro S.mkArr-eqI'*, *auto*)
      by (*metis C.comp-arr-dom C.comp-cod-arr C.in-homE*)
    also have ... = *map* (*CopxC.cod gf*) **using** *gf map-def* **by** *simp*
    finally show *?thesis* **by** *auto*
  qed
  next
  fix *gf gf'*
  assume *gf'*: *CopxC.seq gf' gf*
  hence *seq*: *C.arr* (*fst gf*) $\wedge$ *C.arr* (*snd gf*) $\wedge$ *C.dom* (*snd gf'*) = *C.cod* (*snd gf*) $\wedge$
            C.arr (*fst gf'*) $\wedge$ *C.arr* (*snd gf'*) $\wedge$ *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'* $\odot$ *gf*) =
              S.mkArr (*set* (*CopxC.dom gf*)) (*set* (*CopxC.cod gf'*))
                    (φ (*CopxC.cod gf'*) *o* (λ*h. snd* (*gf'* $\odot$ *gf*) · *h* · *fst* (*gf'* $\odot$ *gf*))

$o\ \psi\ (CopxC.dom\ gf))$
using *gf′ map-def* **using** *CopxC.cod-comp CopxC.dom-comp* **by** *auto*
also have ... $= S.mkArr\ (set\ (CopxC.dom\ gf))\ (set\ (CopxC.cod\ gf'))$
$(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ gf' \cdot h \cdot fst\ gf') \circ \psi\ (CopxC.dom\ gf')$
$\circ$
$(\varphi\ (CopxC.cod\ gf) \circ (\lambda h.\ snd\ gf\ \cdot h \cdot fst\ gf) \circ \psi\ (CopxC.dom\ gf)))$
proof *(intro S.mkArr-eqI′)*
show $S.arr\ (S.mkArr\ (set\ (CopxC.dom\ gf))\ (set\ (CopxC.cod\ gf'))$
$(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ (gf' \odot gf) \cdot h \cdot fst\ (gf' \odot gf))$
$\circ\ \psi\ (CopxC.dom\ gf)))$
using *0 1* **by** *simp*
show $\bigwedge x.\ x \in set\ (CopxC.dom\ gf) \Longrightarrow$
$(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ (gf' \odot gf) \cdot h \cdot fst\ (gf' \odot gf)) \circ$
$\psi\ (CopxC.dom\ gf))\ x =$
$(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ gf' \cdot h \cdot fst\ gf') \circ \psi\ (CopxC.dom\ gf') \circ$
$(\varphi\ (CopxC.cod\ gf) \circ (\lambda h.\ snd\ gf\ \cdot h \cdot fst\ gf) \circ \psi\ (CopxC.dom\ gf)))\ x$
proof −
fix x
assume $x \in set\ (CopxC.dom\ gf)$
hence x: $x \in set\ (C.cod\ (fst\ gf),\ C.dom\ (snd\ gf))$
using *gf′ CopxC.seqE* **by** *(elim CopxC.seqE, fastforce)*
show $(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ (gf' \odot gf) \cdot h \cdot fst\ (gf' \odot gf)) \circ$
$\psi\ (CopxC.dom\ gf))\ x =$
$(\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ gf' \cdot h \cdot fst\ gf') \circ \psi\ (CopxC.dom\ gf') \circ$
$(\varphi\ (CopxC.cod\ gf) \circ (\lambda h.\ snd\ gf\ \cdot h \cdot fst\ gf) \circ \psi\ (CopxC.dom\ gf)))\ x$
proof −
have $(\varphi\ (CopxC.cod\ gf')\ o\ (\lambda h.\ snd\ (gf' \odot gf) \cdot h \cdot fst\ (gf' \odot gf))$
$o\ \psi\ (CopxC.dom\ gf))\ x =$
$\varphi\ (CopxC.cod\ gf')\ (snd\ (gf' \odot gf) \cdot \psi\ (CopxC.dom\ gf)\ x \cdot fst\ (gf' \odot gf))$
by *simp*
also have ... $= \varphi\ (CopxC.cod\ gf')$
$(((\lambda h.\ snd\ gf' \cdot h \cdot fst\ gf') \circ \psi\ (CopxC.dom\ gf') \circ$
$(\varphi\ (CopxC.dom\ gf') \circ (\lambda h.\ snd\ gf\ \cdot h \cdot fst\ gf)))$
$(\psi\ (CopxC.dom\ gf)\ x))$
proof −
have $C.ide\ (C.cod\ (fst\ gf)) \wedge C.ide\ (C.dom\ (snd\ gf))$
using *gf′* **by** *(elim CopxC.seqE, auto)*
hence $\ll\psi\ (C.cod\ (fst\ gf),\ C.dom\ (snd\ gf))\ x : C.cod\ (fst\ gf) \rightarrow C.dom\ (snd\ gf)\gg$
using *x ψ-mapsto* **by** *auto*
hence $\ll snd\ gf \cdot \psi\ (C.cod\ (fst\ gf),\ C.dom\ (snd\ gf))\ x \cdot fst\ gf :$
$C.cod\ (fst\ gf') \rightarrow C.dom\ (snd\ gf')\gg$
using *x seq* **by** *auto*
thus *?thesis*
using *seq ψ-φ C.comp-assoc* **by** *auto*
qed
also have ... $= (\varphi\ (CopxC.cod\ gf') \circ (\lambda h.\ snd\ gf' \cdot h \cdot fst\ gf') \circ \psi\ (CopxC.dom\ gf') \circ$
$(\varphi\ (CopxC.dom\ gf') \circ (\lambda h.\ snd\ gf\ \cdot h \cdot fst\ gf) \circ \psi\ (CopxC.dom\ gf)))$
x
by *auto*

finally show *?thesis* **using** *seq* **by** *simp*
qed
qed
qed
also have ... = $map\ gf' \cdot_S map\ gf$
using *seq gf' map-def arr-map [of gf] arr-map [of gf'] S.comp-mkArr* **by** *auto*
finally show $map\ (gf' \odot gf) = map\ gf' \cdot_S map\ gf$
using *seq gf'* **by** *auto*
qed

interpretation *binary-functor Cop.comp C S map* **..**

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

end

sublocale *hom-functor* ⊆ *binary-functor Cop.comp C S map*
using *is-binary-functor* **by** *auto*

context *hom-functor*
begin

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 $bij\text{-}betw\ (\varphi\ (b, a))\ (C.hom\ b\ a)\ (set\ (b, a))$
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 $map\ (g, f) =$
$S.mkArr\ (set\ (C.cod\ g, C.dom\ f))\ (set\ (C.dom\ g, C.cod\ f))$
$(\varphi\ (C.dom\ g, C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g, C.dom\ f))$
using *assms map-def* **by** *simp*
moreover have $S.arr\ (map\ (g, f))$ **using** *gf* **by** *simp*
ultimately have
$S.Fun\ (map\ (g, f)) =$
$restrict\ (\varphi\ (C.dom\ g, C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g, C.dom\ f))$
$(set\ (C.cod\ g, C.dom\ f))$
using *S.Fun-mkArr* **by** *simp*
hence $S.Fun\ (map\ (g, f))\ ?\varphi h =$

$(\varphi\ (C.dom\ g,\ C.cod\ f) \circ (\lambda h.\ f \cdot h \cdot g) \circ \psi\ (C.cod\ g,\ C.dom\ f))$ *?φh*
using *φh* **by** *simp*
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 $S.Fun\ (map\ (g,\ f)) =$
$restrict\ (\varphi\ (C.dom\ g,\ C.cod\ f)\ o\ (\lambda h.\ f \cdot h \cdot g)\ o\ \psi\ (C.cod\ g,\ C.dom\ f))$
$(set\ (C.cod\ g,\ C.dom\ f))$
using *assms map-def preserves-arr* **by** *force*

lemma *map-simp-1*:
assumes *C.arr g* **and** *C.ide a*
shows $map\ (g,\ a) = S.mkArr\ (set\ (C.cod\ g,\ a))\ (set\ (C.dom\ g,\ a))$
$(\varphi\ (C.dom\ g,\ a)\ o\ Cop.comp\ g\ o\ \psi\ (C.cod\ g,\ a))$
proof −
have *1*: $map\ (g,\ a) = S.mkArr\ (set\ (C.cod\ g,\ a))\ (set\ (C.dom\ g,\ a))$
$(\varphi\ (C.dom\ g,\ a)\ o\ (\lambda h.\ a \cdot h \cdot g)\ o\ \psi\ (C.cod\ g,\ a))$
using *assms map-def* **by** *force*
also have ... $= S.mkArr\ (set\ (C.cod\ g,\ a))\ (set\ (C.dom\ g,\ a))$
$(\varphi\ (C.dom\ g,\ a)\ o\ Cop.comp\ g\ o\ \psi\ (C.cod\ g,\ a))$
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** *auto*
qed

lemma *map-simp-2*:
assumes *C.ide b* **and** *C.arr f*
shows $map\ (b,\ f) = S.mkArr\ (set\ (b,\ C.dom\ f))\ (set\ (b,\ C.cod\ f))$
$(\varphi\ (b,\ C.cod\ f)\ o\ C\ f\ o\ \psi\ (b,\ C.dom\ f))$
proof −
have *1*: $map\ (b,\ f) = S.mkArr\ (set\ (b,\ C.dom\ f))\ (set\ (b,\ C.cod\ f))$
$(\varphi\ (b,\ C.cod\ f)\ o\ (\lambda h.\ f \cdot h \cdot b)\ o\ \psi\ (b,\ C.dom\ f))$
using *assms map-def* **by** *force*
also have ... $= S.mkArr\ (set\ (b,\ C.dom\ f))\ (set\ (b,\ C.cod\ f))$
$(\varphi\ (b,\ C.cod\ f)\ o\ C\ f\ o\ \psi\ (b,\ C.dom\ f))$

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** *auto*
qed

end

Every category C has a hom-functor: take S to be the set category *SetCat* generated by the set of arrows of C and take φ $(b,\ a)$ to be the map *UP* :: *'c* ⇒ *'c SetCat.arr*.

context *category*
begin

interpretation *Cop*: *dual-category C* **..**
interpretation *CopxC*: *product-category Cop.comp C* **..**
interpretation *S*: *set-category* ‹*SetCat.comp* :: *'a setcat.arr comp*›
using *is-set-category* **by** *auto*
interpretation *Hom*: *hom-functor C* ‹*SetCat.comp* :: *'a setcat.arr comp*› ‹λ-. *SetCat.UP*›
apply *unfold-locales*
using *UP-mapsto* **apply** *auto*[*1*]
using *inj-UP injD inj-onI* **by** *metis*

lemma *has-hom-functor*:
shows *hom-functor C* (*SetCat.comp* :: *'a setcat.arr comp*) (λ-. *UP*) **..**

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* +
functor C S F
for *C* :: *'c comp*
and *S* :: *'s comp*
and *F* :: *'c* ⇒ *'s*
begin

abbreviation *SET* :: *'c* ⇒ *'s set*
where *SET a* ≡ *S.set* (*F a*)

abbreviation *DOM* :: *'c* ⇒ *'s set*
where *DOM f* ≡ *S.Dom* (*F f*)

abbreviation *COD* :: *'c* ⇒ *'s set*
where *COD f* ≡ *S.Cod* (*F f*)

abbreviation *FUN* :: *'c* ⇒ *'s* ⇒ *'s*
where *FUN f* ≡ *S.Fun* (*F f*)

end

locale *set-valued-transformation* =
  C: *category C* +
  S: *set-category S* +
  F: *set-valued-functor C S F* +
  G: *set-valued-functor C S G* +
  natural-transformation C S F G τ
for *C* :: *'c comp*
and *S* :: *'s comp*
and *F* :: *'c* ⇒ *'s*
and *G* :: *'c* ⇒ *'s*
and τ :: *'c* ⇒ *'s*
begin

  abbreviation *DOM* :: *'c* ⇒ *'s set*
  where *DOM f* ≡ *S.Dom* (τ *f*)

  abbreviation *COD* :: *'c* ⇒ *'s set*
  where *COD f* ≡ *S.Cod* (τ *f*)

  abbreviation *FUN* :: *'c* ⇒ *'s* ⇒ *'s*
  where *FUN f* ≡ *S.Fun* (τ *f*)

end

16.2 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* +
  Hom: *hom-functor C S* φ +
  Cop-S: *functor-category Cop.comp S* +
  curried-functor' Cop.comp C S Hom.map
for *C* :: *'c comp* (**infixr** · *55*)
and *S* :: *'s comp* (**infixr** $\cdot_S$ *55*)
and φ :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
begin

  notation *Cop-S.in-hom* (≪- : - $\rightarrow_{[Cop,S]}$ -≫)

  abbreviation ψ
  where ψ ≡ *Hom*.ψ

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 $natural\text{-}transformation\ Cop.comp\ S\ (Y\ (C.dom\ f))\ (Y\ (C.cod\ f))\ (Y\ f)$
  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** $\ll g : b' \rightarrow b \gg$
shows $\ll Y\ a\ g : Hom.map\ (b,\ a) \rightarrow_S Hom.map\ (b',\ a) \gg$
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** $\ll f : a \rightarrow a' \gg$
shows $\ll Y\,f\,b : Hom.map\ (b,\ a) \rightarrow_S Hom.map\ (b,\ a') \gg$
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 $C\ S\ \varphi$
for C :: $'c\ comp$ (**infixr** $\cdot$ *55*)
and S :: $'s\ comp$ (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and a :: $'c$ +
assumes *ide-a*: $C.ide\ a$

sublocale *yoneda-functor-fixed-object* ⊆ *functor Cop.comp S* ($Y\ a$)
  using *ide-a Y-ide-is-functor* **by** *auto*
sublocale *yoneda-functor-fixed-object* ⊆ *set-valued-functor Cop.comp S* ($Y\ a$) **..**

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* =
  C: *category* C +
  Cop: *dual-category* C +
  S: *set-category* S +
  F: *set-valued-functor Cop.comp* $S\ F$ +
  yoneda-functor-fixed-object $C\ S\ \varphi\ a$
for C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and F :: $'c \Rightarrow 's$
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 $\mathcal{E}$ :: $('c \Rightarrow 's) \Rightarrow 's$
  where $\mathcal{E}\ \tau = S.Fun\ (\tau\ a)\ (\varphi\ (a,\ a)\ a)$

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 $\mathcal{T}o$ :: $'s \Rightarrow 'c \Rightarrow 's$
  where $\mathcal{T}o\ e\ b = S.mkArr\ (Hom.set\ (b,\ a))\ (F.SET\ b)\ (\lambda x.\ F.FUN\ (\psi\ (b,\ a)\ x)\ e)$

  lemma $\mathcal{T}$*o-e-ide*:
  assumes e: $e \in S.set\ (F\ a)$ **and** b: $C.ide\ b$
  shows $«\mathcal{T}o\ e\ b : Y\ a\ b \rightarrow_S F\ b»$
  and $\mathcal{T}o\ e\ b = S.mkArr\ (Hom.set\ (b,\ a))\ (F.SET\ b)\ (\lambda x.\ F.FUN\ (\psi\ (b,\ a)\ x)\ e)$

```
proof -
  show 𝒯o e b = S.mkArr (Hom.set (b, a)) (F.SET b) (λx. F.FUN (ψ (b, a) x) e)
    using 𝒯o-def by auto
  moreover 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)
    hence «ψ (b, a) x : b → a» using assms ide-a Hom.ψ-mapsto by auto
    hence F.FUN (ψ (b, a) x) ∈ F.SET a → F.SET b
      using S.Fun-mapsto [of F (ψ (b, a) x)] by fastforce
    thus F.FUN (ψ (b, a) x) e ∈ F.SET b using e by auto
  qed
  ultimately show «𝒯o e b : Y a b →S F b»
    using ide-a b S.mkArr-in-hom [of Hom.set (b, a) F.SET b] Hom.set-subset-Univ
    by auto
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 𝒯o-e-induces-transformation:
assumes e: e ∈ S.set (F a)
shows transformation-by-components Cop.comp S (Y a) F (𝒯o e)
proof
  fix b :: 'c
  assume b: Cop.ide b
  show «𝒯o e b : Y a b →S F b»
    using ide-a b e 𝒯o-e-ide by simp
  next
  fix g :: 'c
  assume g: Cop.arr g
  let ?b = Cop.dom g
  let ?b' = Cop.cod g
  show 𝒯o e (Cop.cod g) ·S Y a g = F g ·S 𝒯o e (Cop.dom g)
  proof -
    have 1: 𝒯o e (Cop.cod g) ·S Y a g
               = S.mkArr (Hom.set (?b, a)) (F.SET ?b')
                    ((λx. F.FUN (ψ (?b', a) x) e)
                       o (φ (?b', a) o Cop.comp g o ψ (?b, a)))
    proof -
      have S.arr (S.mkArr (Hom.set (Cop.cod g, a)) (F.SET (Cop.cod g))
                 (λs. F.FUN (ψ (Cop.cod g, a) s) e)) ∧
           S.dom (S.mkArr (Hom.set (Cop.cod g, a)) (F.SET (Cop.cod g))
                 (λs. F.FUN (ψ (Cop.cod g, a) s) e)) = Y a (Cop.cod g) ∧
           S.cod (S.mkArr (Hom.set (Cop.cod g, a)) (F.SET (Cop.cod g))
                 (λs. F.FUN (ψ (Cop.cod g, a) s) e)) = F (Cop.cod g)
        using Cop.cod-char 𝒯o-e-ide [of e ?b'] 𝒯o-e-ide [of e ?b'] e g by force
     moreover have Y a g = S.mkArr (Hom.set (Cop.dom g, a)) (Hom.set (Cop.cod g, a))
                              (φ (Cop.cod g, a) ∘ Cop.comp g ∘ ψ (Cop.dom g, 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 𝒯o-e-ide
            S.comp-mkArr [of Hom.set (?b, a) Hom.set (?b′, a)
                            φ (?b′, a) o Cop.comp g o ψ (?b, a)
                            F.SET ?b′ λx. F.FUN (ψ (?b′, a) x) e]
      by (metis C.ide-dom Cop.arr-char preserves-arr)
  qed
  also have ... = S.mkArr (Hom.set (?b, a)) (F.SET ?b′)
                          (F.FUN g o (λx. F.FUN (ψ (?b, a) x) e))
  proof (intro S.mkArr-eqI′)
    have (λx. F.FUN (ψ (?b′, a) x) e)
            o (φ (?b′, a) o Cop.comp g o ψ (?b, a)) ∈ Hom.set (?b, a) → F.SET ?b′
    proof −
      have S.arr (S (𝒯o e ?b′) (Y a g))
        using ide-a e g 𝒯o-e-ide [of e ?b′] Y-ide-arr(1) [of a C.dom g C.cod g g]
        by auto
      thus ?thesis using 1 by simp
    qed
    thus S.arr (S.mkArr (Hom.set (?b, a)) (F.SET ?b′)
                        ((λx. F.FUN (ψ (?b′, a) x) e)
                          o (φ (?b′, a) o Cop.comp g o ψ (?b, a))))
      using ide-a e g Hom.set-subset-Univ by simp
    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 1: «ψ (Cop.dom g, a) x : Cop.dom g → a»
          using ide-a x g Hom.ψ-mapsto [of ?b a] by auto
        moreover have S.seq (F g) (F (ψ (C.cod g, a) x))
          using 1 g by (intro S.seqI′, auto)
        moreover have ψ (C.dom g, a) (φ (C.dom g, a) (C (ψ (C.cod g, a) x) g)) =
                      C (ψ (C.cod g, a) x) g
          using g 1 Hom.ψ-φ [of C (ψ (?b, a) x) g ?b′ a] by fastforce
        ultimately show ?thesis
          using assms 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 𝒯o e (Cop.dom g)
    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 (𝒯o e ?b) ∧
           𝒯o e ?b = S.mkArr (Hom.set (?b, a)) (F.SET ?b) (λx. F.FUN (ψ (?b, a) x) e)
        using e g 𝒯o-e-ide
        by (metis C.ide-cod Cop.arr-char Cop.dom-char S.in-homE)
      ultimately show ?thesis
        using S.comp-mkArr [of Hom.set (?b, a) F.SET ?b λx. F.FUN (ψ (?b, a) x) e
                            F.SET ?b' F.FUN g]
        by metis
    qed
    finally show ?thesis by blast
  qed
qed

abbreviation 𝒯 :: 's ⇒ 'c ⇒ 's
where 𝒯 e ≡ transformation-by-components.map Cop.comp S (Y a) (𝒯o e)

end

locale yoneda-lemma-fixed-e =
  yoneda-lemma C S φ F a
for C :: 'c comp (infixr · 55)
and S :: 's comp (infixr ·S 55)
and φ :: 'c * 'c ⇒ 'c ⇒ 's
and F :: 'c ⇒ 's
and a :: 'c
and e :: 's +
assumes E: e ∈ F.SET a
begin

  interpretation 𝒯e: transformation-by-components Cop.comp S ‹Y a› F ‹𝒯o e›
    using E 𝒯o-e-induces-transformation by auto

  lemma natural-transformation-𝒯e:
  shows natural-transformation Cop.comp S (Y a) F (𝒯 e) ..

  lemma 𝒯e-ide:
  assumes Cop.ide b
  shows S.arr (𝒯 e b)
  and 𝒯 e b = S.mkArr (Hom.set (b, a)) (F.SET b) (λx. F.FUN (ψ (b, a) x) e)
    using assms apply fastforce
    using assms 𝒯o-def by auto
```

```
end

locale yoneda-lemma-fixed-τ =
  yoneda-lemma C S φ F a +
  τ: set-valued-transformation Cop.comp S Y a F τ
for C :: 'c comp (infixr · 55)
and S :: 's comp (infixr ·S 55)
and φ :: 'c * 'c ⇒ 'c ⇒ 's
and F :: 'c ⇒ 's
and a :: 'c
and τ :: 'c ⇒ 's
begin
```

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 τ b = S.mkArr (Hom.set (b, a)) (F.SET b)
                       (λx. (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 [of a a] by fastforce
    have 1: τ 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) (λx. (F.FUN (ψ (b, a) x) (τ.FUN a ?φa)))
    proof (intro S.mkArr-eqI')
      show 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 ⋀x. x ∈ Hom.set (b, a) ⟹ 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 ((φ (b, a) o Cop.comp ?ψx) a)
                       = (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.arr (Y a ?ψx)
```

```
      using ide-a ψx preserves-arr by (elim C.in-homE, auto)
    moreover have Y a ?ψx = S.mkArr (Hom.set (a, a)) (SET b)
                                (φ (b, a) ∘ Cop.comp ?ψx ∘ ψ (a, a))
      using ide-a b ψx preserves-hom Y-ide-arr Hom.set-map C.arrI by auto
    moreover have S.arr (τ b) ∧ τ b = S.mkArr (SET b) (F.SET b) (τ.FUN b)
      using ide-a b S.mkArr-Fun [of τ b] by simp
    ultimately 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)))
      using 1 S.comp-mkArr S.seqI
      by (metis S.cod-mkArr S.dom-mkArr)
    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 [of ?ψx] 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)
      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 F ?ψx ·S τ a =
                    S.mkArr (Hom.set (a, a)) (F.SET b) (F.FUN ?ψx o τ.FUN a)
        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
    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
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** $\tau'\ a = \tau\ a$
shows $\tau' = \tau$
proof (*intro NaturalTransformation.eqI*)
interpret τ': *natural-transformation Cop.comp S* $\langle Y\ a\rangle$ *F* τ' **using** *assms* **by** *auto*
interpret T': *yoneda-lemma-fixed-*τ *C S* 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 $\bigwedge b.\ Cop.ide\ b \Longrightarrow \tau'\ b = \tau\ b$
using *assms*(*2*) τ*-ide* $T'.\tau$*-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 $\mathcal{T}$*-is-injection*:
assumes $e \in F.SET\ a$
shows *natural-transformation Cop.comp S* $(Y\ a)$ *F* $(\mathcal{T}\ e)$ **and** $\mathcal{E}\ (\mathcal{T}\ e) = e$
proof −
interpret *yoneda-lemma-fixed-e C S* φ *F a e*
using *assms* **by** (*unfold-locales*, *auto*)
interpret $\mathcal{T}e$: *natural-transformation Cop.comp S* $\langle Y\ a\rangle$ *F* $\langle\mathcal{T}\ e\rangle$
using *natural-transformation-*$\mathcal{T}e$ **by** *auto*
show *natural-transformation Cop.comp S* $(Y\ a)$ *F* $(\mathcal{T}\ e)$ **..**
show $\mathcal{E}\ (\mathcal{T}\ e) = e$
unfolding $\mathcal{E}$*-def*
using *assms* $\mathcal{T}e$*-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 $\mathcal{E}\tau$*-in-Fa*:
assumes *natural-transformation Cop.comp S* $(Y\ a)$ *F* τ
shows $\mathcal{E}\ \tau \in F.SET\ a$
proof −
interpret τ: *natural-transformation Cop.comp S* $\langle Y\ a\rangle$ *F* τ **using** *assms* **by** *auto*
interpret *yoneda-lemma-fixed-*τ *C S* φ *F a* τ **..**

show *?thesis*
proof (*unfold* $\mathcal{E}$*-def*)
have $S.arr\ (\tau\ a) \wedge S.Dom\ (\tau\ a) = Hom.set\ (a,\ a) \wedge S.Cod\ (\tau\ a) = F.SET\ a$
using *ide-a Hom.set-map* **by** *auto*
hence $\tau.FUN\ a \in Hom.set\ (a,\ a) \to F.SET\ a$
using *S.Fun-mapsto* **by** *blast*
thus $\tau.FUN\ a\ (\varphi\ (a,\ a)\ a) \in F.SET\ a$
using *ide-a Hom.*φ*-mapsto* **by** *fastforce*
qed
qed

The other half of the Yoneda lemma: The mapping $\mathcal{T}$ is a surjection, with right inverse $\mathcal{E}$, taking natural transformations from $Y\ a$ to F to elements of $F.SET\ a$.

lemma $\mathcal{T}$*-is-surjection*:
assumes *natural-transformation Cop.comp S* $(Y\ a)\ F\ \tau$
shows $\mathcal{E}\ \tau \in F.SET\ a$ **and** $\mathcal{T}\ (\mathcal{E}\ \tau) = \tau$
proof −
interpret *natural-transformation Cop.comp S* $\langle Y\ a\rangle\ F\ \tau$ **using** *assms* **by** *auto*
interpret *yoneda-lemma-fixed-*τ $C\ S\ \varphi\ F\ a\ \tau$ **..**
show *1*: $\mathcal{E}\ \tau \in F.SET\ a$ **using** *assms* $\mathcal{E}\tau$*-in-Fa* **by** *auto*
interpret *yoneda-lemma-fixed-e* $C\ S\ \varphi\ F\ a\ \langle\mathcal{E}\ \tau\rangle$
using *1* **by** (*unfold-locales*, *auto*)
interpret $\mathcal{T}e$: *natural-transformation Cop.comp S* $\langle Y\ a\rangle\ F\ \langle\mathcal{T}\ (\mathcal{E}\ \tau)\rangle$
using *natural-transformation-*$\mathcal{T}e$ **by** *auto*
show $\mathcal{T}\ (\mathcal{E}\ \tau) = \tau$
proof (*intro eqI*)
show *natural-transformation Cop.comp S* $(Y\ a)\ F\ (\mathcal{T}\ (\mathcal{E}\ \tau))$ **..**
show $\mathcal{T}\ (\mathcal{E}\ \tau)\ a = \tau\ a$
using *ide-a* τ*-ide* [*of a*] $\mathcal{T}e$*-ide* $\mathcal{E}$*-def* **by** *simp*
qed
qed

The main result.

theorem *yoneda-lemma*:
shows *bij-betw* $\mathcal{T}\ (F.SET\ a)\ \{\tau.$ *natural-transformation Cop.comp S* $(Y\ a)\ F\ \tau\}$
using $\mathcal{T}$*-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* =
C: *category* C +
Cop: *dual-category* C +
S: *set-category* S +
yoneda-functor-fixed-object $C\ S\ \varphi\ a$ +

Ya′: *yoneda-functor-fixed-object C S φ a′* +
yoneda-lemma C S φ Y a′ a
for *C* :: *′c comp* (**infixr** · *55*)
and *S* :: *′s comp* (**infixr** $\cdot_S$ *55*)
and *φ* :: *′c* ∗ *′c* ⇒ *′c* ⇒ *′s*
and *F* :: *′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 *𝒯-equals-Yoψ*:
assumes *e*: *e* ∈ *Hom.set* (*a*, *a′*)
shows *𝒯 e* = *Y* (*ψ* (*a*, *a′*) *e*)
proof −
let *?ψe* = *ψ* (*a*, *a′*) *e*
have *ψe*: «*?ψe* : *a* → *a′*» **using** *ide-a ide-a′ e Hom.ψ-mapsto* [*of a a′*] **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 φ* ‹*Y a′*› *a e*
using *ide-a ide-a′ e S.set-mkIde Hom.set-map*
by (*unfold-locales*, *simp-all*)
interpret *𝒯e*: *natural-transformation Cop.comp S* ‹*Y a*› ‹*Y a′*› ‹*𝒯 e*›
using *natural-transformation-𝒯e* **by** *auto*
interpret *yoneda-lemma-fixed-τ C S φ* ‹*Y a′*› *a* ‹*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′*) (*𝒯 e*) **..**
moreover have *𝒯 e a* = *Y ?ψe a*
proof −
have *1*: *S.arr* (*𝒯 e a*)
using *ide-a e 𝒯e.preserves-reflects-arr* **by** *simp*
have *2*: *𝒯 e a* = *S.mkArr* (*Hom.set* (*a*, *a*)) (*Ya′.SET a*) (*λx. Ya′.FUN* (*ψ* (*a*, *a*) *x*) *e*)
using *ide-a 𝒯o-def 𝒯e-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*) (*λx. Ya′.FUN* (*ψ* (*a*, *a*) *x*) *e*))
using *ide-a e 1 2* **by** *simp*
show *Hom.set* (*a*, *a*) = *Hom.set* (*a*, *a*) **..**
show *3*: *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 $S.arr\ (Y\ a'\ (\psi\ (a,\ a)\ x)) \wedge$
$Y\ a'\ (\psi\ (a,\ a)\ x) = S.mkArr\ (Hom.set\ (a,\ a'))\ (Hom.set\ (a,\ a'))$
$(\varphi\ (a,\ a') \circ Cop.comp\ (\psi\ (a,\ a)\ x) \circ \psi\ (a,\ a'))$
using *Y-ide-arr ide-a ide-a' ψx* **by** *blast*
hence $Ya'.FUN\ (\psi\ (a,\ a)\ x)\ e = (\varphi\ (a,\ a') \circ Cop.comp\ (\psi\ (a,\ a)\ x) \circ \psi\ (a,\ a'))\ e$
using *e 3 S.Fun-mkArr Ya'.preserves-reflects-arr* [*of* $\psi\ (a,\ a)\ x$] **by** *simp*
also have ... $= (\varphi\ (a,\ a')\ o\ C\ ?\psi e\ o\ \psi\ (a,\ a))\ x$ **by** *simp*
finally show $Ya'.FUN\ (\psi\ (a,\ a)\ x)\ e = (\varphi\ (a,\ a')\ o\ C\ ?\psi e\ o\ \psi\ (a,\ a))\ x$ **by** *auto*
qed
qed
also have ... $= Y\ ?\psi e\ a$
using *ide-a ide-a' Y-arr-ide ψe* **by** *simp*
finally show $\mathcal{T}\ e\ a = Y\ ?\psi e\ a$ **by** *auto*
qed
ultimately show *?thesis* **using** *eqI* **by** *auto*
qed

lemma *Y-injective-on-homs*:
assumes $\ll f : a \to a'\gg$ **and** $\ll f' : a \to a'\gg$ **and** $map\ f = map\ f'$
shows $f = f'$
proof −
have $f = \psi\ (a,\ a')\ (\varphi\ (a,\ a')\ f)$
using *assms ide-a Hom.ψ-φ* **by** *simp*
also have ... $= \psi\ (a,\ a')\ (\mathcal{E}\ (\mathcal{T}\ (\varphi\ (a,\ a')\ f)))$
using *ide-a ide-a' assms(1) 𝒯-is-injection Hom.φ-mapsto Hom.set-map*
by (*elim C.in-homE*, *simp add*: *Pi-iff*)
also have ... $= \psi\ (a,\ a')\ (\mathcal{E}\ (Y\ (\psi\ (a,\ a')\ (\varphi\ (a,\ a')\ f))))$
using *assms Hom.φ-mapsto* [*of a a'*] *𝒯-equals-Yoψ* [*of* $\varphi\ (a,\ a')\ f$] **by** *force*
also have ... $= \psi\ (a,\ a')\ (\mathcal{E}\ (\mathcal{T}\ (\varphi\ (a,\ a')\ f')))$
using *assms Hom.φ-mapsto* [*of a a'*] *ide-a Hom.ψ-φ Y-def*
𝒯-equals-Yoψ [*of* $\varphi\ (a,\ a')\ f'$]
by *fastforce*
also have ... $= \psi\ (a,\ a')\ (\varphi\ (a,\ a')\ f')$
using *ide-a ide-a' assms(2) 𝒯-is-injection Hom.φ-mapsto Hom.set-map*
by (*elim C.in-homE*, *simp add*: *Pi-iff*)
also have ... $= f'$
using *assms ide-a Hom.ψ-φ* **by** *simp*
finally show $f = f'$ **by** *auto*
qed

lemma *Y-surjective-on-homs*:
assumes τ: *natural-transformation Cop.comp S* $(Y\ a)\ (Y\ a')\ \tau$
shows $Y\ (\psi\ (a,\ a')\ (\mathcal{E}\ \tau)) = \tau$
using *ide-a ide-a' τ 𝒯-is-surjection 𝒯-equals-Yoψ ℰτ-in-Fa Hom.set-map* **by** *simp*

end

context *yoneda-functor*
begin

lemma *is-faithful-functor*:
shows *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* φ ⟨*C.cod f*⟩
      using *par* **by** (*unfold-locales*, *auto*)
    interpret *yoneda-lemma-for-hom C S* φ ⟨*Y* (*C.cod f*)⟩ ⟨*C.dom f*⟩ ⟨*C.cod f*⟩
      using *par* **by** (*unfold-locales*, *auto*)
    show $f = f'$ **using** *par ff' Y-injective-on-homs* [*of f f'*] **by** *fastforce*
  qed
qed

lemma *is-full-functor*:
shows *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* $\rightarrow_{[Cop,S]}$ *map a'*»
  show $\exists e$. «e : $a \rightarrow a'$» $\wedge$ *map e = t*
  proof
    interpret *Ya'*: *yoneda-functor-fixed-object C S* φ *a'*
      using *a'* **by** (*unfold-locales*, *auto*)
    interpret *yoneda-lemma-for-hom C S* φ ⟨*Y a'*⟩ *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*: $\mathcal{E}$ (*Cop-S.Map t*) $\in$ *Hom.set* (*a*, *a'*)
      using $\mathcal{E}\tau$*-in-Fa ide-a ide-a' Hom.set-map* **by** *simp*
    moreover have *map* (ψ (*a*, *a'*) ($\mathcal{E}$ (*Cop-S.Map t*))) = *t*
    proof (*intro Cop-S.arr-eqI*)
      have *2*: «*map* (ψ (*a*, *a'*) ($\mathcal{E}$ (*Cop-S.Map t*))) : *map a* $\rightarrow_{[Cop,S]}$ *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'*) ($\mathcal{E}$ (*Cop-S.Map t*)))) **using** *2* **by** *blast*
      show *3*: *Cop-S.Map* (*map* (ψ (*a*, *a'*) ($\mathcal{E}$ (*Cop-S.Map t*)))) = *Cop-S.Map t*
        using *NT Y-surjective-on-homs Y-def* **by** *simp*
      show *4*: *Cop-S.Dom* (*map* (ψ (*a*, *a'*) ($\mathcal{E}$ (*Cop-S.Map t*)))) = *Cop-S.Dom t*
        using *t 2 natural-transformation-axioms Cop-S.Map-dom* **by** (*metis Cop-S.in-homE*)
      show *Cop-S.Cod* (*map* (ψ (*a*, *a'*) ($\mathcal{E}$ (*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'*) ($\mathcal{E}$ (*Cop-S.Map t*)) : $a \rightarrow a'$» $\wedge$
                  map (ψ (*a*, *a'*) ($\mathcal{E}$ (*Cop-S.Map t*))) = *t*

using *ide-a ide-a′ Hom.*ψ*-mapsto* **by** *auto*
qed
qed

end

sublocale *yoneda-functor* $\subseteq$ *faithful-functor C Cop-S.comp map*
using *is-faithful-functor* **by** *auto*
sublocale *yoneda-functor* $\subseteq$ *full-functor C Cop-S.comp map* **using** *is-full-functor* **by** *auto*
sublocale *yoneda-functor* $\subseteq$ *fully-faithful-functor C Cop-S.comp map* **..**

end

Chapter 17

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.

17.1 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** $\cdot_D$ *55*)
  and C :: $'c\ comp$ (**infixr** $\cdot_C$ *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* $(\ll\text{-} : \text{-} \rightarrow_C \text{-}\gg)$
notation *D.in-hom* $(\ll\text{-} : \text{-} \rightarrow_D \text{-}\gg)$

"g is a D-coextension of f along e."

definition *is-coext* :: $'d \Rightarrow 'c \Rightarrow 'd \Rightarrow bool$
where *is-coext* $x'\ f\ g \equiv \ll g : x' \rightarrow_D x\gg \wedge f = e \cdot_C F\ g$

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** $\cdot_D$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot_C$ *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 $\ll$*the-coext* $x'\ f : x' \rightarrow_D x\gg$ **and** $f = e \cdot_C F$ (*the-coext* $x'\ f$)
  using *assms is-terminal the-coext-def is-coext-def theI2* [*of* $\lambda g.$ *is-coext* $x'\ f\ g$]
   apply *metis*
  using *assms is-terminal the-coext-def is-coext-def theI2* [*of* $\lambda g.$ *is-coext* $x'\ f\ g$]
  by *metis*

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 $F: D \rightarrow C$ 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** $\cdot_D$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot_C$ *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-hom* (≪- : - $\rightarrow_C$ -≫)
notation *D.in-hom* (≪- : - $\rightarrow_D$ -≫)

end

17.2 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** $\cdot_C$ *55*)
and D :: $'d$ *comp* (**infixr** $\cdot_D$ *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-hom* (≪- : - $\rightarrow_C$ -≫)
notation *D.in-hom* (≪- : - $\rightarrow_D$ -≫)

"f is a C-extension of g along e."

definition *is-ext* :: $'c \Rightarrow 'd \Rightarrow 'c \Rightarrow bool$
where $is\text{-}ext\ y'\ g\ f \equiv «f : y \rightarrow_C y'» \wedge g = G\ f \cdot_D e$

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** $\cdot_C$ *55*)
and D :: $'d$ *comp* (**infixr** $\cdot_D$ *55*)
and G :: $'c \Rightarrow 'd$
and x :: $'d$
and y :: $'c$
and e :: $'d$ +
assumes *is-initial*: $arrow\text{-}to\text{-}functor\ C\ D\ G\ x\ y'\ g \Longrightarrow (\exists !f.\ is\text{-}ext\ y'\ g\ f)$
begin

definition *the-ext* :: $'c \Rightarrow 'd \Rightarrow 'c$
where *the-ext* y' g = (*THE* f. *is-ext* y' g f)

lemma *the-ext-prop*:
assumes *arrow-to-functor* C D G x y' g
shows «*the-ext* y' g : $y \rightarrow_C y'$» **and** $g = G$ (*the-ext* y' g) $\cdot_D$ e
 using *assms is-initial the-ext-def is-ext-def theI2* [*of* λf. *is-ext* y' g f]
  apply *metis*
 using *assms is-initial the-ext-def is-ext-def theI2* [*of* λf. *is-ext* y' g f]
 by *metis*

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: $C \rightarrow D$ 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** $\cdot_C$ 55)
 and D :: $'d$ *comp* (**infixr** $\cdot_D$ 55)
 and G :: $'c \Rightarrow 'd$ +
 assumes *initial-arrows-exist*: $D.ide\ x \Longrightarrow (\exists\, y\ e.$ *initial-arrow-to-functor* $C\ D\ G\ x\ y\ e)$
begin

 notation *C.in-hom* («- : - $\rightarrow_C$ -»)
 notation *D.in-hom* («- : - $\rightarrow_D$ -»)

end

17.3 Various Definitions of Adjunction

17.3.1 Meta-Adjunction

A "meta-adjunction" consists of a functor F: $D \rightarrow C$, a functor G: $C \rightarrow D$, 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** $\cdot_C$ *55*)
and D :: $'d$ *comp* (**infixr** $\cdot_D$ *55*)
and F :: $'d \Rightarrow 'c$
and G :: $'c \Rightarrow 'd$
and φ :: $'d \Rightarrow 'c \Rightarrow 'd$
and ψ :: $'c \Rightarrow 'd \Rightarrow 'c$ +
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: $\langle\!\langle f : x \rightarrow_C x' \rangle\!\rangle$ **and** g: $\langle\!\langle g : y' \rightarrow_D y \rangle\!\rangle$ **and** h: $\langle\!\langle h : y \rightarrow_D G\ x \rangle\!\rangle$
shows $f \cdot_C \psi\ x\ h \cdot_C F\ g = \psi\ x'\ (G\ f \cdot_D h \cdot_D g)$
proof −
  have $\langle\!\langle f \cdot_C \psi\ x\ h \cdot_C F\ g : F\ y' \rightarrow_C x' \rangle\!\rangle$
    using $f\ g\ h$ ψ-*in-hom* [*of* x h] **by** *fastforce*
  moreover have $\langle\!\langle (G\ f \cdot_D h) \cdot_D g : y' \rightarrow_D G\ x' \rangle\!\rangle$
    using $f\ g\ h$ φ-*in-hom* **by** *auto*
  moreover have $\psi\ x'\ (\varphi\ y'\ (f \cdot_C \psi\ x\ h \cdot_C F\ g)) = \psi\ x'\ (G\ f \cdot_D \varphi\ y\ (\psi\ x\ h) \cdot_D g)$
  proof −
    have $\langle\!\langle \psi\ x\ h : F\ y \rightarrow_C x \rangle\!\rangle$
      using $f\ h$ ψ-*in-hom* **by** *auto*
    thus *?thesis* **using** $f\ g$ φ-*naturality*
      by *force*
  qed
  ultimately show *?thesis*
    using $f\ h$ ψ-φ φ-ψ
    by (*metis C.arrI C.ide-dom C.in-homE D.arrI D.ide-dom D.in-homE*)
qed

end

17.3.2 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* +
  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* φC +
  HomD: *hom-functor D S* 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 Hom-FopxC.map* +
  Hom-DopxG: *set-valued-functor DopxC.comp S Hom-DopxG.map* +
  Φ: *set-valued-transformation DopxC.comp S Hom-FopxC.map Hom-DopxG.map* Φ +
  Ψ: *set-valued-transformation DopxC.comp S Hom-DopxG.map Hom-FopxC.map* Ψ +
  ΦΨ: *inverse-transformations DopxC.comp S Hom-FopxC.map Hom-DopxG.map* Φ Ψ
  for C :: *'c comp* (**infixr** $\cdot_C$ *55*)
  and D :: *'d comp* (**infixr** $\cdot_D$ *55*)
  and S :: *'s comp* (**infixr** $\cdot_S$ *55*)
  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 $C.in\text{-}hom$ $(\ll\text{-} : \text{-} \to_C \text{-}\gg)$
notation $D.in\text{-}hom$ $(\ll\text{-} : \text{-} \to_D \text{-}\gg)$

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

17.3.3 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** $\cdot_C$ *55*)
and $D :: {'d}\ comp$ (**infixr** $\cdot_D$ *55*)
and $F :: {'d} \Rightarrow {'c}$
and $G :: {'c} \Rightarrow {'d}$
and $\eta :: {'d} \Rightarrow {'d}$
and $\varepsilon :: {'c} \Rightarrow {'c}$ +
assumes *triangle-F*: $\varepsilon FoF\eta.map = F$
and *triangle-G*: $G\varepsilon o\eta G.map = G$
begin

notation $C.in\text{-}hom$ $(\ll\text{-} : \text{-} \to_C \text{-}\gg)$
notation $D.in\text{-}hom$ $(\ll\text{-} : \text{-} \to_D \text{-}\gg)$

end

lemma *unit-determines-counit*:
assumes *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$
and *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon'$
shows $\varepsilon = \varepsilon'$
proof –

interpret *Adj*: *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$ **using** *assms(1)* **by** *auto*
interpret *Adj'*: *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon'$ **using** *assms(2)* **by** *auto*
interpret *FGFG*: *composite-functor* $C\ D\ C\ G\ \langle F\ o\ G\ o\ F\rangle$ **..**
interpret $FG\varepsilon$: *natural-transformation* $C\ C\ \langle(F\ o\ G)\ o\ (F\ o\ G)\rangle\ \langle F\ o\ G\rangle\ \langle(F\ o\ G)\ o\ \varepsilon\rangle$
using *Adj.ε.natural-transformation-axioms Adj.FG.natural-transformation-axioms horizontal-composite Adj.FG.functor-axioms*
by *fastforce*
interpret $F\eta G$: *natural-transformation* $C\ C\ \langle F\ o\ G\rangle\ \langle F\ o\ G\ o\ F\ o\ G\rangle\ \langle F\ o\ \eta\ o\ G\rangle$
using *Adj.η.natural-transformation-axioms Adj.Fη.natural-transformation-axioms Adj.G.natural-transformation-axioms horizontal-composite*
by *blast*
interpret $\varepsilon'\varepsilon$: *natural-transformation* $C\ C\ \langle F\ o\ G\ o\ F\ o\ G\rangle$ *Adj.C.map* $\langle\varepsilon'\ o\ \varepsilon\rangle$
proof –
have *natural-transformation* $C\ C\ ((F\ o\ G)\ o\ (F\ o\ G))$ *Adj.C.map* $(\varepsilon'\ o\ \varepsilon)$
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* $(\varepsilon'\ o\ \varepsilon)$
using *o-assoc* **by** *metis*
qed
interpret $\varepsilon'\varepsilon oF\eta G$: *vertical-composite*
$C\ C\ \langle F\ o\ G\rangle\ \langle F\ o\ G\ o\ F\ o\ G\rangle$ *Adj.C.map* $\langle F\ o\ \eta\ o\ G\rangle\ \langle\varepsilon'\ o\ \varepsilon\rangle$ **..**
have ε' = *vertical-composite.map* $C\ C\ (F\ o$ *Adj.GεoηG.map*$)\ \varepsilon'$
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* $C\ C\ (F\ o\ \eta\ o\ G)\ (F\ o\ G\ o\ \varepsilon))\ \varepsilon'$
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* $C\ C\ (F\ o\ \eta\ o\ G)\ (\varepsilon'\ o\ F\ o\ G))\ \varepsilon$
proof –
have *vertical-composite.map* $C\ C$
(*vertical-composite.map* $C\ C\ (F\ o\ \eta\ o\ G)\ (F\ o\ G\ o\ \varepsilon))\ \varepsilon'$
= *vertical-composite.map* $C\ C\ (F\ o\ \eta\ o\ G)$
(*vertical-composite.map* $C\ C\ (F\ o\ G\ o\ \varepsilon)\ \varepsilon'$)
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* $C\ C\ (F\ o\ \eta\ o\ G)$
(*vertical-composite.map* $C\ C\ (\varepsilon'\ o\ F\ o\ G)\ \varepsilon$)
proof –

```
      have ε' ∘ Adj.C.map = ε'
        using Adj'.ε.natural-transformation-axioms hcomp-ide-dom by simp
      moreover have Adj.C.map ∘ ε = ε
        using Adj.ε.natural-transformation-axioms hcomp-ide-cod by simp
      moreover have ε' ∘ (F o G) = ε' o F ∘ G by auto
      ultimately show ?thesis
        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 simp
    qed
    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.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 ... = vertical-composite.map C C (F o G) ε
    using Adj'.triangle-F by simp
  also have ... = ε
    using 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.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.natural-transformation-axioms comp-identity-functor horizontal-composite
    by (metis (no-types, lifting))
  interpret GεF: 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.natural-transformation-axioms
          Adj.Gε.natural-transformation-axioms horizontal-composite
```

by *blast*
interpret $\eta'\eta$: *natural-transformation D D Adj.D.map* $\langle G\ o\ F\ o\ G\ o\ F\rangle$ $\langle \eta'\ o\ \eta\rangle$
proof −
  have *natural-transformation D D Adj.D.map* $((G\ o\ F)\ o\ (G\ o\ F))$ $(\eta'\ o\ \eta)$
    using *Adj.η.natural-transformation-axioms Adj'.η.natural-transformation-axioms*
      horizontal-composite Adj.D.natural-transformation-axioms hcomp-ide-cod
    by (*metis* (*no-types*, *lifting*))
  thus *natural-transformation D D Adj.D.map* $(G\ o\ F\ o\ G\ o\ F)$ $(\eta'\ o\ \eta)$
    using *o-assoc* **by** *metis*
qed
interpret $G\varepsilon Fo\eta'\eta$: *vertical-composite*
      D D Adj.D.map $\langle G\ o\ F\ o\ G\ o\ F\rangle$ $\langle G\ o\ F\rangle$ $\langle \eta'\ o\ \eta\rangle$ $\langle G\ o\ \varepsilon\ o\ F\rangle$ **..**
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* $(G\ o\ (F\ o\ \eta))$ $(G\ o\ (\varepsilon\ o\ F))$)
  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' $(G\ o\ (F\ o\ \eta))$) $(G\ o\ \varepsilon\ o\ F)$
  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 $(\eta'\ o\ G\ o\ F)$) $(G\ o\ \varepsilon\ o\ F)$
proof −
  have $\eta' \circ$ *Adj.D.map* = η'
    using *Adj'.η.natural-transformation-axioms hcomp-ide-dom* **by** *simp*
  moreover have $\eta'\ o\ (G\ o\ F) = \eta'\ o\ G\ o\ F \wedge G\ o\ (F\ o\ \eta) = G\ o\ F\ o\ \eta$ **by** *auto*
  ultimately show *?thesis*
    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 *simp*
qed
also have ... = *vertical-composite.map D D* η
      (*vertical-composite.map D D* $(\eta'\ o\ G\ o\ F)$ $(G\ o\ \varepsilon\ o\ F)$)
  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* $(\eta'\ o\ G)$ $(G\ o\ \varepsilon)$ *o F*)
  using *whisker-right Adj'.ηG.natural-transformation-axioms Adj.F.functor-axioms*
    Adj.Gε.natural-transformation-axioms
  by *fastforce*
also have ... = *vertical-composite.map D D* η $(G\ o\ F)$
  using *Adj'.triangle-G* **by** *simp*
also have ... = η

using *vcomp-ide-dom Adj.GF.functor-axioms Adj.η.natural-transformation-axioms* **by** *simp*
finally show *?thesis* **by** *simp*
qed

17.3.4 Adjunction

The grand unification of everything to do with an adjunction.

locale *adjunction* =
C: *category C* +
D: *category D* +
S: *set-category S* +
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 φC* +
HomD: *hom-functor D S 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 Hom-FopxC.map* +
Hom-DopxG: *set-valued-functor DopxC.comp S 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 φC φD F G Φ Ψ*
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and *S* :: *'s comp* (**infixr** $\cdot_S$ *55*)
and *φC* :: *'c * 'c ⇒ 'c ⇒ 's*
and *φD* :: *'d * 'd ⇒ 'd ⇒ 's*
and *F* :: *'d ⇒ 'c*

and G :: $'c \Rightarrow 'd$
and φ :: $'d \Rightarrow 'c \Rightarrow 'd$
and ψ :: $'c \Rightarrow 'd \Rightarrow 'c$
and η :: $'d \Rightarrow 'd$
and ε :: $'c \Rightarrow 'c$
and Φ :: $'d * 'c \Rightarrow 's$
and Ψ :: $'d * 'c \Rightarrow 's$ +
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 $]\!] \Longrightarrow$
$\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \Phi\Psi.\psi D\ (y,\ G\ x))$

17.4 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 ηo :: $'d \Rightarrow 'd$
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 $\ll g : y' \rightarrow_D y\gg$
shows $\varphi\ y'\ (F\ g) = \eta o\ y \cdot_D g$
using *assms* ηo*-def* φ*-in-hom* [*of* y F y F y]
D.comp-cod-arr [*of* D $(\varphi\ y\ (F\ y))$ g G $(F\ y)$]
φ*-naturality* [*of* F y F y F y g y' y F y]
by *fastforce*

interpretation η: *transformation-by-components D D D.map GF.map* ηo
proof
show $\bigwedge a.\ D.ide\ a \Longrightarrow \ll \eta o\ a : D.map\ a \rightarrow_D GF.map\ a\gg$
using ηo*-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-*ηo [*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* ηo*-def* **by** *simp*

definition $\varepsilon o :: 'c \Rightarrow 'c$
where $\varepsilon o\ x = \psi\ x\ (G\ x)$

lemma εo*-in-hom*:
assumes *C.ide x*
shows $\ll \varepsilon o\ x : F\ (G\ x) \rightarrow_C x\gg$
using *assms C.ide-in-hom* εo*-def* ψ*-in-hom* **by** *force*

lemma ψ*-in-terms-of-*εo:
assumes *C.ide x* **and** $\ll g : y \rightarrow_D G\ x\gg$
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* εo*-def* ψ*-in-hom* [*of* x G x G x]
C.comp-cod-arr [*of* $\psi\ x\ (G\ x) \cdot_C F\ g\ x$]
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 $\ll f : x \rightarrow_C x' \gg$
shows $\psi\ x'\ (G\ f) = f \cdot_C \varepsilon o\ x$
proof (*unfold εo-def*)
 have *0*: $C.ide\ x \wedge C.ide\ x'$ **using** *assms* **by** *auto*
 thus $\psi\ x'\ (G\ f) = f \cdot_C \psi\ x\ (G\ x)$
  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.is-natural-1 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 $\varepsilon.map\ x = \psi\ x\ (G\ x)$
 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›*
proof –
 have *natural-transformation D C F (F o (G o F)) (F o η.map)*
  using *η.natural-transformation-axioms F.natural-transformation-axioms*
   horizontal-composite
  by *fastforce*
 thus *natural-transformation D C F (F o G o F) (F o η.map)*
  using *o-assoc* **by** *metis*
qed

interpretation *εF*: *natural-transformation D C ‹F o G o F› F ‹ε.map o F›*
 using *ε.natural-transformation-axioms F.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.natural-transformation-axioms*

horizontal-composite
by *fastforce*

interpretation $G\varepsilon$: *natural-transformation* C D ‹G *o* F *o* G› G ‹G *o* ε*.map*›
proof –
have *natural-transformation* C D (G *o* (F *o* G)) G (G *o* ε*.map*)
using ε*.natural-transformation-axioms* G*.natural-transformation-axioms*
horizontal-composite
by *fastforce*
thus *natural-transformation* C D (G *o* F *o* G) G (G *o* ε*.map*)
using *o-assoc* **by** *metis*
qed

interpretation $\varepsilon FoF\eta$: *vertical-composite* D C F ‹F *o* G *o* F› F ‹F *o* η*.map*› ‹ε*.map* *o* F› **..**
interpretation $G\varepsilon o\eta G$: *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 $F\ y = \varepsilon o\ (F\ y) \cdot_C F\ (\eta o\ y)$
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 $G\ x = G\ (\varepsilon o\ x) \cdot_D \eta o\ (G\ x)$
using *assms* φ*-in-terms-of-*ηo εo*-def* φ-ψ εo*-in-hom* G*.preserves-ide* D*.ide-in-hom* **by** *metis*

theorem *induces-unit-counit-adjunction*:
shows *unit-counit-adjunction* C D F G η*.map* ε*.map*
proof
show $\varepsilon FoF\eta$*.map* = F
using $\varepsilon FoF\eta$*.is-natural-transformation* $\varepsilon FoF\eta$*.map-simp-ide* *unit-counit-F*
F*.natural-transformation-axioms*
by (*intro NaturalTransformation.eqI*, *auto*)
show $G\varepsilon o\eta G$*.map* = G
using $G\varepsilon o\eta G$*.is-natural-transformation* $G\varepsilon o\eta G$*.map-simp-ide* *unit-counit-G*
G*.natural-transformation-axioms*
by (*intro NaturalTransformation.eqI*, *auto*)
qed

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** $\ll f : F\ y \to_C x\gg$
shows $\varphi\ y\ f = G\ f \cdot_D \eta.map\ y$
using *assms* **by** (*simp add*: φ*-in-terms-of-*ηo)

lemma ψ*-in-terms-of-*ε:
assumes C*.ide* x **and** $\ll g : y \to_D G\ x\gg$
shows $\psi\ x\ g = \varepsilon.map\ x \cdot_C F\ g$

using *assms* **by** (*simp add*: ψ-*in-terms-of*-εo)

definition η :: $'d \Rightarrow 'd$ **where** $\eta \equiv \eta.map$
definition ε :: $'c \Rightarrow 'c$ **where** $\varepsilon \equiv \varepsilon.map$

lemma η-*is-natural-transformation*:
shows *natural-transformation D D D.map GF.map* η
unfolding η-*def* **..**

lemma ε-*is-natural-transformation*:
shows *natural-transformation C C FG.map C.map* ε
unfolding ε-*def* **..**

end

17.5 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* (ε *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*) (ε *x*) $y'\, f = \varphi\ y'\, f$
proof −
interpret εx: *arrow-from-functor D C F* ‹*G x*› *x* ‹ε *x*›
apply *unfold-locales*
using *x* ε.*preserves-hom G.preserves-ide* **by** *auto*
have *1*: $\bigwedge y'\, f.$ *arrow-from-functor D C F y' x f* $\Longrightarrow$
εx.*is-coext* $y'\, f$ ($\varphi\ y'\, f$) $\wedge$ ($\forall g'.\ \varepsilon x$.*is-coext* $y'\, f\, g' \longrightarrow g' = \varphi\ y'\, f$)
proof
fix y' :: $'d$ **and** f :: $'c$
assume *f*: *arrow-from-functor D C F y' x f*
show εx.*is-coext* $y'\, f$ ($\varphi\ y'\, f$)
using *f x* ε-*def* φ-*in-hom* ψ-φ ψ-*in-terms-of*-ε εx.*is-coext-def arrow-from-functor.arrow*
by *metis*
show $\forall g'.\ \varepsilon x$.*is-coext* $y'\, f\, g' \longrightarrow g' = \varphi\ y'\, f$
using εo-*def* ψ-*in-terms-of*-εo *x* ε-*map-simp* φ-ψ εx.*is-coext-def* ε-*def* **by** *simp*
qed
interpret εx: *terminal-arrow-from-functor D C F* ‹*G x*› *x* ‹ε *x*›
apply *unfold-locales* **using** *1* **by** *blast*
show *terminal-arrow-from-functor D C F* (*G x*) *x* (ε *x*) **..**
show $\bigwedge y'\, f.$ *arrow-from-functor D C F y' x f* $\Longrightarrow \varepsilon x$.*the-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*

end

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-initial-arrows-to-functor*:
assumes *y*: *D.ide y*
shows *initial-arrow-to-functor C D G y (F y) (η 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) (η y) x′ g $= \psi\ x'\ g$
proof −
interpret *ηy*: *arrow-to-functor C D G y ‹F y› ‹η y›*
apply *unfold-locales* **using** *y* **by** *auto*
have *1*: $\bigwedge x'\ g.$ *arrow-to-functor C D G y x′ g* $\Longrightarrow$
ηy.is-ext x′ g (ψ x′ g) $\wedge$ ($\forall f'.$ *ηy.is-ext x′ g f′* $\longrightarrow f' = \psi\ x'\ g$)
proof
fix x' :: $'c$ **and** g :: $'d$
assume *g*: *arrow-to-functor C D G y x′ g*
show *ηy.is-ext x′ g (ψ x′ g)*
using *g y ψ-in-hom φ-ψ φ-in-terms-of-η ηy.is-ext-def arrow-to-functor.arrow η-def*
by *metis*
show $\forall f'.$ *ηy.is-ext x′ g f′* $\longrightarrow f' = \psi\ x'\ g$
using *y ηo-def φ-in-terms-of-ηo η-map-simp ψ-φ ηy.is-ext-def η-def* **by** *simp*
qed
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 $\bigwedge x'\ g.$ *arrow-to-functor C D G y x′ g* $\Longrightarrow$ *ηy.the-ext x′ g* $= \psi\ x'\ g$
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

17.6 Unit/Counit Adjunctions Induce Meta-Adjunctions

context *unit-counit-adjunction*
begin

definition φ :: $'d \Rightarrow 'c \Rightarrow 'd$
where $\varphi\ y\ h = G\ h \cdot_D \eta\ y$

definition ψ :: $'c \Rightarrow 'd \Rightarrow 'c$
where $\psi\ x\ h = \varepsilon\ x \cdot_C F\ h$

interpretation *meta-adjunction* $C\ D\ F\ G\ \varphi\ \psi$
proof
fix x :: $'c$ **and** y :: $'d$ **and** f :: $'c$
assume y: $D.ide\ y$ **and** f: $\ll f : F\ y \rightarrow_C x\gg$
show 0: $\ll\varphi\ y\ f : y \rightarrow_D G\ x\gg$
using $f\ y$ *G.preserves-hom* η*.preserves-hom* φ*-def D.ide-in-hom*
by (*metis D.comp-in-homI D.in-homE comp-apply D.map-simp*)
show $\psi\ x\ (\varphi\ y\ f) = f$
proof −
have $\psi\ x\ (\varphi\ y\ f) = (\varepsilon\ x \cdot_C F\ (G\ f)) \cdot_C F\ (\eta\ y)$
using $y\ f$ φ*-def* ψ*-def C.comp-assoc* **by** *auto*
also have ... $= (f \cdot_C \varepsilon\ (F\ y)) \cdot_C F\ (\eta\ y)$
using $y\ f$ ε*.naturality* **by** *auto*
also have ... $= f$
using $y\ f$ $\varepsilon FoF\eta$*.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: $\ll g : y \rightarrow_D G\ x\gg$
show $\ll\psi\ x\ g : F\ y \rightarrow_C x\gg$ **using** $g\ x$ ψ*-def* **by** *fastforce*
show $\varphi\ y\ (\psi\ x\ g) = g$
proof −
have $\varphi\ y\ (\psi\ x\ g) = (G\ (\varepsilon\ x) \cdot_D \eta\ (G\ x)) \cdot_D g$
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* $G\varepsilon o\eta G$*.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: $\ll f : x \rightarrow_C x'\gg$ **and** g: $\ll g : y' \rightarrow_D y\gg$ **and** h: $\ll h : F\ y \rightarrow_C x\gg$
show $\varphi\ y'\ (f \cdot_C h \cdot_C F\ g) = G\ f \cdot_D \varphi\ y\ h \cdot_D g$
using φ*-def* $f\ g\ h$ η*.naturality D.comp-assoc* **by** *fastforce*
qed

theorem *induces-meta-adjunction*:

shows *meta-adjunction* $C\ D\ F\ G\ \varphi\ \psi$ **..**

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 $\eta\ y = \varphi\ y\ (F\ y)$
  using *assms φ-def D.comp-cod-arr* **by** *auto*

lemma *ε-in-terms-of-ψ*:
assumes *C.ide x*
shows $\varepsilon\ x = \psi\ x\ (G\ x)$
  using *assms ψ-def C.comp-arr-dom* **by** *auto*

end

17.7 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* :: $'c \Rightarrow 'd$
where *Go a = (SOME b. ∃ e. terminal-arrow-from-functor D C F b a e)*

definition *εo* :: $'c \Rightarrow 'c$
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 λb. ∃ e. terminal-arrow-from-functor D C F b a e]
      someI-ex [of λe. 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 $f \cdot_C \varepsilon o\ (C.dom\ f)$ along $\varepsilon o\ (C.cod\ f)$.

definition *G* :: $'c \Rightarrow 'd$
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 $\cdot_C$ 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*: $\ll$*G f* : *Go ?x* $\rightarrow_D$ *Go ?x′*$\gg$ ∧ *f* $\cdot_C$ *εo ?x* = *εo ?x′* $\cdot_C$ *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* $\cdot_C$ *εo ?x*)
  using *f xε.arrow* **by** (*unfold-locales*, *auto*)

have *2*: *arrow-from-functor D C F* (*Go ?x'*) *?x''* ($f' \cdot_C$ *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'*) ∧ $f \cdot_C$ *εo ?x* = *εo ?x'* $\cdot_C$ *F* (*G f*)
using *1 x'ε.the-coext-prop* **by** *simp*
have *G f'* = *x''ε.the-coext* (*Go ?x'*) ($f' \cdot_C$ *εo ?x'*)
using *ff' G-def* **by** *auto*
hence *Gf'*: «*G f'* : *Go* (*C.cod f*) $\rightarrow_D$ *Go* (*C.cod f'*)» ∧ $f' \cdot_C$ *εo ?x'* = *εo ?x''* $\cdot_C$ *F* (*G f'*)
using *2 x''ε.the-coext-prop* **by** *simp*
show *G* ($f' \cdot_C f$) = *G f'* $\cdot_D$ *G f*
proof −
have *x''ε.is-coext* (*Go ?x*) (($f' \cdot_C f$) $\cdot_C$ *εo ?x*) (*G f'* $\cdot_D$ *G f*)
proof −
have «*G f'* $\cdot_D$ *G f* : *Go* (*C.dom f*) $\rightarrow_D$ *Go* (*C.cod f'*)» **using** *1 2 Gf Gf'* **by** *auto*
moreover have ($f' \cdot_C f$) $\cdot_C$ *εo ?x* = *εo ?x''* $\cdot_C$ *F* (*G f'* $\cdot_D$ *G f*)
proof −
have ($f' \cdot_C f$) $\cdot_C$ *εo ?x* = $f' \cdot_C f \cdot_C$ *εo ?x*
using *C.comp-assoc* **by** *force*
also have ... = ($f' \cdot_C$ *εo ?x'*) $\cdot_C$ *F* (*G f*)
using *Gf C.comp-assoc* **by** *fastforce*
also have ... = *εo ?x''* $\cdot_C$ *F* (*G f'* $\cdot_D$ *G f*)
using *Gf Gf' C.comp-assoc* **by** *fastforce*
finally show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **using** *x''ε.is-coext-def* **by** *auto*
qed
moreover have *arrow-from-functor D C F* (*Go ?x*) *?x''* (($f' \cdot_C f$) $\cdot_C$ *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*)) ($f \cdot_C$ *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 $\ll \varepsilon o\ x : GF.map\ x \rightarrow_C C.map\ x \gg$
proof −
interpret *terminal-arrow-from-functor D C F* ‹*Go x*› *x* ‹εo *x*›
using *x Go-*ε*o-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 $\varepsilon o\ (C.cod\ f) \cdot_C GF.map\ f = C.map\ f \cdot_C \varepsilon o\ (C.dom\ f)$
proof −
let $?x = C.dom\ f$
let $?x' = C.cod\ f$
interpret $x\varepsilon$: *terminal-arrow-from-functor D C F* ‹*Go ?x*› *?x* ‹εo *?x*›
using *f Go-*ε*o-terminal ex-terminal-arrow* **by** *simp*
interpret $x'\varepsilon$: *terminal-arrow-from-functor D C F* ‹*Go ?x'*› *?x'* ‹εo *?x'*›
using *f Go-*ε*o-terminal ex-terminal-arrow* **by** *simp*
have *1*: *arrow-from-functor D C F* (*Go ?x*) *?x'* ($C\ f$ (εo *?x*))
using *f* $x\varepsilon$*.arrow* **by** (*unfold-locales*, *auto*)
have $G\ f = x'\varepsilon$*.the-coext* (*Go ?x*) ($f \cdot_C \varepsilon o$ *?x*)
using *f G-simp* **by** *blast*
hence $x'\varepsilon$*.is-coext* (*Go ?x*) ($f \cdot_C \varepsilon o$ *?x*) ($G\ f$)
using *1* $x'\varepsilon$*.the-coext-prop* $x'\varepsilon$*.is-coext-def* **by** *auto*
thus *?thesis*
using *f* $x'\varepsilon$*.is-coext-def* **by** *simp*
qed
qed

definition ψ
where $\psi\ x\ h = C\ (\varepsilon.map\ x)\ (F\ h)$

lemma ψ*-in-hom*:
assumes $C.ide\ x$ **and** $\ll g : y \rightarrow_D G\ x \gg$
shows $\ll \psi\ x\ g : F\ y \rightarrow_C x \gg$
unfolding ψ*-def* **using** *assms* ε*.maps-ide-in-hom* **by** *auto*

lemma ψ*-natural*:
assumes f: $\ll f : x \rightarrow_C x' \gg$ **and** g: $\ll g : y' \rightarrow_D y \gg$ **and** h: $\ll h : y \rightarrow_D G\ x \gg$
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 $f\ g\ h\ C.comp\text{-}assoc$ **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: $\ll g : y \rightarrow_D G\ x \gg$
shows *arrow-from-functor.is-coext* $D\ C\ F\ (G\ x)\ (\varepsilon.map\ x)\ y\ (\psi\ x\ g)\ g$
proof –
interpret $x\varepsilon$: *arrow-from-functor* $D\ C\ F\ \langle G\ x\rangle\ x\ \langle\varepsilon.map\ x\rangle$
using x ε*.maps-ide-in-hom* **by** (*unfold-locales*, *auto*)
show $x\varepsilon$*.is-coext* $y\ (\psi\ x\ g)\ g$
using $x\ g$ ψ*-def* $x\varepsilon$*.is-coext-def* *G-ide* **by** *blast*
qed

lemma ψ*-invertible*:
assumes y: $D.ide\ y$ **and** f: $\ll f : F\ y \rightarrow_C x \gg$
shows $\exists!g.\ \ll g : y \rightarrow_D G\ x \gg \wedge \psi\ x\ g = f$
proof
have x: $C.ide\ x$ **using** f **by** *auto*
interpret $x\varepsilon$: *terminal-arrow-from-functor* $D\ C\ F\ \langle Go\ x\rangle\ x\ \langle\varepsilon o\ x\rangle$
using x *ex-terminal-arrow* *Go-*ε*o-terminal* **by** *auto*
have *1*: *arrow-from-functor* $D\ C\ F\ y\ x\ f$
using $y\ f$ **by** (*unfold-locales*, *auto*)
let $?g = x\varepsilon.the\text{-}coext\ y\ f$
have $\psi\ x\ ?g = f$
using *1* $x\ y$ ψ*-def* $x\varepsilon$*.the-coext-prop* *G-ide* ψ*-inverts-coext* $x\varepsilon$*.is-coext-def* **by** *simp*
thus $\ll ?g : y \rightarrow_D G\ x \gg \wedge \psi\ x\ ?g = f$
using *1* x $x\varepsilon$*.the-coext-prop* *G-ide* **by** *simp*
show $\bigwedge g'.\ \ll g' : y \rightarrow_D G\ x \gg \wedge \psi\ x\ g' = f \Longrightarrow g' = ?g$
using *1* $x\ y$ ψ*-inverts-coext* *G-ide* $x\varepsilon$*.the-coext-unique* **by** *force*
qed

definition φ
where $\varphi\ y\ f = (THE\ g.\ \ll g : y \rightarrow_D G\ (C.cod\ f) \gg \wedge \psi\ (C.cod\ f)\ g = f)$

lemma φ*-in-hom*:
assumes $D.ide\ y$ **and** $\ll f : F\ y \rightarrow_C x \gg$
shows $\ll \varphi\ y\ f : y \rightarrow_D G\ x \gg$
using *assms* ψ*-invertible* φ*-def* *theI'* [*of* $\lambda g.\ \ll g : y \rightarrow_D G\ x \gg \wedge \psi\ x\ g = f$]
by *auto*

lemma φ*-*ψ:
assumes $C.ide\ x$ **and** $\ll g : y \rightarrow_D G\ x \gg$
shows $\varphi\ y\ (\psi\ x\ g) = g$
proof –
have $C.cod\ (\psi\ x\ g) = x$
using *assms* ψ*-in-hom* **by** *auto*

hence $\varphi\ y\ (\psi\ x\ g) = (\mathit{THE}\ g'.\ «g' : y \rightarrow_D\ G\ x» \wedge \psi\ x\ g' = \psi\ x\ g)$
using *φ-def* **by** *auto*
moreover have $\exists!g'.\ «g' : y \rightarrow_D\ G\ x» \wedge \psi\ x\ g' = \psi\ x\ g$
using *assms ψ-in-hom ψ-invertible D.ide-dom* **by** *blast*
moreover have $«g : y \rightarrow_D\ G\ x» \wedge \psi\ x\ g = \psi\ x\ g$
using *assms(2)* **by** *auto*
ultimately show $\varphi\ y\ (\psi\ x\ g) = g$ **by** *auto*
qed

lemma *ψ-φ*:
assumes *D.ide y* **and** $«f : F\ y \rightarrow_C\ x»$
shows $\psi\ x\ (\varphi\ y\ f) = f$
using *assms ψ-invertible φ-def theI'* $[of\ \lambda g.\ «g : y \rightarrow_D\ G\ x» \wedge \psi\ x\ g = f]$
by *auto*

lemma *φ-natural*:
assumes $«f : x \rightarrow_C\ x'»$ **and** $«g : y' \rightarrow_D\ y»$ **and** $«h : F\ y \rightarrow_C\ x»$
shows $\varphi\ y'\ (f \cdot_C\ h \cdot_C\ F\ g) = (G\ f \cdot_D\ \varphi\ y\ h) \cdot_D\ g$
proof –
have $C.ide\ x' \wedge D.ide\ y \wedge D.in\text{-}hom\ (\varphi\ y\ h)\ y\ (G\ x)$
using *assms φ-in-hom* **by** *auto*
thus *?thesis*
using *assms D.comp-in-homI G.preserves-hom ψ-natural* $[of\ f\ x\ x'\ g\ y'\ y\ \varphi\ y\ h]$ *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*, *simp-all*)

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* λu. *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 ηo $(D.cod\ g) \cdot_D g$ along ηo $(D.dom\ g)$.

definition F :: $'d \Rightarrow 'c$
where $F\ g = ($*if* $D.arr\ g$ *then*
initial-arrow-to-functor.the-ext $C\ D\ G\ (Fo\ (D.dom\ g))\ (\eta o\ (D.dom\ g))$
$(Fo\ (D.cod\ g))\ (\eta o\ (D.cod\ g) \cdot_D g)$
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 initial-arrows-exist Fo-*η*o-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 $\eta o\ y = D\ (\eta o\ (D.cod\ y))\ y$
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 $\neg D.arr\ g$
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 initial-arrows-exist Fo-*η*o-initial* **by** *simp*
interpret $y'\eta$: *initial-arrow-to-functor C D G ?y'* ‹*Fo ?y'*› ‹ηo *?y'*›
using *g initial-arrows-exist Fo-*η*o-initial* **by** *simp*
have *1*: *arrow-to-functor C D G ?y* (*Fo ?y'*) (*D* (ηo *?y'*) *g*)
using *g* $y'\eta$*.arrow* **by** (*unfold-locales*, *auto*)
have $F\ g = y\eta$*.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* $y\eta$*.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* $C\ D\ G\ ?y\ \langle Fo\ ?y\rangle\ \langle \eta o\ ?y\rangle$
  using *g initial-arrows-exist Fo-ηo-initial* **by** *simp*
interpret $y'\eta$: *initial-arrow-to-functor* $C\ D\ G\ ?y'\ \langle Fo\ ?y'\rangle\ \langle \eta o\ ?y'\rangle$
  using *g initial-arrows-exist Fo-ηo-initial* **by** *simp*
interpret $y''\eta$: *initial-arrow-to-functor* $C\ D\ G\ ?y''\ \langle Fo\ ?y''\rangle\ \langle \eta o\ ?y''\rangle$
  using g' *initial-arrows-exist Fo-ηo-initial* **by** *auto*
have *1*: *arrow-to-functor* $C\ D\ G\ ?y\ (Fo\ ?y')\ (\eta o\ ?y' \cdot_D g)$
  using $g\ y'\eta.arrow$ **by** (*unfold-locales*, *auto*)
have $F\ g = y\eta.the\text{-}ext\ (Fo\ ?y')\ (\eta o\ ?y' \cdot_D g)$
  using *g F-def* **by** *simp*
hence *Fg*: $\ll F\ g : Fo\ ?y \rightarrow_C Fo\ ?y'\gg \wedge \eta o\ ?y' \cdot_D g = G\ (F\ g) \cdot_D \eta o\ ?y$
  using *1* $y\eta.the\text{-}ext\text{-}prop$ **by** *simp*
have *2*: *arrow-to-functor* $C\ D\ G\ ?y'\ (Fo\ ?y'')\ (\eta o\ ?y'' \cdot_D g')$
  using $g'\ y''\eta.arrow$ **by** (*unfold-locales*, *auto*)
have $F\ g' = y'\eta.the\text{-}ext\ (Fo\ ?y'')\ (\eta o\ ?y'' \cdot_D g')$
  using g' *F-def* **by** *auto*
hence Fg': $\ll F\ g' : Fo\ ?y' \rightarrow_C Fo\ ?y''\gg \wedge \eta o\ ?y'' \cdot_D g' = G\ (F\ g') \cdot_D \eta o\ ?y'$
  using *2* $y'\eta.the\text{-}ext\text{-}prop$ **by** *simp*
show $F\ (g' \cdot_D g) = F\ g' \cdot_C F\ g$
proof −
  have $y\eta.is\text{-}ext\ (Fo\ ?y'')\ (\eta o\ ?y'' \cdot_D g' \cdot_D g)\ (F\ g' \cdot_C F\ g)$
  proof −
    have $\ll F\ g' \cdot_C F\ g : Fo\ ?y \rightarrow_C Fo\ ?y''\gg$ **using** *1 2 Fg* Fg' **by** *auto*
    moreover have $\eta o\ ?y'' \cdot_D g' \cdot_D g = G\ (F\ g' \cdot_C F\ g) \cdot_D \eta o\ ?y$
    proof −
      have $\eta o\ ?y'' \cdot_D g' \cdot_D g = (G\ (F\ g') \cdot_D \eta o\ ?y') \cdot_D g$
        using $Fg'\ g\ g'\ y''\eta.arrow$ **by** (*metis D.comp-assoc*)
      also have ... $=\ G\ (F\ g') \cdot_D \eta o\ ?y' \cdot_D g$
        using *D.comp-assoc* **by** *fastforce*
      also have ... $= G\ (F\ g' \cdot_C F\ g) \cdot_D \eta o\ ?y$
        using *Fg* Fg' *D.comp-assoc* **by** *fastforce*
      finally show *?thesis* **by** *auto*
    qed
    ultimately show *?thesis* **using** $y\eta.is\text{-}ext\text{-}def$ **by** *auto*
  qed
  moreover have *arrow-to-functor* $C\ D\ G\ ?y\ (Fo\ ?y'')\ (\eta o\ ?y'' \cdot_D g' \cdot_D g)$
    using $g\ g'\ y''\eta.arrow$ **by** (*unfold-locales*, *auto*)
  ultimately show *?thesis*
    using $g\ g'$ *F-def* $y\eta.the\text{-}ext\text{-}unique$ *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 $F\ g = initial\text{-}arrow\text{-}to\text{-}functor.the\text{-}ext\ C\ D\ G\ (Fo\ (D.dom\ g))\ (\eta o\ (D.dom\ g))$
$(Fo\ (D.cod\ g))\ (\eta o\ (D.cod\ g) \cdot_D g)$
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 $\langle\langle \eta o\ y : D.map\ y \rightarrow_D FG.map\ y \rangle\rangle$
proof −
interpret *initial-arrow-to-functor C D G y* ‹$Fo\ y$› ‹$\eta o\ y$›
using y *Fo-*η*o-initial initial-arrows-exist* **by** *simp*
show *?thesis* **using** y *F-ide arrow* **by** *auto*
qed
next
fix $g :: 'd$
assume g: $D.arr\ g$
show $\eta o\ (D.cod\ g) \cdot_D D.map\ g = FG.map\ g \cdot_D \eta o\ (D.dom\ g)$
proof −
let $?y = D.dom\ g$
let $?y' = D.cod\ g$
interpret $y\eta$: *initial-arrow-to-functor C D G ?y* ‹$Fo\ ?y$› ‹$\eta o\ ?y$›
using g *Fo-*η*o-initial initial-arrows-exist* **by** *simp*
interpret $y'\eta$: *initial-arrow-to-functor C D G ?y'* ‹$Fo\ ?y'$› ‹$\eta o\ ?y'$›
using g *Fo-*η*o-initial initial-arrows-exist* **by** *simp*
have *arrow-to-functor C D G ?y* $(Fo\ ?y')\ (\eta o\ ?y' \cdot_D g)$
using g $y'\eta.arrow$ **by** (*unfold-locales*, *auto*)
moreover have $F\ g = y\eta.the\text{-}ext\ (Fo\ ?y')\ (\eta o\ ?y' \cdot_D g)$
using g *F-simp* **by** *blast*
ultimately have $y\eta.is\text{-}ext\ (Fo\ ?y')\ (\eta o\ ?y' \cdot_D g)\ (F\ g)$
using $y\eta.the\text{-}ext\text{-}prop$ $y\eta.is\text{-}ext\text{-}def$ **by** *auto*
thus *?thesis*
using g $y\eta.is\text{-}ext\text{-}def$ **by** *simp*
qed
qed

definition φ
where $\varphi\ y\ h = D\ (G\ h)\ (\eta.map\ y)$

lemma φ*-in-hom*:

assumes y: $D.ide\ y$ **and** f: $\ll f : F\ y \rightarrow_C x\gg$
shows $\ll\varphi\ y\ f : y \rightarrow_D G\ x\gg$
  unfolding *φ-def* **using** *assms η.maps-ide-in-hom* **by** *auto*

lemma *φ-natural*:
assumes f: $\ll f : x \rightarrow_C x'\gg$ **and** g: $\ll g : y' \rightarrow_D y\gg$ **and** h: $\ll h : F\ y \rightarrow_C x\gg$
shows $\varphi\ y'\ (f \cdot_C h \cdot_C F\ g) = (G\ f \cdot_D \varphi\ y\ h) \cdot_D g$
proof –
  have $(G\ f \cdot_D \varphi\ y\ h) \cdot_D g = (G\ f \cdot_D G\ h \cdot_D \eta.map\ y) \cdot_D g$
    unfolding *φ-def* **by** *auto*
  also have ... $= (G\ f \cdot_D G\ h) \cdot_D \eta.map\ y \cdot_D g$
    using *D.comp-assoc* **by** *fastforce*
  also have ... $= G\ (f \cdot_C h) \cdot_D G\ (F\ g) \cdot_D \eta.map\ y'$
    using *f g h η.naturality* **by** *fastforce*
  also have ... $= (G\ (f \cdot_C h) \cdot_D G\ (F\ g)) \cdot_D \eta.map\ y'$
    using *D.comp-assoc* **by** *fastforce*
  also have ... $= G\ (f \cdot_C h \cdot_C F\ g) \cdot_D \eta.map\ y'$
    using *f g h D.comp-assoc* **by** *fastforce*
  also have ... $= \varphi\ y'\ (f \cdot_C h \cdot_C F\ g)$
    unfolding *φ-def* **by** *auto*
  finally show *?thesis* **by** *auto*
qed

lemma *φ-inverts-ext*:
assumes y: $D.ide\ y$ **and** f: $\ll f : F\ y \rightarrow_C x\gg$
shows *arrow-to-functor.is-ext C D G* $(F\ y)$ $(\eta.map\ y)$ x $(\varphi\ y\ f)$ 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* $(\varphi\ y\ f)$ f
    using *f y φ-def yη.is-ext-def F-ide* **by** (*unfold-locales*, *auto*)
qed

lemma *φ-invertible*:
assumes x: $C.ide\ x$ **and** g: $\ll g : y \rightarrow_D G\ x\gg$
shows $\exists!f.\ \ll f : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f = g$
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 initial-arrows-exist 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 $\varphi\ y$ *?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 $\ll$*?f* $: F\ y \rightarrow_C x\gg$
    using *1 y yη.the-ext-prop F-ide* **by** *simp*
  ultimately show $\ll$*?f* $: F\ y \rightarrow_C x\gg \wedge \varphi\ y$ *?f* $= g$ **by** *auto*
  show $\bigwedge f'.\ \ll f' : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f' = g \Longrightarrow f' =$ *?f*

using *1 y φ-inverts-ext yη.the-ext-unique F-ide* **by** *force*
qed

definition ψ
where $\psi\ x\ g = (\mathit{THE}\ f.\ \ll f : F\ (D.dom\ g) \rightarrow_C x\gg \wedge \varphi\ (D.dom\ g)\ f = g)$

lemma *ψ-in-hom*:
assumes $C.ide\ x$ **and** $\ll g : y \rightarrow_D G\ x\gg$
shows $C.in\text{-}hom\ (\psi\ x\ g)\ (F\ y)\ x$
using *assms φ-invertible ψ-def theI′* $[of\ \lambda f.\ \ll f : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f = g]$
by *auto*

lemma *ψ-φ*:
assumes $D.ide\ y$ **and** $\ll f : F\ y \rightarrow_C x\gg$
shows $\psi\ x\ (\varphi\ y\ f) = f$
proof −
have $D.dom\ (\varphi\ y\ f) = y$ **using** *assms φ-in-hom* **by** *blast*
hence $\psi\ x\ (\varphi\ y\ f) = (\mathit{THE}\ f'.\ \ll f' : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f' = \varphi\ y\ f)$
using *ψ-def* **by** *auto*
moreover have $\exists!f'.\ \ll f' : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f' = \varphi\ y\ f$
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** $\ll g : y \rightarrow_D G\ x\gg$
shows $\varphi\ y\ (\psi\ x\ g) = g$
using *assms φ-invertible ψ-def theI′* $[of\ \lambda f.\ \ll f : F\ y \rightarrow_C x\gg \wedge \varphi\ y\ f = g]$
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

17.8 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 category *SetCat* whose element type is the disjoint sum $'c + 'd$ of the arrow types of C and D.

context *meta-adjunction*
begin

definition *inC* :: $'c \Rightarrow ('c+'d)$ *setcat.arr*
where *inC* ≡ *SetCat.UP o Inl*

definition *inD* :: *'d* ⇒ (*'c*+*'d*) *setcat.arr*
where *inD* ≡ *SetCat.UP o Inr*

interpretation *S*: *set-category* ‹*SetCat.comp* :: (*'c*+*'d*) *setcat.arr comp*›
  using *SetCat.is-set-category* **by** *auto*
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* ‹*SetCat.comp* :: (*'c*+*'d*) *setcat.arr comp*› ‹λ-. *inC*›
  apply *unfold-locales*
  unfolding *inC-def* **using** *SetCat.UP-mapsto*
   apply *auto*[*1*]
  using *SetCat.inj-UP*
  by (*metis injD inj-Inl inj-compose inj-on-def*)
interpretation *HomD*: *hom-functor D* ‹*SetCat.comp* :: (*'c*+*'d*) *setcat.arr comp*› ‹λ-. *inD*›
  apply *unfold-locales*
  unfolding *inD-def* **using** *SetCat.UP-mapsto*
   apply *auto*[*1*]
  using *SetCat.inj-UP*
  by (*metis injD inj-Inr inj-compose inj-on-def*)
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-FopxC*: *composite-functor DopxC.comp CopxC.comp SetCat.comp*
          FopxC.map HomC.map **..**
interpretation *Hom-DopxG*: *composite-functor DopxC.comp DopxD.comp SetCat.comp*
          DopxG.map HomD.map **..**

lemma *inC-ψ* [*simp*]:
assumes *C.ide b* **and** *C.ide a* **and** *x* ∈ *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* ∈ *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* ∘ (λh. *snd gf* $\cdot_C$ *h* $\cdot_C$ *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* ∘ (λh. *G* (*snd gf*) $\cdot_D$ *h* $\cdot_D$ *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 «Φ*o yx* : *Hom-FopxC.map yx* $\rightarrow_S$ *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*))
$\rightarrow_S$ *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 SetCat.comp*
                                        Hom-FopxC.map Hom-DopxG.map Φo
proof
  fix *yx*
  assume *yx*: *DopxC.ide yx*
  show «*Φo yx* : *Hom-FopxC.map yx* $\rightarrow_S$ *Hom-DopxG.map yx*»
    using *yx Φo-in-hom* **by** *auto*
  next
  fix *gf*
  assume *gf*: *DopxC.arr gf*
  show *SetCat.comp* (*Φo* (*DopxC.cod gf*)) (*Hom-FopxC.map gf*)
          = *SetCat.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* (λ*h*. *?f* $\cdot_C$ *h* $\cdot_C$ *?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* (λ*h*. *?Gf* $\cdot_D$ *h* $\cdot_D$ *?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) ∧
        SetCat.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)))
proof –
  have S.seq (Φo (DopxC.cod gf)) (Hom-FopxC.map gf)
    using gf 1 2 Φo-in-hom Hom-FopxC.preserves-hom by (intro S.seqI', auto)
  thus ?thesis
    using S.comp-mkArr 1 2 by metis
qed
have 6: SetCat.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)))
proof –
  have S.seq (Hom-DopxG.map gf) (Φo (DopxC.dom gf))
    using gf 3 4 S.arr-mkArr S.cod-mkArr S.dom-mkArr by (intro S.seqI; metis)
  thus ?thesis
    using 3 4 S.comp-mkArr by metis
qed
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' (HomC.ψ (?Fy', ?x') (inC (?f ·C HomC.ψ (?Fy, ?x) h ·C ?Fg))))
        by simp
      also have ... = 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))
      proof -
        have «?f : C.dom ?f → C.cod ?f»
          using gf by auto
        moreover have «?g : D.dom ?g →D D.cod ?g»
          using gf by auto
        ultimately show ?thesis
          using gf ψh φ-in-hom G.preserves-hom C.in-homE D.in-homE
              φ-naturality [of ?f ?x ?x' ?g ?y' ?y HomC.ψ (?Fy, ?x) h]
          by simp
      qed
      also have ... =
          inD (D ?Gf (D (HomD.ψ (?y, ?Gx) (inD (φ ?y (HomC.ψ (?Fy, ?x) h)))) ?g))
        using gf ψh φ-in-hom by simp
      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
        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)))
proof (intro S.mkArr-eqI')
  show S.arr (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))))
    using 5 by metis
  show ⋀t. t ∈ 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))) t
            = ((inD o (λh. ?Gf ·D h ·D ?g) o HomD.ψ (?y, ?Gx))
                o (inD o φ ?y o HomC.ψ (?Fy, ?x))) t
    using 7 restrict-apply by fast
qed
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 fastforce
moreover have S.ide (SetCat.comp (Ψo yx) (Φ.map yx))
proof -
  have SetCat.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 by fastforce
  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
      SetCat.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 by simp
```

qed
moreover have $S.ide$ $(SetCat.comp\ (\Phi.map\ yx)\ (\Psi o\ yx))$
proof −
have $SetCat.comp\ (\Phi.map\ yx)\ (\Psi o\ yx) =$
$S.mkArr\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))$
$((inD\ o\ \varphi\ (fst\ yx)\ o\ HomC.\psi\ (F\ (fst\ yx),\ snd\ yx))$
$o\ (inC\ o\ \psi\ (snd\ yx)\ o\ HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))))$
using *1 2 3 S.comp-mkArr antipar* **by** *fastforce*
also
have ... $= S.mkArr\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))$
$(\lambda x.\ x)$
proof −
have
$S.mkArr\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))\ (\lambda x.\ x)$
$= ...$
proof
show
$S.arr\ (S.mkArr\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))\ (HomD.set\ (fst\ yx,\ G\ (snd\ yx)))$
$(\lambda x.\ x))$
using *yx HomD.set-subset-Univ* **by** *simp*
show $\bigwedge x.\ x \in (HomD.set\ (fst\ yx,\ G\ (snd\ yx))) \Longrightarrow$
$x = ((inD\ o\ \varphi\ (fst\ yx)\ o\ HomC.\psi\ (F\ (fst\ yx),\ snd\ yx))$
$o\ (inC\ o\ \psi\ (snd\ yx)\ o\ HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))))\ x$
proof −
fix x
assume x: $x \in HomD.set\ (fst\ yx,\ G\ (snd\ yx))$
have $((inD\ o\ \varphi\ (fst\ yx)\ o\ HomC.\psi\ (F\ (fst\ yx),\ snd\ yx))$
$o\ (inC\ o\ \psi\ (snd\ yx)\ o\ HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))))\ x$
$= inD\ (\varphi\ (fst\ yx)\ (HomC.\psi\ (F\ (fst\ yx),\ snd\ yx)$
$(inC\ (\psi\ (snd\ yx)\ (HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))\ x)))))$
by *simp*
also have ... $= inD\ (\varphi\ (fst\ yx)\ (\psi\ (snd\ yx)\ (HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))\ x)))$
proof −
have $\ll\psi\ (snd\ yx)\ (HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))\ x) : F\ (fst\ yx) \rightarrow snd\ yx\gg$
using x yx $HomD.\psi$*-mapsto* [*of fst yx G* $(snd\ yx)$] ψ*-in-hom* **by** *auto*
thus *?thesis* **by** *simp*
qed
also have ... $= inD\ (HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))\ x)$
using x yx $HomD.\psi$*-mapsto* [*of fst yx G* $(snd\ yx)$] φ-ψ **by** *force*
also have ... $= x$ **using** x yx inD-ψ **by** *simp*
finally show $x = ((inD\ o\ \varphi\ (fst\ yx)\ o\ HomC.\psi\ (F\ (fst\ yx),\ snd\ yx))$
$o\ (inC\ o\ \psi\ (snd\ yx)\ o\ HomD.\psi\ (fst\ yx,\ G\ (snd\ yx))))\ x$
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
    SetCat.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* **by** *simp*
 qed
 ultimately show *?thesis* **by** *auto*
qed

interpretation Φ: *natural-isomorphism* *DopxC.comp* *SetCat.comp*
    Hom-FopxC.map *Hom-DopxG.map* Φ.*map*
 apply (*unfold-locales*) **using** Φ-*inv* **by** *blast*

interpretation Ψ: *inverse-transformation* *DopxC.comp* *SetCat.comp*
    Hom-FopxC.map *Hom-DopxG.map* Φ.*map* **..**

interpretation ΦΨ: *inverse-transformations* *DopxC.comp* *SetCat.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 *φC* **where** *φC* ≡ λ-. *inC*
abbreviation *HomD* **where** *HomD* ≡ *HomD.map*
abbreviation *φD* **where** *φD* ≡ λ-. *inD*

theorem *induces-hom-adjunction*: *hom-adjunction* *C* *D* *SetCat.comp* *φC* *φD* *F* *G* Φ Ψ
 using *F.is-extensional* **by** (*unfold-locales*, *auto*)

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* *SetCat.comp*
    Hom-FopxC.map *Hom-DopxG.map* Φ.*map* **..**

interpretation Ψ: *set-valued-transformation* *DopxC.comp* *SetCat.comp*
    Hom-DopxG.map *Hom-FopxC.map* Ψ.*map* **..**

lemma *φ-in-terms-of-*Φ′:
assumes *y*: *D.ide* *y* **and** *f*: «*f*: *F* *y* $\rightarrow_C$ *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 *1*: *S.arr* (Φ (*y*, *x*)) **using** *x* *y* **by** *fastforce*

have *2*: Φ (y, x) = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*))
(*inD o* φ *y o HomC.*ψ (*F y*, *x*))
using *x y* Φ*o-def* **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*))
using *1 2* **by** *simp*
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* $\rightarrow_D$ *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 *1*: *S.arr* (Ψ (*y*, *x*))
using *x y* Ψ*.preserves-reflects-arr* [*of* (*y*, *x*)] **by** *simp*
have *2*: Ψ (*y*, *x*) = *S.mkArr* (*HomD.set* (*y*, *G x*)) (*HomC.set* (*F y*, *x*))
(*inC o* ψ *x o HomD.*ψ (*y*, *G x*))
using *x y* Ψ*-simp* **by** *force*
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*))
using *1 2* **by** *simp*
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

17.9 Hom-Adjunctions Induce Meta-Adjunctions

context *hom-adjunction*
begin

definition φ :: $'d \Rightarrow 'c \Rightarrow 'd$
where
φ *y h* = (*HomD.*ψ (*y*, *G* (*C.cod h*)) *o* Φ*.FUN* (*y*, *C.cod h*) *o* φ*C* (*F y*, *C.cod h*)) *h*

definition ψ :: $'c \Rightarrow 'd \Rightarrow 'c$
where
ψ *x h* = (*HomC.*ψ (*F* (*D.dom h*), *x*) *o* Ψ*.FUN* (*D.dom h*, *x*) *o* φ*D* (*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 ($\lambda h.$ *snd gf* $\cdot_C$ *h* $\cdot_C$ *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 ($\lambda h.$ *G* (*snd gf*) $\cdot_D$ *h* $\cdot_D$ *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** $\ll f : F\ y \rightarrow_C x\gg$
shows Φ.*FUN* (*y*, *x*) $\in$ *HomC.set* (*F y*, *x*) $\rightarrow$ *HomD.set* (*y*, *G x*)
proof –
have *S.arr* (Φ (*y*, *x*)) $\wedge$ Φ.*DOM* (*y*, *x*) = *HomC.set* (*F y*, *x*) $\wedge$
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* $\in$ *C.hom* (*F y*) *x* $\rightarrow$ *D.hom y* (*G x*)
proof
fix *h*
assume *h*: *h* $\in$ *C.hom* (*F y*) *x*
hence *1*: $\ll h : F\ y \rightarrow_C x\gg$ **by** *simp*
show φ *y h* $\in$ *D.hom y* (*G x*)
proof –
have φC (*F y*, *x*) *h* $\in$ *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*) $\in$ *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 $S.arr\ (\Phi\ (y,\ x))$
and $\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))$
proof −
show *1*: $S.arr\ (\Phi\ (y,\ x))$ **using** *assms* **by** *auto*
hence $\Phi\ (y,\ x) = S.mkArr\ (\Phi.DOM\ (y,\ x))\ (\Phi.COD\ (y,\ x))\ (\Phi.FUN\ (y,\ x))$
using *S.mkArr-Fun* **by** *metis*
also have ... $= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))\ (\Phi.FUN\ (y,\ x))$
using *assms HomC.set-map HomD.set-map* **by** *fastforce*
also have ... $= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))$
proof (*intro S.mkArr-eqI'*)
show $S.arr\ (S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))\ (\Phi.FUN\ (y,\ x)))$
using *1 calculation* **by** *argo*
show $\bigwedge h.\ h \in HomC.set\ (F\ y,\ x) \Longrightarrow$
$\Phi.FUN\ (y,\ x)\ h = (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))\ h$
proof −
fix h
assume h: $h \in HomC.set\ (F\ y,\ x)$
hence $\ll \psi C\ (F\ y,\ x)\ h : F\ y \rightarrow_C x\gg$
using *assms HomC.ψ-mapsto [of F y x]* **by** *auto*
hence $(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ HomC.\psi\ (F\ y,\ x))\ h =$
$\varphi D\ (y,\ G\ x)\ (\psi D\ (y,\ G\ x)\ (\Phi.FUN\ (y,\ x)\ (\varphi C\ (F\ y,\ x)\ (\psi C\ (F\ y,\ x)\ h))))$
using *h φ-def* **by** *auto*
also have ... $= \varphi D\ (y,\ G\ x)\ (\psi D\ (y,\ G\ x)\ (\Phi.FUN\ (y,\ x)\ h))$
using *assms h HomC.φ-ψ Φ-Fun-mapsto* **by** *simp*
also have ... $= \Phi.FUN\ (y,\ x)\ h$
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 $\Phi.FUN\ (y,\ x)\ h = (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))\ h$ **by** *auto*
qed
qed
finally show $\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))$
by *force*
qed

lemma *Ψ-Fun-mapsto*:
assumes $C.ide\ x$ **and** $\ll g : y \rightarrow_D G\ x\gg$
shows $\Psi.FUN\ (y,\ x) \in HomD.set\ (y,\ G\ x) \rightarrow HomC.set\ (F\ y,\ x)$
proof −
have $S.arr\ (\Psi\ (y,\ x)) \wedge \Psi.COD\ (y,\ x) = HomC.set\ (F\ y,\ x) \wedge$
$\Psi.DOM\ (y,\ x) = HomD.set\ (y,\ G\ x)$
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 $\psi\ x \in D.hom\ y\ (G\ x) \rightarrow C.hom\ (F\ y)\ x$
proof
  fix h
  assume h: $h \in D.hom\ y\ (G\ x)$
  hence *1*: $\ll h : y \rightarrow_D G\ x \gg$ **by** *auto*
  show $\psi\ x\ h \in C.hom\ (F\ y)\ x$
  proof –
    have $\varphi D\ (y, G\ x)\ h \in HomD.set\ (y, G\ x)$
      using *x h 1 HomD.φ-mapsto* [*of y G x*] **by** *fastforce*
    hence $\Psi.FUN\ (y, x)\ (\varphi D\ (y, G\ x)\ h) \in HomC.set\ (F\ y, x)$
      using *h x Ψ-Fun-mapsto* **by** *auto*
    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 $S.arr\ (\Psi\ (y, x))$
and $\Psi\ (y, x) = S.mkArr\ (HomD.set\ (y, G\ x))\ (HomC.set\ (F\ y, x))$
$(\varphi C\ (F\ y, x)\ o\ \psi\ x\ o\ \psi D\ (y, G\ x))$
proof –
  show *1*: $S.arr\ (\Psi\ (y, x))$ **using** *assms* **by** *auto*
  hence $\Psi\ (y, x) = S.mkArr\ (\Psi.DOM\ (y, x))\ (\Psi.COD\ (y, x))\ (\Psi.FUN\ (y, x))$
    using *S.mkArr-Fun* **by** *metis*
  also have ... $= S.mkArr\ (HomD.set\ (y, G\ x))\ (HomC.set\ (F\ y, x))\ (\Psi.FUN\ (y, x))$
    using *assms HomC.set-map HomD.set-map* **by** *auto*
  also have ... $= S.mkArr\ (HomD.set\ (y, G\ x))\ (HomC.set\ (F\ y, x))$
  $(\varphi C\ (F\ y, x)\ o\ \psi\ x\ o\ \psi D\ (y, G\ x))$
  proof (*intro S.mkArr-eqI′*)
    show $S.arr\ (S.mkArr\ (HomD.set\ (y, G\ x))\ (HomC.set\ (F\ y, x))\ (\Psi.FUN\ (y, x)))$
      using *1 calculation* **by** *argo*
    show $\bigwedge h.\ h \in HomD.set\ (y, G\ x) \Longrightarrow$
    $\Psi.FUN\ (y, x)\ h = (\varphi C\ (F\ y, x)\ o\ \psi\ x\ o\ \psi D\ (y, G\ x))\ h$
    proof –
      fix h
      assume h: $h \in HomD.set\ (y, G\ x)$
      hence $\ll \psi D\ (y, G\ x)\ h : y \rightarrow_D G\ x \gg$
        using *assms HomD.ψ-mapsto* [*of y G x*] **by** *auto*
      hence $(\varphi C\ (F\ y, x)\ o\ \psi\ x\ o\ HomD.\psi\ (y, G\ x))\ h =$
      $\varphi C\ (F\ y, x)\ (\psi C\ (F\ y, x)\ (\Psi.FUN\ (y, x)\ (\varphi D\ (y, G\ x)\ (\psi D\ (y, G\ x)\ h))))$
        using *h ψ-def* **by** *auto*
      also have ... $= \varphi C\ (F\ y, x)\ (\psi C\ (F\ y, x)\ (\Psi.FUN\ (y, x)\ h))$
        using *assms h HomD.φ-ψ Ψ-Fun-mapsto* **by** *simp*
      also have ... $= \Psi.FUN\ (y, x)\ h$
        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 $\Psi.FUN\ (y,\ x)\ h = (\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ HomD.\psi\ (y,\ G\ x))\ h$ **by** *auto*
qed
qed
finally show $\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \psi D\ (y,\ G\ x))$
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 $\varphi\psi$: *meta-adjunction C D F G* $\varphi\ \psi$
proof
fix $y :: 'd$ **and** $x :: 'c$ **and** $h :: 'c$
assume y: $D.ide\ y$ **and** h: $\ll h : F\ y \rightarrow_C x\gg$
have x: $C.ide\ x$ **using** h **by** *auto*
show $\ll\varphi\ y\ h : y \rightarrow_D G\ x\gg$
proof −
have $\Phi.FUN\ (y,\ x) \in HomC.set\ (F\ y,\ x) \rightarrow HomD.set\ (y,\ G\ x)$
using $y\ h\ \Phi$*-Fun-mapsto* **by** *blast*
thus *?thesis*
using $x\ y\ h\ \varphi$*-def* $HomD.\psi$*-mapsto* [*of* $y\ G\ x$] $HomC.\varphi$*-mapsto* [*of* $F\ y\ x$] **by** *auto*
qed
show $\psi\ x\ (\varphi\ y\ h) = h$
proof −
have *0*: $restrict\ (\lambda h.\ h)\ (HomC.set\ (F\ y,\ x))$
$= restrict\ (\varphi C\ (F\ y,\ x)\ o\ (\psi\ x\ o\ \varphi\ y)\ o\ \psi C\ (F\ y,\ x))\ (HomC.set\ (F\ y,\ x))$
proof −
have *1*: $S.ide\ (\Psi\ (y,\ x) \cdot_S \Phi\ (y,\ x))$
using $x\ y\ \Phi\Psi.inv$ [*of* $(y,\ x)$] **by** *auto*
hence *6*: $S.seq\ (\Psi\ (y,\ x))\ (\Phi\ (y,\ x))$ **by** *auto*
have *2*: $\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)) \wedge$
$\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \psi D\ (y,\ G\ x))$
using $x\ y\ \Phi$*-simp* Ψ*-simp* **by** *force*
have *3*: $S\ (\Psi\ (y,\ x))\ (\Phi\ (y,\ x))$
$= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ (\psi\ x\ o\ \varphi\ y)\ o\ \psi C\ (F\ y,\ x))$
proof −
have *4*: $S.arr\ (\Psi\ (y,\ x) \cdot_S \Phi\ (y,\ x))$ **using** *1* **by** *auto*
hence $S\ (\Psi\ (y,\ x))\ (\Phi\ (y,\ x))$
$= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y,\ x))$
$((\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \psi D\ (y,\ G\ x))$
$o\ (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
using *1 2 S.ide-in-hom* **by** *force*
also have ... $= S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ (\psi\ x\ o\ \varphi\ y)\ o\ \psi C\ (F\ y,\ x))$
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 1: «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
      show ((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)
proof −
  have Ψ (y, x) ·S Φ (y, x) = S.dom (S (Ψ (y, x)) (Φ (y, x)))
    using 1 by auto
  also have ... = S.dom (Φ (y, x))
    using 1 S.dom-comp by blast
  finally show ?thesis
    using 2 6 S.mkIde-as-mkArr by (elim S.seqE, auto)
qed
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)))
proof −
  have S.seq (Ψ (y, x)) (Φ (y, x))
    using 1 by fast
  thus ?thesis using 3 by metis
qed
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 by force
      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)
 proof -
  have Φ (y, x) ·S Ψ (y, x) = S.dom (Φ (y, x) ·S Ψ (y, x))
    using 1 by auto
  also have ... = S.dom (Ψ (y, x))
    using 1 S.dom-comp by blast
  finally show ?thesis using 2 6 S.mkIde-as-mkArr by (elim S.seqE, auto)
 qed
 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 $h = \psi D\ (y,\ G\ x)\ (\varphi D\ (y,\ G\ x)\ (\varphi\ y\ (\psi\ x\ (\psi D\ (y,\ G\ x)\ (\varphi D\ (y,\ G\ x)\ h)))))$
using $x\ y\ h$ *HomD*.ψ-φ **by** *simp*
also have ... $= \varphi\ y\ (\psi\ x\ h)$
using $x\ y\ h$ *HomD*.ψ-φ *HomD*.ψ-φ [*of* $\varphi\ y\ (\psi\ x\ h)\ y\ G\ x$] 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: $\ll f : x \rightarrow_C x' \gg$ **and** g: $\ll g : y' \rightarrow_D y \gg$ **and** h: $\ll h : F\ y \rightarrow_C x \gg$
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 $\varphi\ y'\ (f \cdot_C h \cdot_C F\ g) = G\ f \cdot_D \varphi\ y\ h \cdot_D g$
proof −
have *0*: $restrict\ ((\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y,\ G\ x))$
$o\ (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
$(HomC.set\ (F\ y,\ x))$
$= restrict\ ((\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
$o\ (\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f \cdot_C h \cdot_C F\ g))\ o\ \psi C\ (F\ y,\ x))$
$(HomC.set\ (F\ y,\ x))$
proof −
have *1*: $S.arr\ (\Phi\ (y,\ x))\ \wedge$
$\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))$
using $x\ y$ Φ-*simp* [*of* $y\ x$] **by** *auto*
have *2*: $S.arr\ (\Phi\ (y',\ x'))\ \wedge$
$\Phi\ (y',\ x') = S.mkArr\ (HomC.set\ (F\ y',\ x'))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
using $x'\ y'$ Φ-*simp* [*of* $y'\ x'$] **by** *auto*
have *3*: $S.arr\ (S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y,\ G\ x))$
$o\ (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))))$
$\wedge\ S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y',\ G\ x'))$
$((\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y,\ G\ x))$
$o\ (\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
$= S\ (S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y,\ G\ x)))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
proof −
have *1*: $S.seq\ (S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomD.set\ (y',\ G\ x'))$
$(\varphi D\ (y',\ G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y,\ G\ x)))$
$(S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x)))$
proof −

have $S.arr\ (Hom\text{-}DopxG.map\ (g, f)) \wedge$
$Hom\text{-}DopxG.map\ (g, f)$
$= S.mkArr\ (HomD.set\ (y, G\ x))\ (HomD.set\ (y', G\ x'))$
$(\varphi D\ (y', G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y, G\ x))$
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 $S.seq\ (S.mkArr\ (HomD.set\ (y, G\ x))\ (HomD.set\ (y', G\ x'))$
$(\varphi D\ (y', G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y, G\ x)))$
$(S.mkArr\ (HomC.set\ (F\ y, x))\ (HomD.set\ (y, G\ x))$
$(\varphi D\ (y, G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y, x)))$
using *1* **by** (*intro S.seqI′*, *auto*)
moreover have $S.mkArr\ (HomC.set\ (F\ y, x))\ (HomD.set\ (y', G\ x'))$
$((\varphi D\ (y', G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y, G\ x))$
$o\ (\varphi D\ (y, G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y, x)))$
$= S\ (S.mkArr\ (HomD.set\ (y, G\ x))\ (HomD.set\ (y', G\ x'))$
$(\varphi D\ (y', G\ x')\ o\ (\lambda h.\ G\ f \cdot_D h \cdot_D g)\ o\ \psi D\ (y, G\ x)))$
$(S.mkArr\ (HomC.set\ (F\ y, x))\ (HomD.set\ (y, G\ x))$
$(\varphi D\ (y, G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y, x)))$
using *1* **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'*))
(φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*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'*))
((φ*D* (*y'*, *G x'*) *o* φ *y'* *o* ψ*C* (*F y'*, *x'*))
o (φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*C* (*F y*, *x*)))
= *S* (*S.mkArr* (*HomC.set* (*F y'*, *x'*)) (*HomD.set* (*y'*, *G x'*))
(φ*D* (*y'*, *G x'*) *o* φ *y'* *o* ψ*C* (*F y'*, *x'*)))
(*S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomC.set* (*F y'*, *x'*))
(φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*C* (*F y*, *x*)))
using *5* **by** *fastforce*
ultimately show *?thesis* **by** *argo*
qed
moreover have *2*:
S.mkArr (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y'*, *G x'*))
((φ*D* (*y'*, *G x'*) *o* (λ*h*. *G f* ·*D* *h* ·*D* *g*) *o* ψ*D* (*y*, *G x*))
o (φ*D* (*y*, *G x*) *o* φ *y* *o* ψ*C* (*F y*, *x*)))
= *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y'*, *G x'*))
((φ*D* (*y'*, *G x'*) *o* φ *y'* *o* ψ*C* (*F y'*, *x'*))
o (φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*C* (*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*.is-natural-1* Φ*.is-natural-2* **by** *fastforce*
moreover have *Hom-DopxG.map* (*g*, *f*)
= *S.mkArr* (*HomD.set* (*y*, *G x*)) (*HomD.set* (*y'*, *G x'*))
(φ*D* (*y'*, *G x'*) *o* (λ*h*. *G f* ·*D* *h* ·*D* *g*) *o* ψ*D* (*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'*))
(φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*C* (*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'*))
((φ*D* (*y'*, *G x'*) *o* (λ*h*. *G f* ·*D* *h* ·*D* *g*) *o* ψ*D* (*y*, *G x*))
o (φ*D* (*y*, *G x*) *o* φ *y* *o* ψ*C* (*F y*, *x*))))
by *fast*
hence *restrict* ((φ*D* (*y'*, *G x'*) *o* (λ*h*. *D* (*G f*) (*D h g*)) *o* ψ*D* (*y*, *G x*))
o (φ*D* (*y*, *G x*) *o* φ *y* *o* ψ*C* (*F y*, *x*)))
(*HomC.set* (*F y*, *x*))
= *S.Fun* (*S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y'*, *G x'*))
((φ*D* (*y'*, *G x'*) *o* (λ*h*. *G f* ·*D* *h* ·*D* *g*) *o* ψ*D* (*y*, *G x*))
o (φ*D* (*y*, *G x*) *o* φ *y* *o* ψ*C* (*F y*, *x*))))
by *simp*
also have ... = *S.Fun* (*S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y'*, *G x'*))
((φ*D* (*y'*, *G x'*) *o* φ *y'* *o* ψ*C* (*F y'*, *x'*))
o (φ*C* (*F y'*, *x'*) *o* (λ*h*. *f* ·*C* *h* ·*C* *F g*) *o* ψ*C* (*F y*, *x*))))
using *2* **by** *argo*

also have ... = *restrict* $((\varphi D\ (y',\ G\ x')\ o\ \varphi\ y'\ o\ \psi C\ (F\ y',\ x'))$
$o\ (\varphi C\ (F\ y',\ x')\ o\ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x)))$
$(HomC.set\ (F\ y,\ x))$
using *4 S.Fun-mkArr* **by** *meson*
finally show *?thesis* **by** *auto*
qed
hence *5*: $((\varphi D\ (y',\ G\ x') \circ (\lambda h.\ G\ f\ \cdot_D\ h\ \cdot_D\ g) \circ \psi D\ (y,\ G\ x))$
$\circ\ (\varphi D\ (y,\ G\ x) \circ \varphi\ y \circ \psi C\ (F\ y,\ x)))\ (\varphi C\ (F\ y,\ x)\ h) =$
$(\varphi D\ (y',\ G\ x') \circ \varphi\ y' \circ \psi C\ (F\ y',\ x')$
$\circ\ (\varphi C\ (F\ y',\ x') \circ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)) \circ \psi C\ (F\ y,\ x))\ (\varphi C\ (F\ y,\ x)\ h)$
proof −
have $\varphi C\ (F\ y,\ x)\ h \in HomC.set\ (F\ y,\ x)$
using *x y h HomC.φ-mapsto* [*of F y x*] **by** *auto*
thus *?thesis*
using *0 h restr-eqE* [*of* $(\varphi D\ (y',\ G\ x') \circ (\lambda h.\ G\ f\ \cdot_D\ h\ \cdot_D\ g) \circ \psi D\ (y,\ G\ x))$
$\circ\ (\varphi D\ (y,\ G\ x) \circ \varphi\ y \circ \psi C\ (F\ y,\ x))$
$HomC.set\ (F\ y,\ x)$
$(\varphi D\ (y',\ G\ x') \circ \varphi\ y' \circ \psi C\ (F\ y',\ x'))$
$\circ\ (\varphi C\ (F\ y',\ x') \circ (\lambda h.\ f\ \cdot_C\ h\ \cdot_C\ F\ g)\ o\ \psi C\ (F\ y,\ x))$]
by *fast*
qed
show *?thesis*
proof −
have $\varphi\ y'\ (C\ f\ (C\ h\ (F\ g))) =$
$\psi D\ (y',\ G\ x')\ (\varphi D\ (y',\ G\ x')\ (\varphi\ y'\ (\psi C\ (F\ y',\ x')\ (\varphi C\ (F\ y',\ x')$
$(C\ f\ (C\ (\psi C\ (F\ y,\ x)\ (\varphi C\ (F\ y,\ x)\ h))\ (F\ g)))))))$
proof −
have $\psi D\ (y',\ G\ x')\ (\varphi D\ (y',\ G\ x')\ (\varphi\ y'\ (\psi C\ (F\ y',\ x')\ (\varphi C\ (F\ y',\ x')$
$(C\ f\ (C\ (\psi C\ (F\ y,\ x)\ (\varphi C\ (F\ y,\ x)\ h))\ (F\ g)))))))$
$= \psi D\ (y',\ G\ x')\ (\varphi D\ (y',\ G\ x')\ (\varphi\ y'\ (\psi C\ (F\ y',\ x')\ (\varphi C\ (F\ y',\ x')$
$(C\ f\ (C\ h\ (F\ g)))))))$
using *x y h HomC.ψ-φ* **by** *simp*
also have ... $= \psi D\ (y',\ G\ x')\ (\varphi D\ (y',\ G\ x')\ (\varphi\ y'\ (C\ f\ (C\ h\ (F\ g)))))$
using *f g h HomC.ψ-φ* [*of* $C\ f\ (C\ h\ (F\ g))$] **by** *fastforce*
also have ... $= \varphi\ y'\ (C\ f\ (C\ h\ (F\ g)))$
proof −
have $\langle\langle \varphi\ y'\ (f\ \cdot_C\ h\ \cdot_C\ F\ g) : y' \rightarrow_D G\ x' \rangle\rangle$
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
... $= \psi D\ (y',\ G\ x')$
$(\varphi D\ (y',\ G\ x')$
$(G\ f\ \cdot_D\ \psi D\ (y,\ G\ x)\ (\varphi D\ (y,\ G\ x)\ (\varphi\ y\ (\psi C\ (F\ y,\ x)\ (\varphi C\ (F\ y,\ x)\ h))))$
$\cdot_D\ g))$
using *5* **by** *force*
also have ... $= D\ (G\ f)\ (D\ (\varphi\ y\ h)\ g)$

proof −
  have φyh: $\ll\varphi\ y\ h : y \rightarrow_D\ G\ x\gg$
    using $x\ y\ h$ *φ-mapsto* **by** *auto*
  have $\psi D\ (y', G\ x')$
        $(\varphi D\ (y', G\ x')$
          $(G\ f \cdot_D \psi D\ (y, G\ x)\ (\varphi D\ (y, G\ x)\ (\varphi\ y\ (\psi C\ (F\ y, x)\ (\varphi C\ (F\ y, x)\ h))))$
            $\cdot_D\ g)) =$
      $\psi D\ (y', G\ x')\ (\varphi D\ (y', G\ x')\ (G\ f \cdot_D \psi D\ (y, G\ x)\ (\varphi D\ (y, G\ x)\ (\varphi\ y\ h)) \cdot_D\ g))$
    using $x\ y\ f\ g\ h$ **by** *auto*
  also have ... $= \psi D\ (y', G\ x')\ (\varphi D\ (y', G\ x')\ (G\ f \cdot_D \varphi\ y\ h \cdot_D\ g))$
    using $\varphi yh\ x'\ y'\ f\ g$ **by** *simp*
  also have ... $= G\ f \cdot_D \varphi\ y\ h \cdot_D\ g$
  proof −
    have $\ll G\ f \cdot_D \varphi\ y\ h \cdot_D\ g : y' \rightarrow_D\ G\ x'\gg$
      using $x\ x'\ y'\ f\ g\ h$ *φ-mapsto* φyh **by** *blast*
    thus *?thesis*
      using $x\ y\ f\ g\ h\ \varphi yh$ *HomD.ψ-φ* **by** *auto*
  qed
  finally show *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed
qed
qed

theorem *induces-meta-adjunction*:
shows *meta-adjunction* $C\ D\ F\ G\ \varphi\ \psi$ **..**

end

17.10 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 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 $\eta\varepsilon$: *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$
  using *induces-unit-counit-adjunction η-def ε-def* **by** *auto*

interpretation $\Phi\Psi$: *hom-adjunction* $C\ D$ *SetCat.comp* $\varphi C\ \varphi D\ F\ G\ \Phi\ \Psi$
  using *induces-hom-adjunction* **by** *auto*

theorem *induces-adjunction*:

shows *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
apply (*unfold-locales*)
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 *auto*

end

sublocale *meta-adjunction* $\subseteq$ *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
using *induces-adjunction* **by** *auto*

context *unit-counit-adjunction*
begin

interpretation $\varphi\psi$: *meta-adjunction C D F G* φ ψ **using** *induces-meta-adjunction* **by** *auto*

interpretation *F*: *left-adjoint-functor D C F* **using** $\varphi\psi$*.has-left-adjoint-functor* **by** *auto*
interpretation *G*: *right-adjoint-functor C D G* **using** $\varphi\psi$*.has-right-adjoint-functor* **by** *auto*

abbreviation *HomC* **where** *HomC* $\equiv$ $\varphi\psi$*.HomC*
abbreviation φC **where** $\varphi C \equiv \varphi\psi.\varphi C$
abbreviation *HomD* **where** *HomD* $\equiv$ $\varphi\psi$*.HomD*
abbreviation φD **where** $\varphi D \equiv \varphi\psi.\varphi D$
abbreviation Φ **where** $\Phi \equiv \varphi\psi.\Phi$
abbreviation Ψ **where** $\Psi \equiv \varphi\psi.\Psi$

interpretation $\Phi\Psi$: *hom-adjunction C D SetCat.comp* φC φD *F G* Φ Ψ
using $\varphi\psi$*.induces-hom-adjunction* **by** *auto*

theorem *induces-adjunction*:
shows *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
using ε*-in-terms-of-*ψ η*-in-terms-of-*φ $\varphi\psi.\varphi$*-in-terms-of-*Φ' ψ*-def* $\varphi\psi.\psi$*-in-terms-of-*Ψ' $\varphi\psi.\Phi$*-simp* $\varphi\psi.\Psi$*-simp* φ*-def*
apply (*unfold-locales*)
by *auto*

end

The following fails, claiming “roundup bound exceeded”:
sublocale *unit-counit-adjunction* $\subseteq$ *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ **using** *induces-adjunction* **by** *auto*

context *hom-adjunction*
begin

interpretation $\varphi\psi$: *meta-adjunction C D F G* φ ψ
using *induces-meta-adjunction* **by** *auto*

interpretation *F*: *left-adjoint-functor D C F* **using** $\varphi\psi$*.has-left-adjoint-functor* **by** *auto*
interpretation *G*: *right-adjoint-functor C D G* **using** $\varphi\psi$*.has-right-adjoint-functor* **by** *auto*

abbreviation η **where** $\eta \equiv \varphi\psi.\eta$
abbreviation ε **where** $\varepsilon \equiv \varphi\psi.\varepsilon$

interpretation $\eta\varepsilon$: *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$
using $\varphi\psi$.*induces-unit-counit-adjunction* $\varphi\psi.\eta$-*def* $\varphi\psi.\varepsilon$-*def* **by** *auto*

theorem *induces-adjunction*:
shows *adjunction* $C\ D\ S\ \varphi C\ \varphi D\ F\ G\ \varphi\ \psi\ \eta\ \varepsilon\ \Phi\ \Psi$
proof
fix x
assume $C.ide\ x$
thus $\varepsilon\ x = \psi\ x\ (G\ x)$ **using** $\varphi\psi.\varepsilon$-*map-simp* $\varphi\psi.\varepsilon$-*def* **by** *simp*
next
fix y
assume $D.ide\ y$
thus $\eta\ y = \varphi\ y\ (F\ y)$ **using** $\varphi\psi.\eta$-*map-simp* $\varphi\psi.\eta$-*def* **by** *simp*
fix $x\ y\ f$
assume y: $D.ide\ y$ **and** f: $\ll f : F\ y \rightarrow_C x\gg$
show $\varphi\ y\ f = G\ f \cdot_D \eta\ y$ **using** $y\ f$ $\varphi\psi.\varphi$-*in-terms-of-*η $\varphi\psi.\eta$-*def* **by** *simp*
show $\varphi\ y\ f = (\psi D\ (y,\ G\ x) \circ \Phi.FUN\ (y,\ x) \circ \varphi C\ (F\ y,\ x))\ f$ **using** $y\ f$ φ-*def* **by** *auto*
next
fix $x\ y\ g$
assume x: $C.ide\ x$ **and** g: $\ll g : y \rightarrow_D G\ x\gg$
show $\psi\ x\ g = \varepsilon\ x \cdot_C F\ g$ **using** $x\ g$ $\varphi\psi.\psi$-*in-terms-of-*ε $\varphi\psi.\varepsilon$-*def* **by** *simp*
show $\psi\ x\ g = (\psi C\ (F\ y,\ x) \circ \Psi.FUN\ (y,\ x) \circ \varphi D\ (y,\ G\ x))\ g$ **using** $x\ g$ ψ-*def* **by** *fast*
next
fix $x\ y$
assume x: $C.ide\ x$ **and** y: $D.ide\ y$
show $\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \psi C\ (F\ y,\ x))$
using $x\ y$ Φ-*simp* **by** *simp*
show $\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \psi D\ (y,\ G\ x))$
using $x\ y$ Ψ-*simp* **by** *simp*
qed

end

The following fails for unknown reasons:
sublocale *hom-adjunction* $\subseteq$ *adjunction* $C\ D\ S\ \varphi C\ \varphi D\ F\ G\ \varphi\ \psi\ \eta\ \varepsilon\ \Phi\ \Psi$ **using** *induces-adjunction* **by** *auto*

context *left-adjoint-functor*
begin

interpretation $\varphi\psi$: *meta-adjunction* $C\ D\ F\ G\ \varphi\ \psi$
using *induces-meta-adjunction* **by** *auto*

abbreviation *HomC* **where** $HomC \equiv \varphi\psi.HomC$

abbreviation φC **where** $\varphi C \equiv \varphi\psi.\varphi C$
abbreviation $HomD$ **where** $HomD \equiv \varphi\psi.HomD$
abbreviation φD **where** $\varphi D \equiv \varphi\psi.\varphi D$
abbreviation η **where** $\eta \equiv \varphi\psi.\eta$
abbreviation ε **where** $\varepsilon \equiv \varphi\psi.\varepsilon$
abbreviation Φ **where** $\Phi \equiv \varphi\psi.\Phi$
abbreviation Ψ **where** $\Psi \equiv \varphi\psi.\Psi$

theorem *induces-adjunction*:
shows *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
using $\varphi\psi$*.induces-adjunction* **by** *auto*

end

sublocale *left-adjoint-functor* $\subseteq$ *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
using *induces-adjunction* **by** *auto*

context *right-adjoint-functor*
begin

interpretation $\varphi\psi$: *meta-adjunction C D F G* φ ψ
using *induces-meta-adjunction* **by** *auto*

abbreviation $HomC$ **where** $HomC \equiv \varphi\psi.HomC$
abbreviation φC **where** $\varphi C \equiv \varphi\psi.\varphi C$
abbreviation $HomD$ **where** $HomD \equiv \varphi\psi.HomD$
abbreviation φD **where** $\varphi D \equiv \varphi\psi.\varphi D$
abbreviation η **where** $\eta \equiv \varphi\psi.\eta$
abbreviation ε **where** $\varepsilon \equiv \varphi\psi.\varepsilon$
abbreviation Φ **where** $\Phi \equiv \varphi\psi.\Phi$
abbreviation Ψ **where** $\Psi \equiv \varphi\psi.\Psi$

theorem *induces-adjunction*:
shows *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ
using $\varphi\psi$*.induces-adjunction* **by** *auto*

end

The following fails, claiming "roundup bound exceeded":
sublocale *right-adjoint-functor* $\subseteq$ *adjunction C D SetCat.comp* φC φD *F G* φ ψ η ε Φ Ψ **using** *induces-adjunction* **by** *auto*

definition *adjoint-functors*
where *adjoint-functors C D F G* = ($\exists \varphi\ \psi$. *meta-adjunction C D F G* φ ψ)

17.11 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* $A\ B\ F\ G\ \varphi\ \psi$ +
$F'G'$: *meta-adjunction* $B\ C\ F'\ G'\ \varphi'\ \psi'$
for A :: $'a$ *comp* (**infixr** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot_C$ *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

lemma *is-meta-adjunction*:
shows *meta-adjunction* $A\ C\ (F\ o\ F')\ (G'\ o\ G)\ (\lambda z.\ \varphi'\ z\ o\ \varphi\ (F'\ z))\ (\lambda x.\ \psi\ x\ o\ \psi'\ (G\ x))$
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: $\ll f : FoF'.map\ y \rightarrow_A x\gg$
show $\ll(\varphi'\ y \circ \varphi\ (F'\ y))\ f : y \rightarrow_C G'oG.map\ x\gg$
using $y\ f\ FG.\varphi\text{-}in\text{-}hom\ F'G'.\varphi\text{-}in\text{-}hom$ **by** *simp*
show $(\psi\ x \circ \psi'\ (G\ x))\ ((\varphi'\ y \circ \varphi\ (F'\ y))\ f) = f$
using $y\ f\ FG.\varphi\text{-}in\text{-}hom\ F'G'.\varphi\text{-}in\text{-}hom\ FG.\psi\text{-}\varphi\ F'G'.\psi\text{-}\varphi$ **by** *simp*
next
fix $x\ g\ y$
assume x: $A.ide\ x$ **and** g: $\ll g : y \rightarrow_C G'oG.map\ x\gg$
show $\ll(\psi\ x \circ \psi'\ (G\ x))\ g : FoF'.map\ y \rightarrow_A x\gg$
using $x\ g\ FG.\psi\text{-}in\text{-}hom\ F'G'.\psi\text{-}in\text{-}hom$ **by** *auto*
show $(\varphi'\ y \circ \varphi\ (F'\ y))\ ((\psi\ x \circ \psi'\ (G\ x))\ g) = g$
using $x\ g\ FG.\psi\text{-}in\text{-}hom\ F'G'.\psi\text{-}in\text{-}hom\ FG.\varphi\text{-}\psi\ F'G'.\varphi\text{-}\psi$ **by** *simp*
next
fix $f\ x\ x'\ g\ y'\ y\ h$
assume f: $\ll f : x \rightarrow_A x'\gg$ **and** g: $\ll g : y' \rightarrow_C y\gg$ **and** h: $\ll h : FoF'.map\ y \rightarrow_A x\gg$
show $(\varphi'\ y' \circ \varphi\ (F'\ y'))\ (f \cdot_A h \cdot_A FoF'.map\ g) =$
$G'oG.map\ f \cdot_C (\varphi'\ y \circ \varphi\ (F'\ y))\ h \cdot_C g$
using $f\ g\ h\ FG.\varphi\text{-}naturality$ [*of* $f\ x\ x'\ F'\ g\ F'\ y'\ F'\ y\ h$]
$F'G'.\varphi\text{-}naturality$ [*of* $G\ f\ G\ x\ G\ x'\ g\ y'\ y\ \varphi\ (F'\ y)\ h$]

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-is-natural-transformation F′.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′.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.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.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*

lemma *η-char*:
shows *η* = *G′ηF′oη′.map*
proof (*intro NaturalTransformation.eqI*)
show *natural-transformation C C C.map* (*G′ o G o F o F′*) *G′ηF′oη′.map* ..
show *natural-transformation C C C.map* (*G′ o G o F o F′*) *η*

proof −
have *natural-transformation* $C\ C\ C.map\ ((G' \circ G) \circ (F \circ F'))\ \eta$ **..**
moreover have $(G'\ o\ G)\ o\ (F\ o\ F') = G'\ o\ G\ o\ F\ o\ F'$ **by** *auto*
ultimately show *?thesis* **by** *metis*
qed
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 NaturalTransformation.eqI*)
show *natural-transformation* $A\ A\ (F\ o\ F'\ o\ G'\ o\ G)\ A.map\ \varepsilon$
proof −
have *natural-transformation* $A\ A\ ((F \circ F') \circ (G' \circ G))\ A.map\ \varepsilon$ **..**
moreover have $(F\ o\ F')\ o\ (G'\ o\ G) = F\ o\ F'\ o\ G'\ o\ G$ **by** *auto*
ultimately show *?thesis* **by** *metis*
qed
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 ... $= FG.\varepsilon\ a \cdot_A F\ (F'G'.\varepsilon\ (G\ a) \cdot_B F'\ (G'\ (G\ a)))$
unfolding *ε-def*
using a *F'G'.ψ-in-terms-of-ε* [*of* $G\ a\ G'\ (G\ a)\ G'\ (G\ a)$]
F'G'.ε.preserves-hom [*of* $G\ a\ G\ a\ G\ a$]
FG.ψ-in-terms-of-ε [*of* $a\ F'G'.\varepsilon\ (G\ a) \cdot_B F'\ (G'\ (G\ a))\ (F'G'.FG.map\ (G\ a))$]
F'G'.ε-def FG.ε-def
by *fastforce*
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

17.12 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** $\varphi\psi$: *meta-adjunction C D F G* φ ψ
  using *assms adjoint-functors-def* **by** *blast*
obtain φ' ψ' **where** $\varphi'\psi'$: *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** $\varphi\psi$ **by** *auto*
interpret
    Adj: *adjunction C D SetCat.comp Adj.*φ*C Adj.*φ*D 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** $\varphi'\psi'$ **by** *auto*
interpret *Adj'*: *adjunction C D SetCat.comp Adj'.*φ*C Adj'.*φ*D*
      F G' φ' ψ' *Adj'.*η *Adj'.*ε *Adj'.*Φ *Adj'.*Ψ
  using *Adj'.induces-adjunction* **by** *auto*
write *C* (**infixr** $\cdot_C$ *55*)
write *D* (**infixr** $\cdot_D$ *55*)
write *Adj.C.in-hom* ($\ll$- : - $\rightarrow_C$ -$\gg$)
write *Adj.D.in-hom* ($\ll$- : - $\rightarrow_D$ -$\gg$)
let *?*τ*o* = $\lambda a.\ G'\ (Adj.\varepsilon\ a) \cdot_D Adj'.\eta\ (G\ a)$
interpret τ: *transformation-by-components C D G G' ?*τ*o*
proof
  show $\bigwedge a.\ Adj.C.ide\ a \Longrightarrow \ll G'\ (Adj.\varepsilon\ a) \cdot_D Adj'.\eta\ (G\ a) : G\ a \rightarrow_D G'\ a\gg$
    by *fastforce*
  show $\bigwedge f.\ Adj.C.arr\ f \Longrightarrow$
      $(G'\ (Adj.\varepsilon\ (Adj.C.cod\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.cod\ f))) \cdot_D G\ f =$
      $G'\ f \cdot_D G'\ (Adj.\varepsilon\ (Adj.C.dom\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.dom\ f))$
  proof –
    fix *f*
    assume *f*: *Adj.C.arr f*
    let *?x* = *Adj.C.dom f*
    let *?x'* = *Adj.C.cod f*
    have $(G'\ (Adj.\varepsilon\ (Adj.C.cod\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.cod\ f))) \cdot_D G\ f =$
        $G'\ (Adj.\varepsilon\ (Adj.C.cod\ f) \cdot_C F\ (G\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.dom\ f))$
      using *f Adj'.*η*.naturality* [*of G f*] *Adj.D.comp-assoc* **by** *simp*
    also have ... = $G'\ (f \cdot_C Adj.\varepsilon\ (Adj.C.dom\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.dom\ f))$

using *f Adj.ε.naturality* **by** *simp*
also have ... = $G'\ f \cdot_D G'\ (Adj.\varepsilon\ (Adj.C.dom\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.dom\ f))$
using *f Adj.D.comp-assoc* **by** *simp*
finally show $(G'\ (Adj.\varepsilon\ (Adj.C.cod\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.cod\ f))) \cdot_D G\ f =$
$G'\ f \cdot_D G'\ (Adj.\varepsilon\ (Adj.C.dom\ f)) \cdot_D Adj'.\eta\ (G\ (Adj.C.dom\ f))$
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.

$$\begin{tikzcd}
G'a \arrow[rr, "\tau' a"] \arrow[dr, "\eta(G'a)"'] \arrow[dd, "\eta'(G'a)"'] & & Ga \arrow[rr, "\tau a"] \arrow[dr, "\eta'(Ga)"'] & & G'a \\
& GFG'a \arrow[ur, "G(\epsilon' a)"'] \arrow[dr, "\eta'(GFG'a)"'] & \text{(2)} & G'FGa \arrow[ur, "G'(\epsilon a)"'] & \\
G'FG'a \arrow[rr, "G'F\eta(G'a)"'] \arrow[rrrr, bend right, "G'FG'a"'] \arrow[ur, phantom, "\text{(1)}"] & & G'FGFG'a \arrow[ur, "G'FG(\epsilon' a)"'] \arrow[rr, "G'(\epsilon(FG'a))"'] \arrow[d, phantom, "\text{(4)}"] & & G'FG'a \arrow[uu, "G'(\epsilon' a)"'] \arrow[ul, phantom, "\text{(3)}"] \\
& & {} & &
\end{tikzcd}$$

show *Adj.D.ide* (*τ.map a* $\cdot_D$ φ (G' *a*) (*Adj′.ε a*))
proof −
have *τ.map a* $\cdot_D$ φ (G' *a*) (*Adj′.ε a*) = G' *a*
proof −
have *τ.map a* $\cdot_D$ φ (G' *a*) (*Adj′.ε a*) =
$G'\ (Adj.\varepsilon\ a) \cdot_D (Adj'.\eta\ (G\ a) \cdot_D G\ (Adj'.\varepsilon\ a)) \cdot_D Adj.\eta\ (G'\ a)$
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 ... = $G'\ (Adj.\varepsilon\ a) \cdot_D (G'\ (F\ (G\ (Adj'.\varepsilon\ a))) \cdot_D Adj'.\eta\ (G\ (F\ (G'\ a)))) \cdot_D$
$Adj.\eta\ (G'\ a)$

using a $Adj'.\eta.naturality$ $[of\ G\ (Adj'.\varepsilon\ a)]$ **by** $auto$
also have ... $= (G'\,(Adj.\varepsilon\ a) \cdot_D G'\,(F\ (G\ (Adj'.\varepsilon\ a)))) \cdot_D G'\,(F\ (Adj.\eta\ (G'\ a))) \cdot_D Adj'.\eta\ (G'\ a)$
using a $Adj'.\eta.naturality$ $[of\ Adj.\eta\ (G'\ a)]$ $Adj.D.comp\text{-}assoc$ **by** $auto$
also have
... $= G'\,(Adj'.\varepsilon\ a) \cdot_D (G'\,(Adj.\varepsilon\ (F\ (G'\ a))) \cdot_D G'\,(F\ (Adj.\eta\ (G'\ a)))) \cdot_D Adj'.\eta\ (G'\ a)$
proof −
have
$G'\,(Adj.\varepsilon\ a) \cdot_D G'\,(F\ (G\ (Adj'.\varepsilon\ a))) = G'\,(Adj'.\varepsilon\ a) \cdot_D G'\,(Adj.\varepsilon\ (F\ (G'\ a)))$
proof −
have $G'\,(Adj.\varepsilon\ a \cdot_C F\ (G\ (Adj'.\varepsilon\ a))) = G'\,(Adj'.\varepsilon\ a \cdot_C Adj.\varepsilon\ (F\ (G'\ a)))$
using a $Adj.\varepsilon.naturality$ $[of\ Adj'.\varepsilon\ a]$ **by** $auto$
thus *?thesis* **using** a **by** *force*
qed
thus *?thesis* **using** $Adj.D.comp\text{-}assoc$ **by** $auto$
qed
also have ... $= G'\,(Adj'.\varepsilon\ a) \cdot_D Adj'.\eta\ (G'\ a)$
proof −
have $G'\,(Adj.\varepsilon\ (F\ (G'\ a))) \cdot_D G'\,(F\ (Adj.\eta\ (G'\ a))) = G'\,(F\ (G'\ a))$
proof −
have
$G'\,(Adj.\varepsilon\ (F\ (G'\ a))) \cdot_D G'\,(F\ (Adj.\eta\ (G'\ a))) = G'\,(Adj.\varepsilon FoF\eta.map\ (G'\ a))$
using a $Adj.\varepsilon FoF\eta.map\text{-}simp\text{-}1$ **by** $auto$
moreover have $Adj.\varepsilon FoF\eta.map\ (G'\ a) = F\ (G'\ a)$
using a **by** $(simp\ add{:}\ Adj.\eta\varepsilon.triangle\text{-}F)$
ultimately show *?thesis* **by** $auto$
qed
thus *?thesis*
using a $Adj.D.comp\text{-}cod\text{-}arr$ $[of\ Adj'.\eta\ (G'\ a)]$ **by** $auto$
qed
also have ... $= G'\ a$
using a $Adj'.\eta\varepsilon.triangle\text{-}G$ $Adj'.G\varepsilon o\eta G.map\text{-}simp\text{-}1$ $[of\ a]$ **by** $auto$
finally show *?thesis* **by** $auto$
qed
thus *?thesis* **using** a **by** *simp*
qed
show $Adj.D.ide\ (\varphi\ (G'\ a)\ (Adj'.\varepsilon\ a) \cdot_D \tau.map\ a)$
proof −
have $\varphi\ (G'\ a)\ (Adj'.\varepsilon\ a) \cdot_D \tau.map\ a = G\ a$
proof −
have $\varphi\ (G'\ a)\ (Adj'.\varepsilon\ a) \cdot_D \tau.map\ a =$
$G\ (Adj'.\varepsilon\ a) \cdot_D (Adj.\eta\ (G'\ a) \cdot_D G'\,(Adj.\varepsilon\ a)) \cdot_D Adj'.\eta\ (G\ a)$
using a $\tau.map\text{-}simp\text{-}ide$ $Adj.\varphi\text{-}in\text{-}terms\text{-}of\text{-}\eta$ $Adj'.\varepsilon.preserves\text{-}hom$ $[of\ a\ a\ a]$ $Adj.C.ide\text{-}in\text{-}hom$ $Adj.D.comp\text{-}assoc$ $Adj.\eta\text{-}def$
by $auto$
also have
... $= G\ (Adj'.\varepsilon\ a) \cdot_D (G\ (F\ (G'\,(Adj.\varepsilon\ a))) \cdot_D Adj.\eta\ (G'\,(F\ (G\ a)))) \cdot_D Adj'.\eta\ (G\ a)$

```
            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
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 18

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.

18.1 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* +

F: *functor Cop.comp S F* +
Hom: *hom-functor C S* φ +
Ya: *yoneda-functor-fixed-object C S* φ a +
natural-isomorphism Cop.comp S $\langle Ya.Y\ a\rangle$ F Φ
for C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and F :: $'c \Rightarrow 's$
and a :: $'c$
and Φ :: $'c \Rightarrow 's$
begin

abbreviation Y **where** $Y \equiv Ya.Y$
abbreviation ψ **where** $\psi \equiv Hom.\psi$

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* +
F: *set-valued-functor Cop.comp S F* +
yoneda-functor C S φ +
Ya: *yoneda-functor-fixed-object C S* φ a +
Ya': *yoneda-functor-fixed-object C S* φ a' +
Φ: *representation-of-functor C S* φ F a Φ +
Φ': *representation-of-functor C S* φ F a' Φ'
for C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and F :: $'c \Rightarrow 's$
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and a :: $'c$
and Φ :: $'c \Rightarrow 's$
and a' :: $'c$
and Φ' :: $'c \Rightarrow 's$
begin

interpretation Ψ: *inverse-transformation Cop.comp S* $\langle Y\ a\rangle$ F Φ **..**
interpretation Ψ': *inverse-transformation Cop.comp S* $\langle Y\ a'\rangle$ F Φ' **..**
interpretation $\Phi\Psi'$: *vertical-composite Cop.comp S* $\langle Y\ a\rangle$ F $\langle Y\ a'\rangle$ Φ $\Psi'.map$ **..**
interpretation $\Phi'\Psi$: *vertical-composite Cop.comp S* $\langle Y\ a'\rangle$ F $\langle Y\ a\rangle$ Φ' $\Psi.map$ **..**

lemma *are-uniquely-isomorphic*:
shows $\exists!\varphi.\ \langle\!\langle \varphi : a \rightarrow a' \rangle\!\rangle \wedge C.iso\ \varphi \wedge map\ \varphi = Cop\text{-}S.MkArr\ (Y\ a)\ (Y\ a')\ \Phi\Psi'.map$
proof −
have *natural-isomorphism Cop.comp S* $(Y\ a)$ F Φ **..**
moreover have *natural-isomorphism Cop.comp S F* $(Y\ a')$ $\Psi'.map$ **..**
ultimately have *1*: *natural-isomorphism Cop.comp S* $(Y\ a)$ $(Y\ a')$ $\Phi\Psi'.map$

using *NaturalTransformation.natural-isomorphisms-compose* **by** *blast*
interpret $\Phi\Psi'$: *natural-isomorphism Cop.comp S* ‹*Y a*› ‹*Y a'*› $\Phi\Psi'$*.map*
using *1* **by** *auto*

have *natural-isomorphism Cop.comp S* (*Y a'*) *F* Φ' **..**
moreover have *natural-isomorphism Cop.comp S F* (*Y a*) Ψ*.map* **..**
ultimately have *2*: *natural-isomorphism Cop.comp S* (*Y a'*) (*Y a*) $\Phi'\Psi$*.map*
using *NaturalTransformation.natural-isomorphisms-compose* **by** *blast*
interpret $\Phi'\Psi$: *natural-isomorphism Cop.comp S* ‹*Y a'*› ‹*Y a*› $\Phi'\Psi$*.map*
using *2* **by** *auto*

interpret $\Phi\Psi'$-$\Phi'\Psi$: *inverse-transformations Cop.comp S* ‹*Y a*› ‹*Y a'*› $\Phi\Psi'$*.map* $\Phi'\Psi$*.map*
proof
fix *x*
assume *X*: *Cop.ide x*
show *S.inverse-arrows* ($\Phi\Psi'$*.map x*) ($\Phi'\Psi$*.map x*)
proof
have *1*: *S.arr* ($\Phi\Psi'$*.map x*) $\wedge$ $\Phi\Psi'$*.map x* = Ψ'*.map x* $\cdot_S$ Φ *x*
using *X* $\Phi\Psi'$*.preserves-reflects-arr* [*of x*]
by (*simp add*: $\Phi\Psi'$*.map-simp-2*)
have *2*: *S.arr* ($\Phi'\Psi$*.map x*) $\wedge$ $\Phi'\Psi$*.map x* = Ψ*.map x* $\cdot_S$ Φ' *x*
using *X* $\Phi'\Psi$*.preserves-reflects-arr* [*of x*]
by (*simp add*: $\Phi'\Psi$*.map-simp-1*)
show *S.ide* ($\Phi\Psi'$*.map x* $\cdot_S$ $\Phi'\Psi$*.map x*)
using *1 2 X* Ψ*.is-natural-2* Ψ'*.inverts-components* Ψ*.inverts-components*
by (*metis S.inverse-arrows-def S.inverse-arrows-compose*)
show *S.ide* ($\Phi'\Psi$*.map x* $\cdot_S$ $\Phi\Psi'$*.map x*)
using *1 2 X* Ψ'*.inverts-components* Ψ*.inverts-components*
by (*metis S.inverse-arrows-def S.inverse-arrows-compose*)
qed
qed

have *Cop-S.inverse-arrows* (*Cop-S.MkArr* (*Y a*) (*Y a'*) $\Phi\Psi'$*.map*)
(*Cop-S.MkArr* (*Y a'*) (*Y a*) $\Phi'\Psi$*.map*)
proof −
have *Ya*: *functor Cop.comp S* (*Y a*) **..**
have *Ya'*: *functor Cop.comp S* (*Y a'*) **..**
have $\Phi\Psi'$: *natural-transformation Cop.comp S* (*Y a*) (*Y a'*) $\Phi\Psi'$*.map* **..**
have $\Phi'\Psi$: *natural-transformation Cop.comp S* (*Y a'*) (*Y a*) $\Phi'\Psi$*.map* **..**
show *?thesis*
proof (*intro Cop-S.inverse-arrowsI*)
have *0*: *inverse-transformations Cop.comp S* (*Y a*) (*Y a'*) $\Phi\Psi'$*.map* $\Phi'\Psi$*.map* **..**
have *1*: *Cop-S.antipar* (*Cop-S.MkArr* (*Y a*) (*Y a'*) $\Phi\Psi'$*.map*)
(*Cop-S.MkArr* (*Y a'*) (*Y a*) $\Phi'\Psi$*.map*)
using *Ya Ya'* $\Phi\Psi'$ $\Phi'\Psi$ *Cop-S.dom-char Cop-S.cod-char Cop-S.seqI*
Cop-S.arr-MkArr Cop-S.cod-MkArr Cop-S.dom-MkArr
by *presburger*
show *Cop-S.ide* (*Cop-S.comp* (*Cop-S.MkArr* (*Y a*) (*Y a'*) $\Phi\Psi'$*.map*)
(*Cop-S.MkArr* (*Y a'*) (*Y a*) $\Phi'\Psi$*.map*))

```
      using 0 1 NaturalTransformation.inverse-transformations-inverse(2) Cop-S.comp-MkArr
        by (metis Cop-S.cod-MkArr Cop-S.ide-char′ Cop-S.seqE)
      show Cop-S.ide (Cop-S.comp (Cop-S.MkArr (Y a′) (Y a) Φ′Ψ.map)
                                  (Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map))
      using 0 1 NaturalTransformation.inverse-transformations-inverse(1) Cop-S.comp-MkArr
        by (metis Cop-S.cod-MkArr Cop-S.ide-char′ Cop-S.seqE)
    qed
  qed
  hence 3: Cop-S.iso (Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map) using Cop-S.isoI by blast
  hence Cop-S.arr (Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map) using Cop-S.iso-is-arr by blast
  hence Cop-S.in-hom (Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map) (map a) (map a′)
    using Ya.ide-a Ya′.ide-a Cop-S.dom-char Cop-S.cod-char by auto
  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
    by auto
  from this obtain φ
    where φ: «φ : a → a′» ∧ map φ = Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map
    by blast
  from φ have C.iso φ
    using 3 reflects-iso [of φ a a′] by simp
  hence EX: ∃φ. «φ : a → a′» ∧ C.iso φ ∧ map φ = Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map
    using φ by blast
  have
    UN: ⋀φ′. «φ′ : a → a′» ∧ map φ′ = Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map ⟹ φ′ = φ
  proof −
    fix φ′
    assume φ′: «φ′ : a → a′» ∧ map φ′ = Cop-S.MkArr (Y a) (Y a′) ΦΨ′.map
    have C.par φ φ′ ∧ map φ = map φ′ using φ φ′ by auto
    thus φ′ = φ using is-faithful by fast
  qed
  from EX UN show ?thesis by auto
qed

end
```

18.2 Diagrams and Cones

A *diagram* in a category C is a functor D: $J \to C$. 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** $\cdot_J$ *55*)
and C :: *'c comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
begin

notation *J.in-hom* («- : - $\rightarrow_J$ -»)

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: $J \rightarrow C$ 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** $\cdot_J$ *55*)
and C :: *'c comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and a :: $'c$
and χ :: $'j \Rightarrow 'c$
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 : $a \rightarrow D$ (*J.cod j*)»
using *assms* **by** *auto*

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*
apply *unfold-locales* **using** χ.*A.value-is-ide* **by** *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.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)
proof −
interpret χ: *cone J C D a* χ **using** *assms*(*1*) **by** *auto*
interpret τ: *natural-transformation J C D D′* τ **using** *assms*(*2*) **by** *auto*
interpret $\tau o\chi$: *vertical-composite J C* χ.*A.map D D′* χ τ ..
interpret $\tau o\chi$: *cone J C D′ a* $\tau o\chi$.*map* ..
show *?thesis* ..
qed

context *functor*
begin

lemma *preserves-diagrams*:
fixes *J* :: *′j comp*
assumes *diagram J A D*
shows *diagram J B* (*F o D*)
proof −
interpret *D*: *diagram J A D* **using** *assms* **by** *auto*
interpret *FoD*: *composite-functor J A B D F* ..
show *diagram J B* (*F o D*) ..
qed

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*›
apply *unfold-locales* **using** χ.*ide-apex* **by** *auto*
have *1*: *F o* χ.*A.map* = *Fa.map*

proof
  fix *f*
  show $(F \circ \chi.A.map)\ f = Fa.map\ f$
    using *is-extensional Fa.is-extensional* χ*.A.is-extensional*
    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*
      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* $\chi \equiv$ *Limit.cone J C D a* χ

abbreviation *cones* :: $'c \Rightarrow ('j \Rightarrow 'c)$ *set*
where *cones a* $\equiv \{\ \chi.$ *cone a* $\chi\ \}$

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* :: $'c \Rightarrow ('j \Rightarrow 'c) \Rightarrow ('j \Rightarrow 'c)$
where *cones-map f* $\equiv (\lambda\chi \in$ *cones* (*C.cod f*). $\lambda j.$ *if J.arr j then* $\chi\ j \cdot f$ *else C.null*)

lemma *cones-map-mapsto*:
assumes *C.arr f*
shows *cones-map f* $\in$
    extensional (*cones* (*C.cod f*)) $\cap$ (*cones* (*C.cod f*) $\rightarrow$ *cones* (*C.dom f*))
proof
  show *cones-map f* $\in$ *extensional* (*cones* (*C.cod f*)) **by** *blast*
  show *cones-map f* $\in$ *cones* (*C.cod f*) $\rightarrow$ *cones* (*C.dom f*)
  proof
    fix χ
    assume $\chi \in$ *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*›
      apply *unfold-locales* **using** *assms* **by** *auto*
    have *cone* (*C.dom f*) ($\lambda j.$ *if J.arr j then* $\chi\ j \cdot f$ *else C.null*)
      using *assms B.value-is-ide* χ*.is-natural-1* χ*.is-natural-2*
      apply (*unfold-locales*, *auto*)
      using χ*.is-natural-1*
       apply (*metis C.comp-assoc*)
      using χ*.is-natural-2 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 $j \cdot f$ *else C.null*) $\in$ *cones* (*C.dom f*) **by** *auto*
 qed
qed

lemma *cones-map-ide*:
assumes $\chi \in$ *cones a*
shows *cones-map a* $\chi = \chi$
proof −
 interpret χ: *cone J C D a* χ **using** *assms* **by** *auto*
 show *?thesis*
 proof
  fix *j*
  show *cones-map a* χ $j = \chi$ j
   using *assms* χ*.A.value-is-ide* χ*.preserves-hom C.comp-arr-dom* χ*.is-extensional*
   by (*cases J.arr j*, *auto*)
 qed
qed

lemma *cones-map-comp*:
assumes *C.seq f g*
shows *cones-map* ($f \cdot g$) = *restrict* (*cones-map g o cones-map f*) (*cones* (*C.cod f*))
proof (*intro restr-eqI*)
 show *cones* (*C.cod* ($f \cdot g$)) = *cones* (*C.cod f*) **using** *assms* **by** *simp*
 show $\bigwedge\chi$. $\chi \in$ *cones* (*C.cod* ($f \cdot g$)) $\Longrightarrow$
     (λj. *if J.arr j then* χ $j \cdot f \cdot g$ *else C.null*) = (*cones-map g o cones-map f*) χ
 proof −
  fix χ
  assume χ: $\chi \in$ *cones* (*C.cod* ($f \cdot g$))
  show (λj. *if J.arr j then* χ $j \cdot f \cdot g$ *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 $j \cdot f$ *else C.null*) $j \cdot g$ *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 $j \cdot f$ *else C.null*) *g else C.null*)
     using χ *assms* **by** *auto*
   qed
   also have ... = (λj. *if J.arr j then* χ $j \cdot f \cdot g$ *else C.null*)
   proof −
    have $\bigwedge j$. *J.arr j* $\Longrightarrow$ (χ $j \cdot f$) $\cdot g = \chi$ $j \cdot f \cdot g$
    proof −
     interpret χ: *cone J C D* ⟨*C.cod f*⟩ χ **using** *assms* χ **by** *auto*
     fix *j*

```
        assume j: J.arr j
        show (χ j · f) · g = χ j · f · g
          using assms C.comp-assoc by simp
      qed
      thus ?thesis by auto
    qed
    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-magma.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 NaturalTransformation.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.is-extensional 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 ... = $\tau o\chi f.map\ j$
  using $j\ f\ \chi.cone\text{-}axioms\ \tau o\chi f.map\text{-}simp\text{-}2$ **by** *auto*
finally show $D'.cones\text{-}map\ f\ \tau o\chi.map\ j = \tau o\chi f.map\ j$ **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 set category S equipped with an injective mapping ι::$('j \Rightarrow 'c) \Rightarrow 's$ serves this purpose.

locale *cones-functor* =
  C: *category* C +
  Cop: *dual-category* C +
  J: *category* J +
  D: *diagram* $J\ C\ D$ +
  S: *concrete-set-category* S *UNIV* ι
for J :: *'j comp* (**infixr** $\cdot_J$ *55*)
and C :: *'c comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and S :: *'s comp* (**infixr** $\cdot_S$ *55*)
and ι :: $('j \Rightarrow 'c) \Rightarrow 's$
begin

notation *S.in-hom* ($\ll$- : - $\rightarrow_S$ -$\gg$)

abbreviation o **where** o $\equiv$ *S*.o

definition *map* :: $'c \Rightarrow 's$
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 $\in$ ι ‘ *D.cones* (*C.cod f*) $\rightarrow$ ι ‘ *D.cones* (*C.dom f*)
    using *assms D.cones-map-mapsto* **by** *force*

thus *?thesis* **using** *assms* $S.\iota$*-mapsto* **by** *auto*
qed

lemma *map-ide*:
assumes $C.ide\ a$
shows $map\ a = S.mkIde\ (\iota\ `\ D.cones\ a)$
proof –
have $map\ a = S.mkArr\ (\iota\ `\ D.cones\ a)\ (\iota\ `\ D.cones\ a)\ (\iota\ o\ D.cones\text{-}map\ a\ o\ \mathrm{o})$
using *assms map-simp* **by** *force*
also have ... $= S.mkArr\ (\iota\ `\ D.cones\ a)\ (\iota\ `\ D.cones\ a)\ (\lambda x.\ x)$
using $S.\iota$*-mapsto D.cones-map-ide* **by** *force*
also have ... $= S.mkIde\ (\iota\ `\ D.cones\ a)$
using *assms S.mkIde-as-mkArr* $S.\iota$*-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 $map\ (g \cdot^{op} f) = map\ g \cdot_S map\ f$
proof –
have *0*: $S.seq\ (map\ g)\ (map\ f)$
using *assms arr-map* [*of f*] *arr-map* [*of g*] *map-simp*
by (*intro S.seqI*, *auto*)
have $map\ (g \cdot^{op} f) = S.mkArr\ (\iota\ `\ D.cones\ (C.cod\ f))\ (\iota\ `\ D.cones\ (C.dom\ g))$
$((\iota\ o\ D.cones\text{-}map\ g\ o\ \mathrm{o})\ o\ (\iota\ o\ D.cones\text{-}map\ f\ o\ \mathrm{o}))$
proof –
have *1*: $S.arr\ (map\ (g \cdot^{op} f))$
using *assms arr-map* [*of C f g*] **by** *simp*
have $map\ (g \cdot^{op} f) = S.mkArr\ (\iota\ `\ D.cones\ (C.cod\ f))\ (\iota\ `\ D.cones\ (C.dom\ g))$
$(\iota\ o\ D.cones\text{-}map\ (C\ f\ g)\ o\ \mathrm{o})$
using *assms map-simp* [*of C f g*] **by** *simp*
also have ... $= S.mkArr\ (\iota\ `\ D.cones\ (C.cod\ f))\ (\iota\ `\ D.cones\ (C.dom\ g))$
$((\iota\ o\ D.cones\text{-}map\ g\ o\ \mathrm{o})\ o\ (\iota\ o\ D.cones\text{-}map\ f\ o\ \mathrm{o}))$
using *assms 1 calculation D.cones-map-mapsto D.cones-map-comp* **by** *auto*
finally show *?thesis* **by** *blast*
qed
also have ... $= map\ g \cdot_S map\ f$
using *assms 0* **by** (*elim S.seqE*, *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* $\subseteq$ *functor Cop.comp S map* **using** *is-functor* **by** *auto*
sublocale *cones-functor* $\subseteq$ *set-valued-functor Cop.comp S map* **..**

18.3 Limits

18.3.1 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 $J\ C\ D\ a\ \chi$
for J :: *'j comp*   (**infixr** $\cdot_J$ *55*)
and C :: *'c comp*   (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and a :: $'c$
and χ :: $'j \Rightarrow 'c$ +
assumes *is-universal*: *cone* $J\ C\ D\ a'\ \chi' \Longrightarrow \exists!f.\ «f : a' \rightarrow a» \wedge D.\textit{cones-map}\ f\ \chi = \chi'$
begin

definition *induced-arrow* :: $'c \Rightarrow ('j \Rightarrow 'c) \Rightarrow 'c$
where *induced-arrow* $a'\ \chi' = (\textit{THE}\ f.\ «f : a' \rightarrow a» \wedge D.\textit{cones-map}\ f\ \chi = \chi')$

lemma *induced-arrowI*:
assumes χ': $\chi' \in D.\textit{cones}\ a'$
shows $«\textit{induced-arrow}\ a'\ \chi' : a' \rightarrow a»$
and $D.\textit{cones-map}\ (\textit{induced-arrow}\ a'\ \chi')\ \chi = \chi'$
proof –
  have $\exists!f.\ «f : a' \rightarrow a» \wedge D.\textit{cones-map}\ f\ \chi = \chi'$
    using *assms* χ' *is-universal* **by** *simp*
  hence *1*: $«\textit{induced-arrow}\ a'\ \chi' : a' \rightarrow a» \wedge D.\textit{cones-map}\ (\textit{induced-arrow}\ a'\ \chi')\ \chi = \chi'$
    using *theI'* [*of* $\lambda f.\ «f : a' \rightarrow a» \wedge D.\textit{cones-map}\ f\ \chi = \chi'$] *induced-arrow-def*
    by *presburger*
  show $«\textit{induced-arrow}\ a'\ \chi' : a' \rightarrow a»$ **using** *1* **by** *simp*
  show $D.\textit{cones-map}\ (\textit{induced-arrow}\ a'\ \chi')\ \chi = \chi'$ **using** *1* **by** *simp*
qed

lemma *cones-map-induced-arrow*:
shows *induced-arrow* $a' \in$ *D.cones* $a' \rightarrow$ *C.hom* a' a
and $\bigwedge\chi'$. $\chi' \in$ *D.cones* $a' \Longrightarrow$ *D.cones-map* (*induced-arrow* a' χ') $\chi = \chi'$
  using *induced-arrowI* **by** *auto*

lemma *induced-arrow-cones-map*:
assumes *C.ide* a'
shows (λf. *D.cones-map* f χ) $\in$ *C.hom* a' $a \rightarrow$ *D.cones* a'
and $\bigwedge f$. «$f : a' \rightarrow a$» $\Longrightarrow$ *induced-arrow* a' (*D.cones-map* f χ) $= f$
proof –
  have a': *C.ide* a' **using** *assms* **by** (*simp add*: *cone.ide-apex*)
  have *cone-*χ: *cone J C D a* χ **..**
  show (λf. *D.cones-map* f χ) $\in$ *C.hom* a' $a \rightarrow$ *D.cones* a'
    using *cone-*χ *D.cones-map-mapsto* **by** *blast*
  fix f
  assume f: «$f : a' \rightarrow a$»
  show *induced-arrow* a' (*D.cones-map* f χ) $= f$
  proof –
    have *D.cones-map* f $\chi \in$ *D.cones* a'
      using f *cone-*χ *D.cones-map-mapsto* **by** *blast*
    hence $\exists!f'$. «$f' : a' \rightarrow a$» $\wedge$ *D.cones-map* f' $\chi =$ *D.cones-map* f χ
      using *assms is-universal* **by** *auto*
    thus *?thesis*
      using f *induced-arrow-def*
          the1-equality [*of* $\lambda f'$. «$f' : a' \rightarrow a$» $\wedge$ *D.cones-map* f' $\chi =$ *D.cones-map* f χ]
      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-betw* (λf. *D.cones-map* f χ) (*C.hom* a' a) (*D.cones* a')
proof (*intro bij-betwI*)
  show (λf. *D.cones-map* f χ) $\in$ *C.hom* a' $a \rightarrow$ *D.cones* a'
    using *assms induced-arrow-cones-map* **by** *blast*
  show *induced-arrow* $a' \in$ *D.cones* $a' \rightarrow$ *C.hom* a' a
    using *assms cones-map-induced-arrow* **by** *blast*
  show $\bigwedge f$. $f \in$ *C.hom* a' $a \Longrightarrow$ *induced-arrow* a' (*D.cones-map* f χ) $= f$
    using *assms induced-arrow-cones-map* **by** *blast*
  show $\bigwedge\chi'$. $\chi' \in$ *D.cones* $a' \Longrightarrow$ *D.cones-map* (*induced-arrow* a' χ') $\chi = \chi'$
    using *assms cones-map-induced-arrow* **by** *blast*
qed

lemma *induced-arrow-eqI*:
assumes *D.cone* a' χ' **and** «$f : a' \rightarrow a$» **and** *D.cones-map* f $\chi = \chi'$
shows *induced-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-arrow a* $\chi = a$
proof −
have $\ll a : a \rightarrow a \gg \wedge D.cones\text{-}map\ a\ \chi = \chi$
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* $\chi \equiv$ *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* :: $'c \Rightarrow bool$
where *has-as-limit a* $\equiv (\exists \chi.$ *limit-cone a* $\chi)$

abbreviation *has-limit*
where *has-limit* $\equiv (\exists a\ \chi.$ *limit-cone a* $\chi)$

definition *some-limit* :: $'c$
where *some-limit* = (*SOME a*. $\exists \chi.$ *limit-cone a* χ)

definition *some-limit-cone* :: $'j \Rightarrow 'c$
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 $\exists a.$ *has-as-limit a* **using** *assms* **by** *simp*
hence *has-as-limit some-limit*
using *some-limit-def someI-ex* [*of* $\lambda a. \exists \chi.$ *limit-cone a* χ] **by** *simp*
thus *limit-cone some-limit some-limit-cone*
using *assms some-limit-cone-def someI-ex* [*of* $\lambda\chi.$ *limit-cone some-limit* χ]
by *simp*
qed

lemma *ex-limitE*:
assumes $\exists a.$ *has-as-limit a*
obtains $a\ \chi$ **where** *limit-cone a* χ
using *assms someI-ex* **by** *blast*

end

18.3.2 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: *concrete-set-category S UNIV* ι +
  Cones: *cones-functor J C D S* ι +
  Hom: *hom-functor C S* φ +
  representation-of-functor C S φ *Cones.map a* Φ
for J :: $'j$ *comp* (**infixr** $\cdot_J$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and ι :: $('j \Rightarrow 'c) \Rightarrow 's$
and a :: $'c$
and Φ :: $'c \Rightarrow 's$

18.3.3 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: *concrete-set-category S UNIV* ι +
  Cones: *cones-functor J C D S* ι +
  Hom: *hom-functor C S* φ +
  Φ: *representation-of-functor C S* φ *Cones.map a* Φ +
  χ: *limit-cone J C D a* χ
for J :: $'j$ *comp* (**infixr** $\cdot_J$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and φ :: $'c * 'c \Rightarrow 'c \Rightarrow 's$
and ι :: $('j \Rightarrow 'c) \Rightarrow 's$
and a :: $'c$
and Φ :: $'c \Rightarrow 's$
and χ :: $'j \Rightarrow 'c$ +
assumes χ-*in-terms-of*-Φ: $\chi = S.o\ (S.Fun\ (\Phi\ a)\ (\varphi\ (a,\ a)\ a))$
and Φ-*in-terms-of*-χ:
    $Cop.ide\ a' \Longrightarrow \Phi\ a' = S.mkArr\ (Hom.set\ (a',\ a))\ (\iota\ \text{`}\ D.cones\ a')$

$(\lambda x.\ \iota\ (D.cones\text{-}map\ (Hom.\psi\ (a', a)\ x)\ \chi))$

The assumption χ-*in-terms-of*-Φ states that the universal cone χ is obtained by applying the function $S.Fun\ (\Phi\ a)$ 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.

18.3.4 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 $'c + ('j \Rightarrow 'c)$-sets for this.

```
context limit-cone
begin

  interpretation Cop: dual-category C ..
  interpretation CopxC: product-category Cop.comp C ..
  interpretation S: set-category ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
    using SetCat.is-set-category by auto

  interpretation S: concrete-set-category ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
                                          UNIV ‹UP o Inr›
    apply unfold-locales
    using UP-mapsto
     apply auto[1]
    using inj-UP inj-Inr inj-compose
    by metis

  notation SetCat.comp      (infixr ·S 55)

   interpretation Cones: cones-functor J C D ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr
comp›
                                ‹UP o Inr› ..

  interpretation Hom: hom-functor C ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
                                ‹λ-. UP o Inl›
    apply (unfold-locales)
    using UP-mapsto
     apply auto[1]
    using SetCat.inj-UP injD inj-onI inj-Inl inj-compose
    by (metis (no-types, lifting))

  interpretation Y: yoneda-functor C ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
                                ‹λ-. UP o Inl› ..
  interpretation Ya: yoneda-functor-fixed-object
                    C ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
```

$\langle\lambda$-. $UP\ o\ Inl\rangle\ a$
apply (*unfold-locales*) **using** *ide-apex* **by** *auto*

abbreviation $inl :: {'c} \Rightarrow {'c} + ({'j} \Rightarrow {'c})$ **where** $inl \equiv Inl$
abbreviation $inr :: ({'j} \Rightarrow {'c}) \Rightarrow {'c} + ({'j} \Rightarrow {'c})$ **where** $inr \equiv Inr$
abbreviation ι **where** $\iota \equiv UP\ o\ inr$
abbreviation o **where** $\mathrm{o} \equiv Cones.\mathrm{o}$
abbreviation φ **where** $\varphi \equiv \lambda$-. $UP\ o\ inl$
abbreviation ψ **where** $\psi \equiv Hom.\psi$
abbreviation Y **where** $Y \equiv Y.Y$

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 *cone-*χ [*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 Φ*o-in-hom*:
assumes a': $C.ide\ a'$
shows $\ll \Phi o\ a' : S.mkIde\ (Hom.set\ (a',\ a)) \rightarrow_S S.mkIde\ (\iota\ `\ D.cones\ a')\gg$
proof $-$
have $\ll S.mkArr\ (Hom.set\ (a',\ a))\ (\iota\ `\ D.cones\ a')\ (\lambda x.\ \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)) :$
$S.mkIde\ (Hom.set\ (a',\ a)) \rightarrow_S S.mkIde\ (\iota\ `\ D.cones\ a')\gg$
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 $\ll \psi\ (a',\ a)\ x : a' \rightarrow a\gg$
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 cone-*χ **by** *force*
thus ι (*D.cones-map* (ψ (a', a) x) χ) $\in \iota$ ‘ *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 ι ‘ *D.cones* a' $\subseteq$ *S.Univ*
using *UP-mapsto* **by** *auto*
ultimately show *?thesis* **using** *S.mkArr-in-hom* **by** *simp*
qed
thus *?thesis* **using** Φ*o-def* [*of* a'] **by** *auto*
qed

interpretation Φ: *transformation-by-components*
Cop.comp SetCat.comp ‹*Y a*› *Cones.map* Φ*o*
proof
fix a'
assume A': *Cop.ide* a'
show «Φ*o* a' : *Y a* a' $\rightarrow_S$ *Cones.map* a'»
using A' *Ya-ide* Φ*o-in-hom Cones.map-ide* **by** *auto*
next
fix g
assume g: *Cop.arr* g
show Φ*o* (*Cop.cod* g) $\cdot_S$ *Y a g* = *Cones.map* g $\cdot_S$ Φ*o* (*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*) $\wedge$ *Y a g* = *S.mkArr ?A ?B ?F*
using *ide-apex g Ya.preserves-arr Ya-arr* **by** *fastforce*
moreover have *S.arr* (Φ*o* (*Cop.cod* g))
using g Φ*o-in-hom* [*of Cop.cod g*] **by** *auto*
moreover have Φ*o* (*Cop.cod* g) = *S.mkArr ?B ?C ?G*
using g Φ*o-def* [*of C.dom g*] **by** *auto*
moreover have *S.seq* (Φ*o* (*Cop.cod* g)) (*Y a g*)
using *ide-apex g* Φ*o-in-hom* [*of Cop.cod g*] **by** *auto*
ultimately have *1*: *S.seq* (Φ*o* (*Cop.cod* g)) (*Y a g*) $\wedge$
Φ*o* (*Cop.cod* g) $\cdot_S$ *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 Φ*o* (*Cop.dom* g) = *S.mkArr ?A ?B' ?G'*
using g Φ*o-def* **by** *fastforce*
moreover have *S.seq* (*Cones.map* g) (Φ*o* (*Cop.dom* g))

using g *Cones.preserves-hom* [*of* g *C.cod* g *C.dom* g] Φ*o-in-hom* [*of Cop.dom* g]
by *force*
ultimately have
2: *S.seq* (*Cones.map* g) (Φo (*Cop.dom* g)) $\wedge$
Cones.map $g \cdot_S \Phi o$ (*Cop.dom* g) = *S.mkArr* ?*A* ?*C* (?*F'* o ?*G'*)
using *S.seqI'* [*of* Φo (*Cop.dom* g) *Cones.map* g] **by** *force*

have Φo (*Cop.cod* g) $\cdot_S$ *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 $\bigwedge x.\ x \in$?*A* $\Longrightarrow$ (?*G* o ?*F*) x = (?*F'* o ?*G'*) x
proof −
fix x
assume x: $x \in$?*A*
hence *1*: «ψ (*C.cod* g, a) x : *C.cod* $g \rightarrow a$»
using *ide-apex* g *Hom.*ψ*-mapsto* [*of C.cod* g a] **by** *auto*
have (?*G* o ?*F*) $x = \iota$ (*D.cones-map* (ψ (*C.dom* g, a)
(φ (*C.dom* g, a) (ψ (*C.cod* g, a) $x \cdot g$))) 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) $x \cdot g$))) χ)
proof −
have ?*F* $x = \varphi$ (*C.dom* g, a) (ψ (*C.cod* g, a) $x \cdot g$) **by** *simp*
thus ?*thesis* **by** *presburger*
qed
finally show ?*thesis* **by** *auto*
qed
also have ... = ι (*D.cones-map* (ψ (*C.cod* g, a) $x \cdot g$) χ)
proof −
have «ψ (*C.cod* g, a) $x \cdot g$: *C.dom* $g \rightarrow a$» **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 cone-*χ *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 cone-*χ *D.cones-map-mapsto S.*o-ι **by** *simp*
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* $g \cdot_S \Phi o$ (*Cop.dom* g)
using *2* **by** *auto*
finally show ?*thesis* **by** *auto*
qed
qed

interpretation Φ: *set-valued-transformation*
$\quad$ *Cop.comp SetCat.comp* $\langle Y\ a \rangle$ *Cones.map* $\Phi.map$ **..**

interpretation Φ: *natural-isomorphism Cop.comp SetCat.comp* $\langle Y\ a \rangle$ *Cones.map* $\Phi.map$
proof
 fix a'
 assume a': *Cop.ide* a'
 show *S.iso* $(\Phi.map\ a')$
 proof –
  let $?F = \lambda x.\ \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
  have *bij*: *bij-betw* $?F\ (Hom.set\ (a',\ a))\ (\iota\ `\ D.cones\ a')$
  proof –
   have $\bigwedge x\ x'.\ [\![\ x \in Hom.set\ (a',\ a);\ x' \in Hom.set\ (a',\ a);$
    $\iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x')\ \chi)\ ]\!]$
     $\Longrightarrow x = x'$
   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: $\ll\psi\ (a',\ a)\ x : a' \rightarrow a\gg$ **using** x *ide-apex* a' *Hom.*ψ*-mapsto* **by** *auto*
    have $\psi x'$: $\ll\psi\ (a',\ a)\ x' : a' \rightarrow a\gg$ **using** x' *ide-apex* a' *Hom.*ψ*-mapsto* **by** *auto*
    have *1*: $\exists!f.\ \ll f : a' \rightarrow a\gg \wedge\ \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
    proof –
     have $D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi \in D.cones\ a'$
      using ψx a' *cone-*χ *D.cones-map-mapsto* **by** *force*
     hence *2*: $\exists!f.\ \ll f : a' \rightarrow a\gg \wedge D.cones\text{-}map\ f\ \chi = D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi$
      using a' *is-universal* **by** *simp*
     show $\exists!f.\ \ll f : a' \rightarrow a\gg \wedge\ \iota\ (D.cones\text{-}map\ f\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
     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 S.inj-*ι *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'': $\ll f'' : a' \rightarrow a\gg \wedge\ \iota\ (D.cones\text{-}map\ f''\ \chi) = \iota\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ x)\ \chi)$
      by *blast*

```
      have φ (a′, a) f′′ ∈ Hom.set (a′, a) ∧
           ι (D.cones-map (ψ (a′, a) (φ (a′, a) f′′)) χ) = ι (D.cones-map (ψ (a′, a) x) χ)
      proof
        show φ (a′, a) f′′ ∈ Hom.set (a′, a) using f′′ Hom.set-def by auto
        show ι (D.cones-map (ψ (a′, a) (φ (a′, a) f′′)) χ) =
                ι (D.cones-map (ψ (a′, a) x) χ)
          using f′′ Hom.ψ-φ by presburger
      qed
      moreover have
         ⋀x′′. x′′ ∈ Hom.set (a′, a) ∧
                ι (D.cones-map (ψ (a′, a) x′′) χ) = ι (D.cones-map (ψ (a′, a) x) χ)
                  ⟹ x′′ = φ (a′, a) f′′
      proof −
        fix x′′
        assume x′′: x′′ ∈ Hom.set (a′, a) ∧
                  ι (D.cones-map (ψ (a′, a) x′′) χ) = ι (D.cones-map (ψ (a′, a) x) χ)
        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′ cone-χ 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 S.o-ι 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′ S.ι-o 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 S.arr ?f
      using ide-apex a′ Hom.set-subset-Univ S.ι-mapsto S.arr-mkArr by auto
    thus ?thesis using bij S.iso-char by fastforce
  qed
  moreover have ?f = Φ.map a′
    using a′ Φo-def by force
  finally show ?thesis by auto
 qed
qed

interpretation R: representation-of-functor
                  C ‹SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp›
                  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 Φo-def Φ.preserves-reflects-arr [of a]
      by simp
    also have ... = ι (D.cones-map a χ)
    proof −
      have φ (a, a) a ∈ Hom.set (a, a)
        using ide-apex Hom.φ-mapsto by fastforce
      hence (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)
        using restrict-apply′ [of φ (a, a) a Hom.set (a, a)] by blast
      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] cone-χ 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 cone-χ S.o-ι 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 (SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp) φ ι a Φ χ
  proof
    show χ = o (Φ.FUN a (φ (a, a) a)) using χ-in-terms-of-Φ by auto
    fix a′
    show Cop.ide a′ ⟹ Φ.map a′ = S.mkArr (Hom.set (a′, a)) (ι ‘ D.cones a′)
                                         (λx. ι (D.cones-map (ψ (a′, a) x) χ))
      using Φ.map-simp-ide Φo-def [of a′] by force
  qed

  no-notation SetCat.comp      (infixr ·S 55)

end

sublocale limit-cone ⊆ limit-situation J C D SetCat.comp :: ('c + ('j ⇒ 'c)) setcat.arr comp
                              φ ι a Φ χ
  using induces-limit-situation by auto
```

18.3.5 Representations of the Cones Functor Induce Limit Situations

context *representation-of-cones-functor*
begin

interpretation Φ: *set-valued-transformation Cop.comp S* $\langle Y\ a\rangle$ *Cones.map* Φ **..**
interpretation Ψ: *inverse-transformation Cop.comp S* $\langle Y\ a\rangle$ *Cones.map* Φ **..**
interpretation Ψ: *set-valued-transformation Cop.comp S Cones.map* $\langle Y\ a\rangle$ Ψ*.map* **..**

abbreviation o
where o $\equiv$ *Cones*.o

abbreviation χ
where $\chi \equiv$ o (*S.Fun* (Φ *a*) (φ (*a*, *a*) *a*))

lemma *Cones-SET-eq-ι-img-cones*:
assumes *C.ide* a'
shows *Cones.SET* $a' = \iota$ ‘ *D.cones* a'
proof –
  have ι ‘ *D.cones* $a' \subseteq$ *S.Univ* **using** *S.ι-mapsto* **by** *auto*
  thus *?thesis* **using** *assms Cones.map-ide* **by** *auto*
qed

lemma $\iota\chi$:
shows $\iota\ \chi =$ *S.Fun* (Φ *a*) (φ (*a*, *a*) *a*)
proof –
  have *S.Fun* (Φ *a*) (φ (*a*, *a*) *a*) $\in$ *Cones.SET a*
    using *Ya.ide-a Hom.φ-mapsto S.Fun-mapsto* [*of* Φ *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* Φ *a*] *Hom.φ-mapsto Hom.set-map*
        Cones-SET-eq-ι-img-cones **by** *fastforce*
  thus *D.cone a* χ
    by (*metis S*.o-ι *UNIV-I imageE 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 Φ*.FUN* a' *x* = *Cones.FUN* (ψ (a', *a*) *x*) ($\iota\ \chi$)
proof –
  have ψx: «ψ (a', *a*) *x* : $a' \rightarrow a$»
    using *Ya.ide-a* a' *x Hom.ψ-mapsto* **by** *blast*

```
    have φa: φ (a, a) a ∈ Hom.set (a, a) using Ya.ide-a Hom.φ-mapsto by fastforce
    have Φ.FUN a' x = (Φ.FUN a' o Ya.FUN (ψ (a', a) x)) (φ (a, a) a)
    proof −
      have φ (a', a) (a · ψ (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))
                          (φ (a', a) ∘ Cop.comp (ψ (a', a) x) ∘ ψ (a, a)))
        using Ya.ide-a a' Hom.set-subset-Univ Hom.ψ-mapsto [of a a] Hom.φ-mapsto ψx
        by force
      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 (ψ (a', a) x) o Φ.FUN a) (φ (a, a) a)
    proof −
      have (Φ.FUN a' o Ya.FUN (ψ (a', a) x)) (φ (a, a) a)
              = S.Fun (Φ a' ·S Y a (ψ (a', a) x)) (φ (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 (ψ (a', a) x)) (Φ a)) (φ (a, a) a)
        using ψx is-natural-1 [of ψ (a', a) x] is-natural-2 [of ψ (a', a) x] by auto
      also have ... = (Cones.FUN (ψ (a', a) x) o Φ.FUN a) (φ (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' $\iota\chi'$ *Ya.ide-a Hom.set-map* **by** *auto*
qed
show f: $\ll$*?f* : $a' \rightarrow a$$\gg$ **using** *A* a' *Ya.ide-a Hom.*ψ*-mapsto* [*of* a' *a*] **by** *auto*
have *E*: $\bigwedge f.$ $\ll f : a' \rightarrow a\gg \Longrightarrow$ *Cones.FUN* f $(\iota\ \chi) = \Phi$*.FUN* a' $(\varphi\ (a', a)\ f)$
proof −
fix f
assume f: $\ll f : a' \rightarrow a\gg$
have $\varphi\ (a', a)\ f \in$ *Hom.set* (a', a)
using a' *Ya.ide-a f Hom.*φ*-mapsto* **by** *auto*
thus *Cones.FUN* f $(\iota\ \chi) = \Phi$*.FUN* a' $(\varphi\ (a', a)\ f)$
using a' f Φ*-FUN-simp* **by** *simp*
qed
have *I*: Φ*.FUN* a' (Ψ*.FUN* a' $(\iota\ \chi')) = \iota\ \chi'$
proof −
have Φ*.FUN* a' (Ψ*.FUN* a' $(\iota\ \chi'))$ =
compose (Ψ*.DOM* a') (Φ*.FUN* a') (Ψ*.FUN* a') $(\iota\ \chi')$
using a' $\iota\chi'$ *Cones.map-ide* Ψ*.preserves-hom* [*of* a' a' a'] **by** *force*
also have ... = $(\lambda x \in \Psi$*.DOM* $a'.\ x)$ $(\iota\ \chi')$
using a' Ψ*.inverts-components S.inverse-arrows-char* **by** *force*
also have ... = $\iota\ \chi'$
using a' $\iota\chi'$ *Cones.map-ide* Ψ*.preserves-hom* [*of* a' a' a'] **by** *force*
finally show *?thesis* **by** *auto*
qed
show $f\chi$: *D.cones-map ?f* $\chi = \chi'$
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 $[\![\ll f' : a' \rightarrow a\gg$; *D.cones-map* f' $\chi = \chi'$ $]\!] \Longrightarrow f' =$ *?f*
proof −
assume f': $\ll f' : a' \rightarrow a\gg$ **and** $f'\chi$: *D.cones-map* f' $\chi = \chi'$
show $f' =$ *?f*
proof −
have *1*: $\varphi\ (a', a)\ f' \in$ *Hom.set* $(a', a) \wedge \varphi\ (a', a)$ *?f* $\in$ *Hom.set* (a', a)
using *Ya.ide-a* a' f f' *Hom.*φ*-mapsto* **by** *auto*
have *S.iso* $(\Phi\ a')$ **using** χ'*.ide-apex components-are-iso* **by** *auto*
hence *2*: *S.arr* $(\Phi\ a') \wedge$ *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' $(\varphi\ (a', a)\ f') = \Phi$*.FUN* a' $(\varphi\ (a', a)$ *?f*$)$
proof −
have Φ*.FUN* a' $(\varphi\ (a', a)$ *?f*$) = \iota\ \chi'$
using *A I Hom.*φ*-*ψ *Ya.ide-a* a' **by** *simp*
also have ... = *Cones.FUN* f' $(\iota\ \chi)$
using f f' *A E cone-*χ *Cones.preserves-arr* $f\chi$ $f'\chi$ **by** (*elim C.in-homE*, *auto*)
also have ... = Φ*.FUN* a' $(\varphi\ (a', a)\ f')$

```
        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 by (elim S.in-homE, auto)
      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
```

18.4 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* (- :: $'j$) $\equiv \forall J :: 'j$ *comp. category* $J \longrightarrow$ *has-limits-of-shape* J

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** $\varphi\ \psi$ **where** *IF*: *inverse-functors* $J\ J'\ \varphi\ \psi$
    using *isomorphic-categories-def isomorphic-categories-axioms-def* **by** *blast*
  interpret *IF*: *inverse-functors* $J\ J'\ \varphi\ \psi$ **using** *IF* **by** *auto*
  have $\psi\varphi$: $\psi\ o\ \varphi = J.map$ **using** *IF.inv* **by** *metis*
  have $\varphi\psi$: $\varphi\ o\ \psi = J'.map$ **using** *IF.inv'* **by** *metis*
  have $\bigwedge D'.$ *diagram* $J'\ C\ D' \Longrightarrow \exists\, a\ \chi.$ *limit-cone* $J'\ C\ D'\ a\ \chi$
  proof −
    fix D'
    assume D': *diagram* $J'\ C\ D'$
    interpret D': *diagram* $J'\ C\ D'$ **using** D' **by** *auto*
    interpret D: *composite-functor* $J\ J'\ C\ \varphi\ D'$ **..**
    interpret D: *diagram* $J\ C\ \langle D'\ o\ \varphi\rangle$ **..**
    have D: *diagram* $J\ C\ (D'\ o\ \varphi)$ **..**
    from *assms*(*1*) **obtain** $a\ \chi$ **where** χ: $D.$*limit-cone* $a\ \chi$
      using D *has-limits-of-shape-def* **by** *blast*
    interpret χ: *limit-cone* $J\ C\ \langle D'\ o\ \varphi\rangle\ a\ \chi$ **using** χ **by** *auto*
    interpret A': *constant-functor* $J'\ C\ a$
      using χ.*ide-apex* **by** (*unfold-locales*, *auto*)
    have $\chi o\psi$: *cone* $J'\ C\ (D'\ o\ \varphi\ o\ \psi)\ a\ (\chi\ o\ \psi)$
      using *comp-cone-functor IF.G.functor-axioms* χ.*cone-axioms* **by** *fastforce*
    hence $\chi o\psi$: *cone* $J'\ C\ D'\ a\ (\chi\ o\ \psi)$
      using $\varphi\psi$ **by** (*metis* D'.*functor-axioms Fun.comp-assoc comp-functor-identity*)
    interpret $\chi o\psi$: *cone* $J'\ C\ D'\ a\ \langle\chi\ o\ \psi\rangle$ **using** $\chi o\psi$ **by** *auto*
    interpret $\chi o\psi$: *limit-cone* $J'\ C\ D'\ a\ \langle\chi\ o\ \psi\rangle$
    proof
      fix $a'\ \chi'$
      assume χ': D'.*cone* $a'\ \chi'$
      interpret χ': *cone* $J'\ C\ D'\ a'\ \chi'$ **using** χ' **by** *auto*
      have $\chi' o\varphi$: *cone* $J\ C\ (D'\ o\ \varphi)\ a'\ (\chi'\ o\ \varphi)$
        using χ' *comp-cone-functor IF.F.functor-axioms* **by** *fastforce*
      interpret $\chi' o\varphi$: *cone* $J\ C\ \langle D'\ o\ \varphi\rangle\ a'\ \langle\chi'\ o\ \varphi\rangle$ **using** $\chi' o\varphi$ **by** *auto*
      have *cone* $J\ C\ (D'\ o\ \varphi)\ a'\ (\chi'\ o\ \varphi)$ **..**
      hence *1*: $\exists !f.\ \ll f : a' \rightarrow a\gg \wedge\ D.$*cones-map* $f\ \chi = \chi'\ o\ \varphi$
        using χ.*is-universal* **by** *simp*
      show $\exists !f.\ \ll f : a' \rightarrow a\gg \wedge\ D'.$*cones-map* $f\ (\chi\ o\ \psi) = \chi'$
      proof
        let $?f = THE\ f.\ \ll f : a' \rightarrow a\gg \wedge\ D.$*cones-map* $f\ \chi = \chi'\ o\ \varphi$
        have f: $\ll ?f : a' \rightarrow a\gg \wedge\ D.$*cones-map* $?f\ \chi = \chi'\ o\ \varphi$

  using *1 theI'* [*of* λf. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ' *o* φ] **by** *blast*
have *f-in-hom*: «*?f* : *a'* → *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'*.is-extensional* **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* $\chi o \psi$ **by** *fastforce*
    also have ... = *D.cones-map ?f* χ (ψ *j'*)
      using *j' f-in-hom* χ χ*.cone-*χ **by** *fastforce*
    also have ... = χ' *j'*
      using *j' f* χ $\varphi\psi$ *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 $\chi' o \varphi$*.is-extensional* χ*.cone-*χ *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*.cone-*χ **by** *auto*
    also have ... = *C* ((χ *o* ψ) (φ *j*)) *f'*
      using *j f'* $\psi\varphi$ **by** (*metis comp-apply J.map-simp*)
    also have ... = *D'.cones-map f'* (χ *o* ψ) (φ *j*)
      using *j f'* $\chi o \psi$ **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 $\ll f' : a' \rightarrow a\gg \wedge$ *D.cones-map* $f'\ \chi = \chi'\ o\ \varphi$
  using f' **by** *auto*
moreover have $\bigwedge P\ x\ x'.\ (\exists!x.\ P\ x) \wedge P\ x \wedge P\ x' \Longrightarrow x = x'$
  by *auto*
ultimately show $f' = ?f$ **using** *1 f* **by** *blast*
qed
qed
have *limit-cone* $J'\ C\ D'\ a\ (\chi\ o\ \psi)$ ..
thus $\exists\ a\ \chi.$ *limit-cone* $J'\ C\ D'\ a\ \chi$ **by** *blast*
qed
thus *?thesis* **using** *has-limits-of-shape-def* **by** *auto*
qed

end

18.4.1 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** $\cdot_J$ *55*)
and C :: *'c comp* (**infixr** $\cdot$ *55*)
begin

notation *J.in-hom* $(\ll\text{-} : \text{-} \rightarrow_J \text{-}\gg)$
notation *J-C.comp* (**infixr** $\cdot_{[J,C]}$ *55*)
notation *J-C.in-hom* $(\ll\text{-} : \text{-} \rightarrow_{[J,C]} \text{-}\gg)$

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 $\neg$ *C.arr f* $\Longrightarrow$ *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*)
apply *unfold-locales* **using** *f* **by** *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*)
apply *unfold-locales* **using** *f* **by** *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.is-extensional* **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* ($g \cdot f$) = *map g* $\cdot_{[J,C]}$ *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)
        apply unfold-locales using g by 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)
      proof -
        have Fun-fg.map = (λj. if J.arr j then g · f else C.null)
          using 1 f g Fun-fg.map-def by auto
        thus ?thesis by auto
      qed
      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

end

sublocale diagonal-functor ⊆ functor C J-C.comp map
  using is-functor by auto

context diagonal-functor
begin
```

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*
  proof −
    have *J-C.dom x* = *map a* **using** *x.arrow* **by** *blast*
    hence *J-C.Map* (*J-C.dom x*) = *J-C.Map* (*map a*) **by** *simp*
    hence *J-C.Dom x* = *J-C.Map* (*map a*)
      using *A.value-is-ide x.arrow J-C.in-homE* **by** (*metis J-C.Map-dom*)
    moreover have *J-C.Map* (*map a*) = α.*map*
      using *A.value-is-ide preserves-ide map-def* **by** *simp*
    ultimately show *?thesis* **using** α.*map-def A.map-def* **by** *auto*
  qed
  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* $\rightarrow_{[J,C]}$ *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* $\rightarrow_{[J,C]}$ *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 : $a' \rightarrow a$» **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 →[J,C] J-C.MkIde D»
  proof −
    have 1: map a = J-C.MkIde A.map
      using χ.ide-apex A.equals-dom-if-value-is-ide A.equals-cod-if-value-is-ide map-def
      by auto
    have J-C.arr ?x using x by blast
    moreover have J-C.dom ?x = map a
     using x J-C.dom-char 1 x χ.ide-apex A.equals-dom-if-value-is-ide χ.D.functor-axioms
          J-C.ide-char
      by auto
    moreover have J-C.cod ?x = J-C.MkIde D using x J-C.cod-char by auto
    ultimately show ?thesis by fast
  qed
  have 4: «?x' : map a' →[J,C] J-C.MkIde D»
  proof −
    have 1: map a' = J-C.MkIde A'.map
      using χ'.ide-apex A'.equals-dom-if-value-is-ide A'.equals-cod-if-value-is-ide map-def
      by auto
    have J-C.arr ?x' using x' by blast
    moreover have J-C.dom ?x' = map a'
    using x' J-C.dom-char 1 x' χ'.ide-apex A'.equals-dom-if-value-is-ide χ.D.functor-axioms
          J-C.ide-char
      by force
    moreover have J-C.cod ?x' = J-C.MkIde D using x' J-C.cod-char by auto
    ultimately show ?thesis by fast
  qed
  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 ·[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 ·[J,C] map g))
      using assms(1) 2 by auto
```

also have ... = $J\text{-}C.Dom$ (*?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 $J\text{-}C.Cod$ *?x'* = $J\text{-}C.Cod$ (*?x* $\cdot_{[J,C]}$ *map g*)
proof −
have $J\text{-}C.Cod$ *?x'* = $J\text{-}C.Cod$ ($J\text{-}C.cod$ *?x'*)
using *x' J-C.Cod-cod* by *simp*
also have ... = $J\text{-}C.Cod$ ($J\text{-}C.MkIde$ *D*)
using *4* by *force*
also have ... = $J\text{-}C.Cod$ ($J\text{-}C.cod$ (*?x* $\cdot_{[J,C]}$ *map g*))
using *2 3 J-C.cod-comp J-C.in-homE* by *metis*
also have ... = $J\text{-}C.Cod$ (*?x* $\cdot_{[J,C]}$ *map g*)
using *seq-xg J-C.Cod-cod J-C.seqI'* by *blast*
finally show *?thesis* by *auto*
qed
show $J\text{-}C.Map$ *?x'* = $J\text{-}C.Map$ (*?x* $\cdot_{[J,C]}$ *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'.is-extensional g.is-extensional*
apply (*unfold-locales*, *auto*)
by (*elim J.seqE*, *auto*)
have $J\text{-}C.Map$ (*?x* $\cdot_{[J,C]}$ *map g*) = $\chi og.map$
using *assms*(*1*) *2 J-C.comp-char map-def* by *auto*
also have ... = $J\text{-}C.Map$ *?x'*
using *x'* $\chi og.map\text{-}def$ *J-C.arr-char* [*of ?x'*] *natural-transformation.is-extensional*
assms(*1*) *cone-*χ $\chi og.map\text{-}simp\text{-}2$
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*: $\ll g : a' \rightarrow a\gg$
and *x'*: $\ll x' : map\ a' \rightarrow_{[J,C]} d\gg$
shows *arrow-from-functor.is-coext C J-C.comp map a x a' x' g*
$\longleftrightarrow$ $J\text{-}C.Map\ x' = diagram.cones\text{-}map\ J\ C\ (J\text{-}C.Map\ d)\ g\ (J\text{-}C.Map\ x)$
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.is-extensional* **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*: $x' = x \cdot_{[J,C]}$ *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*)) = $x \cdot_{[J,C]}$ *map*
g
        using *d g cones-map-is-composition arrow-determines-cone(2) χ.cone-axioms*
            x.arrow-from-functor-axioms
        by *auto*
      hence *f1*: *J-C.MkArr A'.map* (*J-C.Map d*) (*D.cones-map g* (*J-C.Map x*)) = *x'*
        by (*metis 1*)
      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*
      thus *?thesis*
        using *f1* **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*: $x \cdot_{[J,C]}$ *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.is-extensional* **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** $\cdot_J$ *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

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'
  proof -
    fix g
    assume g: «g : a' →C ?a»
    show x.is-coext a' ?x' g ⟷ D.cones-map g ?χ = χ'
    proof -
      have «?x' : Δ.map a' →[J,C] ?d»
        using x'.arrow by simp
      thus ?thesis
        using 3 g Δ.coextension-iff-cones-map [of ?a ?d]
        by (metis (no-types, lifting) 1 2 Δ.cone-determines-arrow(2) cone-χ')
    qed
  qed
  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) 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 meta-adjunction J-C.comp C Δ.map Δ.G Δ.φ Δ.ψ ..
    hence 1: adjoint-functors J-C.comp C Δ.map Δ.G
      using adjoint-functors-def by blast
    interpret G: right-adjoint-to-diagonal-functor J C Δ.G Δ.φ Δ.ψ
      using 1 by (unfold-locales, auto)
    have ⋀D. diagram J C D ⟹ ∃ a. diagram.has-as-limit J C D a
      using A G.gives-limits by blast
    hence ⋀D. diagram J C D ⟹ ∃ a χ. limit-cone J C D a χ
      by metis
    thus has-limits-of-shape J using 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** *auto*
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 $\bigwedge a'\ x'$. *arrow-from-functor C J-C.comp* Δ*.map* a' *d* $x' \Longrightarrow \exists!g$. *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 $?\chi'$ = *J-C.Map* x'
interpret χ': *cone J C ?D* a' $?\chi'$
using *D* x' Δ*.arrow-determines-cone* **by** *auto*
have *cone-*χ': *cone J C ?D* a' $?\chi'$ **..**
let *?g* = χ*.induced-arrow* a' $?\chi'$
show $\exists!g$. *x.is-coext* a' x' *g*
proof
show *x.is-coext* a' x' *?g*
proof (*unfold x.is-coext-def*)
have *1*: «*?g* : $a' \rightarrow a$» $\wedge$ *D.cones-map ?g* $\chi = ?\chi'$
using χ*.induced-arrow-def* χ*.is-universal cone-*χ'
theI' [*of* λf. «*f* : $a' \rightarrow a$» $\wedge$ *D.cones-map f* $\chi = ?\chi'$]
by *presburger*
hence *2*: $x' = ?x \cdot_{[J,C]}$ Δ*.map ?g*
proof −
have x' = *J-C.MkArr* A'*.map ?D* $?\chi'$
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* : $a' \rightarrow a$» $\wedge$ $x' = ?x \cdot_{[J,C]}$ 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* : $a' \rightarrow a$» $\wedge$ *D.cones-map g* $\chi = ?\chi'$
proof
show *G*: «*g* : $a' \rightarrow a$» **using** *X x.is-coext-def* **by** *blast*
show *D.cones-map g* $\chi = ?\chi'$

proof −
have $?\chi' = J\text{-}C.Map\ (?x \cdot_{[J,C]} \Delta.map\ g)$
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 ⟨$\chi.\Phi.Ya.Cop\text{-}S.Map\ d$⟩
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 $J\text{-}C.Map\ (?x \cdot_{[J,C]} \Delta.map\ g) = vertical\text{-}composite.map\ J\ C\ map\text{-}g.map$
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*.is-natural-2* 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* $\lambda g.\ \ll g : a' \to a\gg \wedge D.cones\text{-}map\ g\ \chi = ?\chi'\ g$]
by *presburger*
qed
qed
qed
qed
thus $\exists\, a\ x.$ *terminal-arrow-from-functor C J-C.comp* Δ*.map a d x* **by** *auto*
qed
qed
qed

18.5 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 $?G\chi = G\ o\ \chi$
interpret $G\chi$: *cone J D GE.map* ‹*G a*› $?G\chi$
using χ*.cone-axioms preserves-cones* **by** *blast*

Claim that $G\chi$ is a limit cone for diagram GE.

interpret $G\chi$: *limit-cone J D GE.map* ‹*G a*› $?G\chi$
proof

Let κ be an arbitrary cone over GE.

fix $b\ \kappa$
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*

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 $?\chi' = \lambda j.$ *if J.arr j then Adj.*ε *(C.cod (E j))* $\cdot_C$ *F* $(\kappa\ j)$ *else C.null*
have *cone-*χ': *E.cone (F b)* $?\chi'$
proof
show $\bigwedge j.\ \neg J.arr\ j \Longrightarrow ?\chi'\ j = C.null$ **by** *simp*
fix *j*
assume *j*: *J.arr j*
show *C.dom* $(?\chi'\ j)$ = *Fb.map (J.dom j)* **using** *j* ψ*-in-hom* **by** *simp*
show *C.cod* $(?\chi'\ j)$ = *E (J.cod j)* **using** *j* ψ*-in-hom* **by** *simp*
show *E j* $\cdot_C$ $?\chi'$ *(J.dom j)* = $?\chi'\ j$

```
    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.is-natural-1 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.is-natural-1 κ.is-natural-2 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.is-extensional 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 E.cone (C.cod f) χ
            using f χ.cone-axioms by blast
          hence χ j ·C f = E.cones-map f χ j
            using χ.is-extensional by simp
          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.is-extensional 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*: $E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ =\ ?\chi'$
proof
fix j
have $\neg J.arr\ j \Longrightarrow E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ j\ =\ ?\chi'\ j$
using *1* χ*.cone-axioms* **by** *auto*
moreover have $J.arr\ j \Longrightarrow E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ j\ =\ ?\chi'\ j$
proof −
fix j
assume j: $J.arr\ j$
have $E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ j = \chi\ j\ \cdot_C\ \psi\ \ a\ \ g'$
using *1* χ*.cone-axioms* χ*.is-extensional* **by** *auto*
also have ... $= (\chi\ j\ \cdot_C\ Adj.\varepsilon\ \ a)\ \cdot_C\ F\ g'$
using $j\ a\ g'$ *Adj.*ψ*-in-terms-of-*ε *C.comp-assoc Adj.*ε*-def* **by** *auto*
also have ... $= (Adj.\varepsilon\ \ (C.cod\ \ (E\ j))\ \cdot_C\ F\ \ (G\ \ (\chi\ \ j)))\ \cdot_C\ F\ g'$
using $j\ a\ g'$ *Adj.*ε*.naturality* [*of* $\chi\ j$] **by** *simp*
also have ... $= Adj.\varepsilon\ \ (C.cod\ \ (E\ j))\ \cdot_C\ F\ \ (\kappa\ j)$
using $j\ a\ g'$ $G\chi$*.cone-axioms C.comp-assoc* **by** *auto*
finally show $E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ j\ =\ ?\chi'\ j$ **by** (*simp add*: j)
qed
ultimately show $E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ j\ =\ ?\chi'\ j$ **by** *auto*
qed
have $\psi\ \ a\ \ g' = f$
proof −
have $\exists!f.\ \ll f\ :\ F\ b\ \rightarrow_C\ a\gg\ \wedge\ E.cones\text{-}map\ f\ \ \chi\ =\ ?\chi'$
using *cone-*χ' χ*.is-universal* **by** *simp*
moreover have $\ll\psi\ \ a\ \ g'\ :\ F\ b\ \rightarrow_C\ a\gg\ \wedge\ E.cones\text{-}map\ (\psi\ \ a\ \ g')\ \chi\ =\ ?\chi'$
using *1 2* **by** *simp*
ultimately show *?thesis*
using *ex1E* [*of* $\lambda f.\ \ll f\ :\ F\ b\ \rightarrow_C\ a\gg\ \wedge\ E.cones\text{-}map\ f\ \ \chi\ =\ ?\chi'\ \psi\ \ a\ \ g' = f$]
using *1 2 Adj.*ε*.is-extensional C.comp-null*(*2*) *C.ex-un-null* χ*.cone-axioms* f
mem-Collect-eq restrict-apply
by *blast*
qed
hence $\varphi\ \ b\ \ (\psi\ \ a\ \ g') = \varphi\ \ b\ f$ **by** *auto*
hence $g' = \varphi\ \ b\ f$ **using** χ*.ide-apex* g' **by** (*simp add*: φ*-*ψ)
moreover have $?g\ =\ \varphi\ \ b\ f$ **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 $\exists!g.\ \ll g\ :\ b\ \rightarrow_D\ G\ a\gg\ \wedge\ GE.cones\text{-}map\ g\ ?G\chi = \kappa$ **by** *blast*
qed
have $GE.limit\text{-}cone\ (G\ a)\ ?G\chi$ **..**
thus *?thesis* **by** *auto*
qed

end

18.6 Special Kinds of Limits

18.6.1 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** $\cdot_J$ *55*)
and C :: *'c comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow {'c}$ +
assumes *is-empty*: $\neg J.arr\ j$
begin

lemma *has-as-limit-iff-terminal*:
shows *has-as-limit a* $\longleftrightarrow$ *C.terminal a*
proof
  assume *a*: *has-as-limit a*
  show *C.terminal a*
  proof
    have $\exists\chi$. *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*: $\chi = (\lambda j.\ C.null)$ **using** *is-empty* χ*.is-extensional* **by** *auto*
    show $\bigwedge a'.\ C.ide\ a' \Longrightarrow \exists!f.\ \ll f : a' \rightarrow a\gg$
    proof −
      fix a'
      assume a': $C.ide\ a'$
      interpret A': *constant-functor J C* a'
        apply *unfold-locales* **using** a' **by** *auto*
      let $?\chi' = \lambda j.\ C.null$
      have *cone-*χ': *cone* a' $?\chi'$
        using a' *is-empty* **apply** *unfold-locales* **by** *auto*
      hence $\exists!f.\ \ll f : a' \rightarrow a\gg \wedge$ *cones-map f* $\chi = ?\chi'$
        using χ*.is-universal* **by** *force*
      moreover have $\bigwedge f.\ \ll f : a' \rightarrow a\gg \Longrightarrow$ *cones-map f* $\chi = ?\chi'$
        using *1 cone-*χ **by** *auto*
      ultimately show $\exists!f.\ \ll f : a' \rightarrow a\gg$ **by** *blast*
    qed
  qed
  next
  assume *a*: *C.terminal a*
  show *has-as-limit a*
  proof −
    let $?\chi = \lambda j.\ C.null$
    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'.is-extensional 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
```

18.6.2 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*
apply *unfold-locales* **using** *assms*(*1*) **by** *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* $\pi = \pi$
proof –
interpret π: *cone J C D a* π
using *assms* **by** *auto*
show *mkCone* $\pi = \pi$ **using** π.*is-extensional* **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*: $i \in I \Longrightarrow$ *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** $\cdot_J$ *55*)
and C :: $'c$ *comp* (**infixr** $\cdot$ *55*)
and D :: $'j \Rightarrow 'c$
and a :: $'c$
and π :: $'j \Rightarrow 'c$
begin

lemma *is-cone*:
shows $D.cone\ a\ \pi$ **..**

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** $\bigwedge j.\ J.arr\ j \Longrightarrow «F\ j: b \rightarrow D\ j»$
shows $\exists!f.\ «f : b \rightarrow a» \wedge (\forall j.\ J.arr\ j \longrightarrow \pi\ j \cdot f = F\ j)$
proof −
let $?\chi = D.mkCone\ F$
interpret B: *constant-functor J C b*
apply *unfold-locales* **using** *assms*(*1*) **by** *auto*
have *cone-χ*: $D.cone\ b\ ?\chi$
using *assms D.is-discrete*
apply *unfold-locales*
apply *auto*
apply (*meson C.comp-ide-arr C.ide-in-hom C.seqI' D.preserves-ide*)
using *C.comp-arr-dom* **by** *blast*
interpret χ: *cone J C D b ?χ* **using** *cone-χ* **by** *auto*
have $\exists!f.\ «f : b \rightarrow a» \wedge D.cones\text{-}map\ f\ \pi = ?\chi$
using *cone-χ is-universal* **by** *force*
moreover have
$\bigwedge f.\ «f : b \rightarrow a» \Longrightarrow D.cones\text{-}map\ f\ \pi = ?\chi \longleftrightarrow (\forall j.\ J.arr\ j \longrightarrow \pi\ j \cdot f = F\ j)$
proof −
fix f
assume f: $«f : b \rightarrow a»$
show $D.cones\text{-}map\ f\ \pi = ?\chi \longleftrightarrow (\forall j.\ J.arr\ j \longrightarrow \pi\ j \cdot f = F\ j)$
proof
assume *1*: $D.cones\text{-}map\ f\ \pi = ?\chi$
show $\forall j.\ J.arr\ j \longrightarrow \pi\ j \cdot f = F\ j$
proof −
have $\bigwedge j.\ J.arr\ j \Longrightarrow \pi\ j \cdot f = F\ j$
proof −
fix j
assume j: $J.arr\ j$
have $\pi\ j \cdot f = D.cones\text{-}map\ f\ \pi\ j$
using *j f cone-axioms* **by** *force*
also have ... $= F\ j$ **using** *j 1* **by** *simp*
finally show $\pi\ j \cdot f = F\ j$ **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.is-extensional 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
    apply unfold-locales using assms(1) by 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 is-extensional 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
proof -
  interpret L: limit-cone J C D a π
    using assms by auto
  show product-cone J C D a π ..
qed

end
```

context *category*
begin

definition *has-as-product*
where *has-as-product J D a* ≡ (∃ π. *product-cone J C D a* π)

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 $I \neq \mathit{UNIV}$. 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*) ≡
  I ≠ *UNIV* ∧
  (∀ *J D*. *discrete-diagram J C D* ∧ *Collect* (*partial-magma.arr J*) = *I*
    ⟶ (∃ *a*. *has-as-product J D a*))

lemma *ex-productE*:
assumes ∃ *a*. *has-as-product J D a*
obtains *a* π **where** *product-cone J C D a* π
  using *assms has-as-product-def someI-ex* [*of* λ*a*. *has-as-product J D a*] **by** *metis*

lemma *has-products-if-has-limits*:
assumes *has-limits* (*undefined* :: *'j*) **and** *I* ≠ (*UNIV* :: *'j set*)
shows *has-products I*
proof −
  have ⋀*J D*. ⟦ *discrete-diagram J C D*; *Collect* (*partial-magma.arr J*) = *I* ⟧
      ⟹ (∃ *a*. *has-as-product J D a*)
  proof −
    fix *J* :: *'j comp* **and** *D*
    assume *D*: *discrete-diagram J C D*
    interpret *J*: *category J*
      using *D discrete-diagram.axioms* **by** *auto*
    interpret *D*: *discrete-diagram J C D*
      using *D* **by** *auto*
    assume *J*: *Collect J.arr* = *I*
    obtain *a* π **where** π: *D.limit-cone a* π
      using *assms*(*1*) *J has-limits-def has-limits-of-shape-def* [*of J*]
        D.diagram-axioms J.category-axioms
      by *metis*
    have *product-cone J C D a* π
      using π *D.product-coneI* **by** *auto*
    hence *has-as-product J D a*
      using *has-as-product-def* **by** *blast*
    thus ∃ *a*. *has-as-product J D a*
      by *auto*
  qed

thus *?thesis*
unfolding *has-products-def* **using** *assms*(*2*) **by** *auto*
qed

end

18.6.3 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** $\cdot_J$ *55*)

definition *map*
where *map* ≡ (λ*j*. *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*: $\bigwedge j.\ J.arr\ j \Longrightarrow C.cod\ (map\ j) = map\ (J.cod\ j)$
proof −
fix *j*
assume *j*: *J.arr j*
show $C.cod\ (map\ j) = map\ (J.cod\ j)$
proof −
have $j = J.Zero \lor j = J.One \Longrightarrow$ *?thesis* **using** *is-parallel map-def* **by** *auto*
moreover have $j = J.j0 \lor j = J.j1 \Longrightarrow$ *?thesis*
using *is-parallel map-def J.Zero-not-eq-j0 J.One-not-eq-j0 J.Zero-not-eq-One J.Zero-not-eq-j1 J.One-not-eq-j1 J.Zero-not-eq-One J.cod-simp*
by *presburger*
ultimately show *?thesis* **using** *j J.arr-char* **by** *fast*
qed
qed
next
fix *j j'*
assume *jj'*: *J.seq j' j*
show $map\ (j' \cdot_J j) = map\ j' \cdot map\ j$
proof −
have *1*: $(j = J.Zero \land j' \neq J.One) \lor (j \neq J.Zero \land j' = J.One)$
using *jj' J.seq-char* **by** *blast*
moreover have $j = J.Zero \land j' \neq J.One \Longrightarrow$ *?thesis*
using *jj' map-def is-parallel J.arr-char J.cod-simp J.dom-simp J.seq-char*
by (*metis* (*no-types*, *lifting*) *C.arr-dom-iff-arr C.comp-arr-dom C.dom-dom J.comp-arr-dom*)
moreover have $j \neq J.Zero \land j' = J.One \Longrightarrow$ *?thesis*
using *jj' J.ide-char map-def J.Zero-not-eq-One is-parallel*
by (*metis* (*no-types*, *lifting*) *C.arr-cod-iff-arr C.comp-arr-dom C.comp-cod-arr C.comp-ide-arr C.ext C.ide-cod J.comp-simp*(*2*))
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*⟩
    apply *unfold-locales* **using** *assms* **by** *auto*
  show *cone* (*C.dom e*) (*mkCone e*)
    using *assms mkCone-def* **apply** *unfold-locales*
        apply *auto*[*2*]
   using *C.dom-comp C.seqE C.cod-comp J.Zero-not-eq-One J.arr-char′ J.cod-char map-def*
      apply (*metis* (*no-types*, *lifting*) *C.not-arr-null parallel-pair.cod-simp*(*1*) *preserves-arr*)
  proof −
    fix *j*
    assume *j*: *J.arr j*
    show *map j* · *mkCone e* (*J.dom j*) = *mkCone e j*
    proof −
      have *1*: ∀ *a*. *if a* = *J.Zero then map a* = *C.dom f0*
                    else if a = *J.One then map a* = *C.cod f0*
                    else if a = *J.j0 then map a* = *f0*
                    else if a = *J.j1 then map a* = *f1*
                    else map a = *C.null*
        using *map-def* **by** *auto*
      hence *2*: *map j* = *f1* ∨ *j* = *J.One* ∨ *j* = *J.Zero* ∨ *j* = *J.j0*
        using *j parallel-pair.arr-char* **by** *meson*
      have *j* = *J.Zero* ∨ *map j* · *mkCone e* (*J.dom j*) = *mkCone e j*
        using *assms j 1 2 mkCone-def C.cod-comp*
       by (*metis* (*no-types*, *lifting*) *C.comp-cod-arr J.arr-char J.dom-simp*(*2*−*4*) *is-parallel*)
      thus *?thesis*
        using *assms 1 j*
        by (*metis* (*no-types*, *lifting*) *C.comp-cod-arr C.seqE mkCone-def J.dom-simp*(*1*))
    qed
    next
    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 *J.cod j* = *J.Zero* ⟹ *mkCone e* (*J.cod j*) · *E.map j* = *mkCone e j*
        unfolding *mkCone-def*
        using *assms j J.arr-char J.cod-char C.comp-arr-dom mkCone-def J.Zero-not-eq-One*
        by (*metis* (*no-types*, *lifting*) *C.seqE E.map-simp*)
      moreover have *J.cod j* ≠ *J.Zero* ⟹ *mkCone e* (*J.cod j*) · *E.map j* = *mkCone e j*
        unfolding *mkCone-def*
        using *assms j C.comp-arr-dom* **by** *auto*
      ultimately show *mkCone e* (*J.cod j*) · *E.map j* = *mkCone e j* **by** *blast*
    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
      using assms J.arr-char J.dom-char J.cod-char
            J.One-not-eq-j0 J.One-not-eq-j1 J.Zero-not-eq-j0 J.Zero-not-eq-j1 J.j0-not-eq-j1
      by (metis (no-types, lifting) Limit.cone-def χ.is-natural-1 χ.naturality
          χ.preserves-reflects-arr constant-functor.map-simp map-simp(3) map-simp(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.is-extensional 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.is-natural-1 χ.is-natural-2 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.is-extensional 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 1: 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
  hence 2: C.seq f1 e
    using D.is-parallel by fastforce
  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 $\neg J.arr\ j \Longrightarrow D.cones\text{-}map\ h\ (D.mkCone\ e)\ j = D.mkCone\ e'\ j$
    using *h cone-axioms D.mkCone-def* **by** *auto*
  moreover have $j = J.Zero \Longrightarrow D.cones\text{-}map\ h\ (D.mkCone\ e)\ j = D.mkCone\ e'\ j$
    using *h e′ cone-χ D.mkCone-def J.arr-char* [*of J.Zero*] **by** *force*
  moreover have
      $J.arr\ j \wedge j \neq J.Zero \Longrightarrow D.cones\text{-}map\ h\ (D.mkCone\ e)\ j = D.mkCone\ e'\ j$
  proof −
    assume *j*: $J.arr\ j \wedge j \neq J.Zero$
    have $D.cones\text{-}map\ h\ (D.mkCone\ e)\ j = C\ (D.mkCone\ e\ j)\ h$
      using *j h equalizes D.mkCone-def D.cone-mkCone J.arr-char*
          J.Zero-not-eq-One J.Zero-not-eq-j0 J.Zero-not-eq-j1
      by *auto*
    also have ... $= (f0 \cdot e) \cdot h$
      using *j D.mkCone-def J.arr-char J.Zero-not-eq-One J.Zero-not-eq-j0*
          J.Zero-not-eq-j1
      by *auto*
    also have ... $= f0 \cdot e \cdot h$
      using *h equalizes C.comp-assoc* **by** *blast*
    also have ... $= D.mkCone\ e'\ j$
      using *j e′ h equalizes D.mkCone-def J.arr-char* [*of J.One*] *J.Zero-not-eq-One*
      by *auto*
    finally show *?thesis* **by** *auto*
  qed
  ultimately show $D.cones\text{-}map\ h\ (D.mkCone\ e)\ j = D.mkCone\ e'\ j$ **by** *blast*
qed
qed
qed
thus *?thesis* **using** *1* **by** *blast*
qed

lemma *induced-arrowI′*:
assumes *D.is-equalized-by e′*
shows $\ll induced\text{-}arrow\ (C.dom\ e')\ (D.mkCone\ e') : C.dom\ e' \rightarrow C.dom\ e\gg$
and $e \cdot induced\text{-}arrow\ (C.dom\ e')\ (D.mkCone\ e') = e'$
proof −
  interpret *A′*: *constant-functor J.comp C* $\langle C.dom\ e'\rangle$
    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 $e \cdot induced\text{-}arrow\ (C.dom\ e')\ (D.mkCone\ e') =$
      $D.cones\text{-}map\ (induced\text{-}arrow\ (C.dom\ e')\ (D.mkCone\ e'))\ (D.mkCone\ e)\ J.Zero$
    using *cone induced-arrowI*(*1*) *D.mkCone-def J.arr-char cone-χ* **by** *force*
  also have ... $= e'$
  proof −
    have
      $D.cones\text{-}map\ (induced\text{-}arrow\ (C.dom\ e')\ (D.mkCone\ e'))\ (D.mkCone\ e) = D.mkCone$
e'

using *cone induced-arrowI* **by** *blast*
thus *?thesis*
using *J.arr-char D.mkCone-def* **by** *simp*
qed
finally have *1*: $e \cdot$ *induced-arrow* (*C.dom* e') (*D.mkCone* e') = e'
by *auto*
show «*induced-arrow* (*C.dom* e') (*D.mkCone* e') : *C.dom* $e' \rightarrow$ *C.dom* e»
using *1 cone induced-arrowI* **by** *simp*
show $e \cdot$ *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*))

end

18.7 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: $J \rightarrow C$ is the equalizer e of f, g: $\Pi_i\ F_i \rightarrow \Pi_u\ F_{cod\ u}$ ($u \in arr\ J$, $i \in J$) 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** $\cdot_J$ *55*)
assumes *category J* **and** *has-products* (*Collect* (*partial-magma.ide J*))
and *has-products* (*Collect* (*partial-magma.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 $\bigwedge D.$ *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*

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* Δo*.map p*
using *assms(2)* Δo*.diagram-axioms has-products-def Obj.arr-char*
by (*metis* (*no-types*, *lifting*) *Collect-cong* Δo*.discrete-diagram-axioms mem-Collect-eq*)
from *this* **obtain** Πo πo **where** πo: *product-cone Obj.comp C* Δo*.map* Πo πo
using *ex-productE* [*of Obj.comp* Δo*.map*] **by** *auto*
interpret πo: *product-cone Obj.comp C* Δo*.map* Πo πo **using** πo **by** *auto*
have πo*-in-hom*: $\bigwedge j.$ *Obj.arr j* $\Longrightarrow$ «$\pi o\ j$: $\Pi o \rightarrow D\ j$»
using πo*.preserves-dom* πo*.preserves-cod* Δo*.map-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* Δa*.map p*
using *assms(3) has-products-def* [*of Collect J.arr*] Δa*.discrete-diagram-axioms*
by *blast*
from *this* **obtain** Πa πa **where** πa: *product-cone Arr.comp C* Δa*.map* Πa πa
using *ex-productE* [*of Arr.comp* Δa*.map*] **by** *auto*
interpret πa: *product-cone Arr.comp C* Δa*.map* Πa πa **using** πa **by** *auto*
have πa*-in-hom*: $\bigwedge j.$ *Arr.arr j* $\Longrightarrow$ «$\pi a\ j$: $\Pi a \rightarrow D$ (*J.cod j*)»
using πa*.preserves-cod* πa*.preserves-dom* Δa*.map-def* **by** *auto*

Next, construct a parallel pair of arrows f, g: $\Pi o \rightarrow \Pi a$ that expresses the commutativity constraints imposed by the diagram.

interpret Πo: *constant-functor Arr.comp C* Πo
using πo*.ide-apex* **by** (*unfold-locales*, *auto*)
let $?\chi = \lambda j.$ *if Arr.arr j then* πo (*J.cod j*) *else null*
interpret χ: *cone Arr.comp C* Δa*.map* Πo $?\chi$
using πo*.ide-apex* πo*-in-hom* Δa*.map-def* Δo*.map-def* Δo*.is-discrete* πo*.is-natural-2*
comp-cod-arr
by (*unfold-locales*, *auto*)

let $?f = \pi a$*.induced-arrow* Πo $?\chi$
have *f-in-hom*: «$?f$: $\Pi o \rightarrow \Pi a$»
using χ*.cone-axioms* πa*.induced-arrowI* **by** *blast*
have *f-map*: Δa*.cones-map* $?f$ $\pi a = ?\chi$
using χ*.cone-axioms* πa*.induced-arrowI* **by** *blast*
have *ff*: $\bigwedge j.$ *J.arr j* $\Longrightarrow$ $\pi a\ j \cdot ?f = \pi o$ (*J.cod j*)

proof −
  fix j
  assume j: $J.arr\ j$
  have $\pi a\ j \cdot ?f = \Delta a.cones\text{-}map\ ?f\ \pi a\ j$
    using *f-in-hom* πa*.is-cone* πa*.is-extensional* **by** *auto*
  also have ... $= \pi o\ (J.cod\ j)$
    using j *f-map* **by** *fastforce*
  finally show $\pi a\ j \cdot ?f = \pi o\ (J.cod\ j)$ **by** *auto*
qed

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 πo*.ide-apex* πo*-in-hom* Δo*.map-def* Δa*.map-def* *comp-arr-dom comp-cod-arr*
  by (*unfold-locales*, *auto*)
let $?g = \pi a.induced\text{-}arrow\ \Pi o\ ?\chi'$
have *g-in-hom*: $\ll ?g : \Pi o \to \Pi a \gg$
  using χ'*.cone-axioms* πa*.induced-arrowI* **by** *blast*
have *g-map*: $\Delta a.cones\text{-}map\ ?g\ \pi a = ?\chi'$
  using χ'*.cone-axioms* πa*.induced-arrowI* **by** *blast*
have *gg*: $\bigwedge j.\ J.arr\ j \Longrightarrow \pi a\ j \cdot ?g = D\ j \cdot \pi o\ (J.dom\ j)$
proof −
  fix j
  assume j: $J.arr\ j$
  have $\pi a\ j \cdot ?g = \Delta a.cones\text{-}map\ ?g\ \pi a\ j$
    using *g-in-hom* πa*.is-cone* πa*.is-extensional* **by** *force*
  also have ... $= D\ j \cdot \pi o\ (J.dom\ j)$
    using j *g-map* **by** *fastforce*
  finally show $\pi a\ j \cdot ?g = D\ j \cdot \pi o\ (J.dom\ j)$ **by** *auto*
qed

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: $\bigwedge h.\ \ll h : dom\ h \to \Pi o \gg \Longrightarrow$
    $?f \cdot h = ?g \cdot h \longleftrightarrow (\forall j.\ J.arr\ j \longrightarrow ?\chi\ j \cdot h = ?\chi'\ j \cdot h)$
proof
  fix h
  assume h: $\ll h : dom\ h \to \Pi o \gg$
  show $?f \cdot h = ?g \cdot h \Longrightarrow \forall j.\ J.arr\ j \longrightarrow ?\chi\ j \cdot h = ?\chi'\ j \cdot h$
  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.cone-χ 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.cone-χ 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
      proof -
        have πa j · ?f = Δa.cones-map ?f πa j
          using j f-in-hom πa.cone-axioms Δa.map-def πa.cone-χ by auto
        thus ?thesis using f-map by fastforce
      qed
      also have ... = ?χ' j · h
        using 1 j by auto
      also have ... = (πa j · ?g) · h
      proof -
        have πa j · ?g = Δa.cones-map ?g πa j
          using j g-in-hom πa.cone-axioms Δa.map-def πa.cone-χ by auto
        thus ?thesis using g-map by simp
      qed
      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 $\bigwedge j.\ Arr.arr\ j \Longrightarrow \ll\pi a\ j \cdot ?f \cdot h : dom\ h \rightarrow \Delta a.map\ j\gg$
  using *f-in-hom h πa-in-hom* **by** (*elim in-homE*, *auto*)
hence *3*: $\exists!k.\ \ll k : dom\ h \rightarrow \Pi a\gg \wedge (\forall j.\ Arr.arr\ j \longrightarrow \pi a\ j \cdot k = \pi a\ j \cdot ?f \cdot h)$
  using $h\ \pi a\ \pi a.is\text{-}universal'$ [*of dom h* $\lambda j.\ \pi a\ j \cdot ?f \cdot h$] $\Delta a.map\text{-}def$
    ide-dom [*of h*]
  by *blast*
have *4*: $\bigwedge P\ x\ x'.\ \exists!k.\ P\ k\ x \Longrightarrow P\ x\ x \Longrightarrow P\ x'\ x \Longrightarrow x' = x$ **by** *auto*
let $?P = \lambda\ k\ x.\ \ll k : dom\ h \rightarrow \Pi a\gg \wedge$
    $(\forall j.\ j \in Collect\ J.arr \longrightarrow \pi a\ j \cdot k = \pi a\ j \cdot x)$
have $?P\ (?g \cdot h)\ (?g \cdot h)$
  using *g-in-hom h* **by** *force*
moreover have $?P\ (?f \cdot h)\ (?g \cdot h)$
  using *2 f-in-hom g-in-hom h* **by** *force*
ultimately show *?thesis*
  using *3 4* [*of ?P ?f* $\cdot$ *h ?g* $\cdot$ *h*] **by** *auto*
qed
qed
qed
have E': $\bigwedge e.\ \ll e : dom\ e \rightarrow \Pi o\gg \Longrightarrow$
  $?f \cdot e = ?g \cdot e \longleftrightarrow$
  $(\forall j.\ J.arr\ j \longrightarrow$
    $(D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot e) \cdot dom\ e = D\ j \cdot \pi o\ (J.dom\ j) \cdot e)$
proof –
have *1*: $\bigwedge e\ j.\ \ll e : dom\ e \rightarrow \Pi o\gg \Longrightarrow J.arr\ j \Longrightarrow$
  $?\chi\ j \cdot e = (D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot e) \cdot dom\ e$
proof –
fix *e j*
assume *e*: $\ll e : dom\ e \rightarrow \Pi o\gg$
assume *j*: *J.arr j*
have $\ll\pi o\ (J.cod\ j) \cdot e : dom\ e \rightarrow D\ (J.cod\ j)\gg$
  using *e j πo-in-hom* **by** *auto*
thus $?\chi\ j \cdot e = (D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot e) \cdot dom\ e$
  using *j comp-arr-dom comp-cod-arr* **by** (*elim in-homE*, *auto*)
qed
have *2*: $\bigwedge e\ j.\ \ll e : dom\ e \rightarrow \Pi o\gg \Longrightarrow J.arr\ j \Longrightarrow ?\chi'\ j \cdot e = D\ j \cdot \pi o\ (J.dom\ j) \cdot e$
proof –
fix *e j*
assume *e*: $\ll e : dom\ e \rightarrow \Pi o\gg$
assume *j*: *J.arr j*
show $?\chi'\ j \cdot e = D\ j \cdot \pi o\ (J.dom\ j) \cdot e$
  using *j comp-assoc* **by** *fastforce*
qed
show $\bigwedge e.\ \ll e : dom\ e \rightarrow \Pi o\gg \Longrightarrow$
  $?f \cdot e = ?g \cdot e \longleftrightarrow$
  $(\forall j.\ J.arr\ j \longrightarrow$
    $(D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot e) \cdot dom\ e = D\ j \cdot \pi o\ (J.dom\ j) \cdot e)$
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*: $\ll e : dom\ e \to \Pi o \gg$
  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 $?\mu = \lambda j.$ *if J.arr j then* $D\ j \cdot \pi o\ (J.dom\ j) \cdot e$ *else null*
have μ: $\bigwedge j.\ J.arr\ j \Longrightarrow \ll ?\mu\ j : dom\ e \to D\ (J.cod\ j) \gg$
proof –
  fix *j*
  assume *j*: *J.arr j*
  show $\ll ?\mu\ j : dom\ e \to D\ (J.cod\ j) \gg$
    using *j e-in-hom* πo*-in-hom* [*of J.dom j*] **by** *auto*
qed
interpret μ: *cone J C D* ⟨*dom e*⟩ $?\mu$
  apply *unfold-locales*
    apply *simp*
proof –
  fix *j*
  assume *j*: *J.arr j*
  show $dom\ (?\mu\ j) = domE.map\ (J.dom\ j)$ **using** *j* μ *domE.map-simp* **by** *force*
  show $cod\ (?\mu\ j) = D\ (J.cod\ j)$ **using** *j* μ *D.preserves-cod* **by** *blast*
  show $D\ j \cdot ?\mu\ (J.dom\ j) = ?\mu\ j$
    using *j* μ [*of J.dom j*] *comp-cod-arr* **apply** *simp*
    by (*elim in-homE*, *auto*)
  show $?\mu\ (J.cod\ j) \cdot domE.map\ j = ?\mu\ j$
    using *j e-map E'* **by** (*simp add*: *e-in-hom*)
qed

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*: $\bigwedge a\ \tau.$ *cone J C D a* $\tau \Longrightarrow$
    cone Obj.comp C $\Delta o.map\ a\ (\Delta o.mkCone\ \tau) \wedge$
    $?f \cdot \pi o.induced\text{-}arrow\ a\ (\Delta o.mkCone\ \tau)$
      $= ?g \cdot \pi o.induced\text{-}arrow\ a\ (\Delta o.mkCone\ \tau)$
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* $\Delta o.map\ a$ ⟨$\Delta o.mkCone\ \tau$⟩
    using *A.value-is-ide* Δo*.map-def comp-cod-arr comp-arr-dom*
    by (*unfold-locales*, *auto*)
  let $?e = \pi o.induced\text{-}arrow\ a\ (\Delta o.mkCone\ \tau)$

have *mkCone-τ*: $\Delta o.mkCone\ \tau \in \Delta o.cones\ a$
proof −
have $\bigwedge j.\ Obj.arr\ j \Longrightarrow «\tau\ j : a \rightarrow \Delta o.map\ j»$
using *Obj.arr-char τ.A.map-def Δo.map-def* **by** *force*
thus *?thesis*
using *τ.ide-apex Δo.cone-mkCone* **by** *simp*
qed
have *e*: $«?e : a \rightarrow \Pi o»$
using *mkCone-τ πo.induced-arrowI* **by** *simp*
have *ee*: $\bigwedge j.\ J.ide\ j \Longrightarrow \pi o\ j \cdot ?e = \tau\ j$
proof −
fix *j*
assume *j*: $J.ide\ j$
have $\pi o\ j \cdot ?e = \Delta o.cones\text{-}map\ ?e\ \pi o\ j$
using *j e πo.cone-axioms* **by** *force*
also have ... $= \Delta o.mkCone\ \tau\ j$
using *j mkCone-τ πo.induced-arrowI* [*of* $\Delta o.mkCone\ \tau\ a$] **by** *fastforce*
also have ... $= \tau\ j$
using *j* **by** *simp*
finally show $\pi o\ j \cdot ?e = \tau\ j$ **by** *auto*
qed
have $\bigwedge j.\ J.arr\ j \Longrightarrow$
$(D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot ?e) \cdot dom\ ?e = D\ j \cdot \pi o\ (J.dom\ j) \cdot ?e$
proof −
fix *j*
assume *j*: $J.arr\ j$
have *1*: $«\pi o\ (J.cod\ j) : \Pi o \rightarrow D\ (J.cod\ j)»$ **using** *j πo-in-hom* **by** *simp*
have *2*: $(D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot ?e) \cdot dom\ ?e$
$= D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot ?e$
proof −
have $seq\ (D\ (J.cod\ j))\ (\pi o\ (J.cod\ j))$
using *j 1* **by** *auto*
moreover have $seq\ (\pi o\ (J.cod\ j))\ ?e$
using *j e* **by** *fastforce*
ultimately show *?thesis* **using** *comp-arr-dom* **by** *auto*
qed
also have *3*: ... $= \pi o\ (J.cod\ j) \cdot ?e$
using *j e 1 comp-cod-arr* **by** (*elim in-homE*, *auto*)
also have ... $= D\ j \cdot \pi o\ (J.dom\ j) \cdot ?e$
using *j e ee 2 3 τ.naturality τ.A.map-simp τ.ide-apex comp-cod-arr* **by** *auto*
finally show $(D\ (J.cod\ j) \cdot \pi o\ (J.cod\ j) \cdot ?e) \cdot dom\ ?e = D\ j \cdot \pi o\ (J.dom\ j) \cdot ?e$
by *auto*
qed
hence $C\ ?f\ ?e = C\ ?g\ ?e$
using *E′ πo.induced-arrowI τo.cone-axioms mem-Collect-eq* **by** *blast*
thus $cone\ Obj.comp\ C\ \Delta o.map\ a\ (\Delta o.mkCone\ \tau) \wedge C\ ?f\ ?e = C\ ?g\ ?e$
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
    apply unfold-locales using τ.ide-apex by 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.is-extensional 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 (elim in-homE, auto)
        also have ... = Δo.cones-map ?e′ πo (J.dom j)
          using j e′-in-hom πo.cone-axioms by (elim in-homE, 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.is-natural-1 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′
      proof −
        have ?f · e · h′ = (?f · e) · h′
          using comp-assoc by auto
        also have ... = ?g · e · h′
          using EQU.equalizes comp-assoc by auto
        finally show ?thesis by auto
      qed
      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 $\Delta o.cones\text{-}map\ (e \cdot h')\ \pi o\ j = \pi o\ j \cdot e \cdot h'$
using *2 j* $\pi o.cone\text{-}axioms$ **by** *auto*
also have ... $= (\pi o\ j \cdot e) \cdot h'$
using *comp-assoc* **by** *auto*
also have ... $= \Delta o.mkCone\ ?\mu\ j \cdot h'$
using *j e-in-hom* πo*-in-hom comp-ide-arr* [*of D j* $\pi o\ j \cdot e$]
by *fastforce*
also have ... $= \Delta o.mkCone\ \tau\ j$
using *j* h' μ*.cone-axioms mem-Collect-eq* **by** *auto*
finally show $\Delta o.cones\text{-}map\ (e \cdot h')\ \pi o\ j = \Delta o.mkCone\ \tau\ j$ **by** *auto*
qed
ultimately show $\Delta o.cones\text{-}map\ (e \cdot h')\ \pi o\ j = \Delta o.mkCone\ \tau\ j$ **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
have $\ll e \cdot h' : a \to \Pi o\gg$ **using** *1* **by** *simp*
moreover have $e \cdot h' = ?e'$
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* $e \cdot h'$]
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 $\exists\, a\ \mu.$ *limit-cone J C D a* μ **by** *auto*
qed
thus $\forall D.$ *diagram J C D* $\longrightarrow$ ($\exists\, a\ \mu.$ *limit-cone J C D a* μ) **by** *blast*
qed

end

18.8 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* +
diagram J S D
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *S* :: *'s comp* (**infixr** $\cdot$ *55*)
and *D* :: *'j* $\Rightarrow$ *'s*
begin

notation *S.in-hom* ($\ll$- : - $\to$ -$\gg$)

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 $has\text{-}as\text{-}limit\ a \longleftrightarrow S.ide\ a \wedge (\exists \varphi.\ bij\text{-}betw\ \varphi\ (S.hom\ S.unity\ a)\ (cones\ S.unity))$
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: $has\text{-}as\text{-}limit\ a$
hence $S.ide\ a$
  using *limit-cone-def cone.ide-apex* **by** *metis*
from a **obtain** χ **where** χ: $limit\text{-}cone\ a\ \chi$ **by** *auto*
interpret χ: $limit\text{-}cone\ J\ S\ D\ a\ \chi$ **using** χ **by** *auto*
have $bij\text{-}betw\ (\lambda f.\ cones\text{-}map\ f\ \chi)\ (S.hom\ S.unity\ a)\ (cones\ S.unity)$
  using $\chi.bij\text{-}betw\text{-}hom\text{-}and\text{-}cones\ S.ide\text{-}unity$ **by** *simp*
thus $S.ide\ a \wedge (\exists \varphi.\ bij\text{-}betw\ \varphi\ (S.hom\ S.unity\ a)\ (cones\ S.unity))$
  using $\langle S.ide\ a\rangle$ **by** *blast*
next

Conversely, an arbitrary bijection φ between $S.hom\ S.unity\ a$ and cones unity extends pointwise to a natural bijection $\Phi\ a'$ 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: $S.ide\ a \wedge (\exists \varphi.\ bij\text{-}betw\ \varphi\ (S.hom\ S.unity\ a)\ (cones\ S.unity))$
hence *ide-a*: $S.ide\ a$ **by** *auto*
show *has-as-limit* a
proof −
  from a **obtain** φ **where** φ: $bij\text{-}betw\ \varphi\ (S.hom\ S.unity\ a)\ (cones\ S.unity)$ **by** *blast*
  have X: $\bigwedge f\ j\ y.\ [\![\ll f : S.dom\ f \to a\gg;\ J.arr\ j;\ \ll y : S.unity \to S.dom\ f\gg]\!]$
      $\Longrightarrow \ll \varphi\ (f \cdot y)\ j : S.unity \to S.cod\ (D\ j)\gg$
  proof −
    fix $f\ j\ y$
    assume f: $\ll f : S.dom\ f \to a\gg$ **and** j: $J.arr\ j$ **and** y: $\ll y : S.unity \to S.dom\ f\gg$
    interpret χ: $cone\ J\ S\ D\ S.unity\ \langle\varphi\ (S\ f\ y)\rangle$
      using $f\ y\ \varphi$ *bij-betw-imp-funcset funcset-mem* **by** *blast*
    show $\ll \varphi\ (f \cdot y)\ j : S.unity \to S.cod\ (D\ j)\gg$ **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* $= \lambda f\, j\ \chi j.\ \ll\chi j : S.dom\ f \to S.cod\ (D\ j)\gg \wedge$
$(\forall\, y.\ \ll y : S.unity \to S.dom\ f\gg \longrightarrow \chi j \cdot y = \varphi\ (f \cdot y)\ j)$
let $?\chi = \lambda f\, j.$ *if* $J.arr\ j$ *then* $(THE\ \chi j.\ ?P\ f\ j\ \chi j)$ *else* $S.null$
have χ: $\bigwedge f\, j.\ [\![\ \ll f : S.dom\ f \to a\gg;\ J.arr\ j\ ]\!] \Longrightarrow\ ?P\ f\ j\ (?\chi\ f\ j)$
proof −
fix $b\ f\ j$
assume f: $\ll f : S.dom\ f \to a\gg$ **and** j: $J.arr\ j$
interpret B: *constant-functor* $J\ S\ \langle S.dom\ f\rangle$
using f **by** (*unfold-locales*, *auto*)
have $(\lambda y.\ \varphi\ (f \cdot y)\ j) \in S.hom\ S.unity\ (S.dom\ f) \to S.hom\ S.unity\ (S.cod\ (D\ j))$
using $f\ j\ X$ *Pi-I'* **by** *simp*
hence $\exists!\chi j.\ ?P\ f\ j\ \chi j$
using $f\ j$ *S.fun-complete'* [*of* $S.dom\ f\ S.cod\ (D\ j)\ \lambda y.\ \varphi\ (f \cdot y)\ j$]
by (*elim S.in-homE*, *auto*)
thus $?P\ f\ j\ (?\chi\ f\ j)$ **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*: $\bigwedge f.\ \ll f : S.dom\ f \to a\gg \Longrightarrow$ *cone* $(S.dom\ f)\ (?\chi\ f)$
proof −
fix f
assume f: $\ll f : S.dom\ f \to a\gg$
interpret B: *constant-functor* $J\ S\ \langle S.dom\ f\rangle$
using f **by** (*unfold-locales*, *auto*)
show *cone* $(S.dom\ f)\ (?\chi\ f)$
proof
show $\bigwedge j.\ \neg J.arr\ j \Longrightarrow\ ?\chi\ f\ j = S.null$ **by** *simp*
fix j
assume j: $J.arr\ j$
have *0*: $\ll ?\chi\ f\ j : S.dom\ f \to S.cod\ (D\ j)\gg$ **using** $f\ j\ \chi$ **by** *simp*
show $S.dom\ (?\chi\ f\ j) = B.map\ (J.dom\ j)$ **using** $f\ j\ \chi$ **by** *auto*
show $S.cod\ (?\chi\ f\ j) = D\ (J.cod\ j)$ **using** $f\ j\ \chi$ **by** *auto*
have *par1*: $S.par\ (D\ j \cdot\ ?\chi\ f\ (J.dom\ j))\ (?\chi\ f\ j)$
using $f\ j$ *0* χ [*of* $f\ J.dom\ j$] **by** (*elim S.in-homE*, *auto*)
have *par2*: $S.par\ (?\chi\ f\ (J.cod\ j) \cdot B.map\ j)\ (?\chi\ f\ j)$
using $f\ j$ *0* χ [*of* $f\ J.cod\ j$] **by** (*elim S.in-homE*, *auto*)
have *nat*: $\bigwedge y.\ \ll y : S.unity \to S.dom\ f\gg \Longrightarrow$
$(D\ j \cdot\ ?\chi\ f\ (J.dom\ j)) \cdot y =\ ?\chi\ f\ j \cdot y\ \wedge$
$(?\chi\ f\ (J.cod\ j) \cdot B.map\ j) \cdot y =\ ?\chi\ f\ j \cdot y$
proof −
fix y
assume y: $\ll y : S.unity \to S.dom\ f\gg$
show $(D\ j \cdot\ ?\chi\ f\ (J.dom\ j)) \cdot y =\ ?\chi\ f\ j \cdot y\ \wedge$
$(?\chi\ f\ (J.cod\ j) \cdot B.map\ j) \cdot y =\ ?\chi\ f\ j \cdot y$
proof
have *1*: $\varphi\ (f \cdot y) \in$ *cones* $S.unity$
using $f\ y\ \varphi$ *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
        have (D j · ?χ f (J.dom j)) · y = D j · ?χ f (J.dom j) · y
          using S.comp-assoc by simp
        also have ... = D j · φ (f · y) (J.dom j)
          using f y χ χ.is-extensional by simp
        also have ... = φ (f · y) j using j by auto
        also have ... = ?χ f j · y
          using f j y χ by force
        finally show (D j · ?χ f (J.dom j)) · y = ?χ f j · y by auto
        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.is-extensional by simp
        also have ... = φ (f · y) j
          using j χ.is-natural-2
          by (metis J.arr-cod χ.A.map-simp J.cod-cod)
        also have ... = ?χ f j · y
          using f y χ χ.is-extensional 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
      using par1 nat 0
      apply (intro S.arr-eqI' [of D j · ?χ f (J.dom j) ?χ f j])
       apply force
      by auto
    show ?χ f (J.cod j) · B.map j = ?χ f j
      using par2 nat 0 f j χ
      apply (intro S.arr-eqI' [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'.is-extensional fχa.is-extensional 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' [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 $\ll y : S.unity \rightarrow S.dom\ (cones\text{-}map\ ?f\ (?\chi\ a)\ j)\gg$
hence y: $\ll y : S.unity \rightarrow a'\gg$
using *j f χa.preserves-dom* **by** *simp*
have *1*: $\ll ?\chi\ a\ j : a \rightarrow D\ (J.cod\ j)\gg$
using *j χa.preserves-hom* **by** *force*
have *2*: $\ll ?f \cdot y : S.unity \rightarrow a\gg$
using *f-in-hom y* **by** *blast*
have $cones\text{-}map\ ?f\ (?\chi\ a)\ j \cdot y = (?\chi\ a\ j \cdot ?f) \cdot y$
proof −
have $S.cod\ ?f = a$ **using** *f-in-hom* **by** *blast*
thus *?thesis* **using** *j χa.cone-axioms* **by** *simp*
qed
also have ... $= ?\chi\ a\ j \cdot ?f \cdot y$
using *1 j y f-in-hom S.comp-assoc S.seqI′* **by** *blast*
also have ... $= \varphi\ (a \cdot ?f \cdot y)\ j$
using *1 2 ide-a f j y χ* [*of a*] **by** (*simp add*: *S.ide-in-hom*)
also have ... $= \varphi\ (?f \cdot y)\ j$
using *a 2 y S.comp-cod-arr* **by** (*elim S.in-homE*, *auto*)
also have ... $= \varphi\ (?\psi\ (cones\text{-}map\ y\ \chi'))\ j$
using *j y f* **by** *simp*
also have ... $= cones\text{-}map\ y\ \chi'\ j$
proof −
have $cones\text{-}map\ y\ \chi' \in cones\ S.unity$
using *cone-χ′ y cones-map-mapsto* **by** *force*
hence $\varphi\ (?\psi\ (cones\text{-}map\ y\ \chi')) = cones\text{-}map\ y\ \chi'$
using *φ bij-betw-inv-into-right* [*of φ*] **by** *simp*
thus *?thesis* **by** *auto*
qed
also have ... $= \chi'\ j \cdot y$
using *cone-χ′ j y* **by** *auto*
finally show $cones\text{-}map\ ?f\ (?\chi\ a)\ j \cdot y = \chi'\ j \cdot y$
by *auto*
qed
qed
ultimately show $cones\text{-}map\ ?f\ (?\chi\ a)\ j = \chi'\ j$ **by** *blast*
qed
qed
show $\ll ?f : a' \rightarrow a\gg \wedge cones\text{-}map\ ?f\ (?\chi\ a) = \chi'$
using *f-in-hom f-map* **by** *simp*
show $\bigwedge f'. \ll f' : a' \rightarrow a\gg \wedge cones\text{-}map\ f'\ (?\chi\ a) = \chi' \Longrightarrow f' = ?f$
proof −
fix f'
assume f': $\ll f' : a' \rightarrow a\gg \wedge cones\text{-}map\ f'\ (?\chi\ a) = \chi'$
have *f′-in-hom*: $\ll f' : a' \rightarrow a\gg$ **using** f' **by** *simp*
have *f′-map*: $cones\text{-}map\ f'\ (?\chi\ a) = \chi'$ **using** f' **by** *simp*
show $f' = ?f$
proof (*intro S.arr-eqI′* [*of f′ ?f*])
show $S.par\ f'\ ?f$
using *f-in-hom f′-in-hom* **by** (*elim S.in-homE*, *auto*)

show $\bigwedge y'. \ll y' : S.unity \rightarrow S.dom\ f'\gg \Longrightarrow f' \cdot y' = ?f \cdot y'$
proof –
fix y'
assume y': $\ll y' : S.unity \rightarrow S.dom\ f'\gg$
have *0*: $\varphi\ (f' \cdot y') = cones\text{-}map\ y'\ \chi'$
proof
fix j
have *1*: $\ll f' \cdot y' : S.unity \rightarrow a\gg$ **using** f'-*in-hom* y' **by** *auto*
hence *2*: $\varphi\ (f' \cdot y') \in cones\ S.unity$
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* $\langle\varphi\ (f' \cdot y')\rangle$ **using** *2* **by** *auto*
have $\neg J.arr\ j \Longrightarrow \varphi\ (f' \cdot y')\ j = cones\text{-}map\ y'\ \chi'\ j$
using f' y' *cone-*χ' χ''*.is-extensional mem-Collect-eq restrict-apply*
by (*elim S.in-homE*, *auto*)
moreover have $J.arr\ j \Longrightarrow \varphi\ (f' \cdot y')\ j = cones\text{-}map\ y'\ \chi'\ j$
proof –
assume j: $J.arr\ j$
have *3*: $\ll ?\chi\ a\ j : a \rightarrow D\ (J.cod\ j)\gg$
using j χa*.preserves-hom* **by** *force*
have $\varphi\ (f' \cdot y')\ j = \varphi\ (a \cdot f' \cdot y')\ j$
using a f' y' j *S.comp-cod-arr* **by** (*elim S.in-homE*, *auto*)
also have ... $= ?\chi\ a\ j \cdot f' \cdot y'$
using *1 3* χ [*of a*] a f' y' j **by** *fastforce*
also have ... $= (?\chi\ a\ j \cdot f') \cdot y'$
using *S.comp-assoc* **by** *simp*
also have ... $= cones\text{-}map\ f'\ (?\chi\ a)\ j \cdot y'$
using f' y' j χa*.cone-axioms* **by** *auto*
also have ... $= \chi'\ j \cdot y'$
using f' **by** *blast*
also have ... $= cones\text{-}map\ y'\ \chi'\ j$
using y' j *cone-*χ' f' *mem-Collect-eq restrict-apply* **by** *force*
finally show $\varphi\ (f' \cdot y')\ j = cones\text{-}map\ y'\ \chi'\ j$ **by** *auto*
qed
ultimately show $\varphi\ (f' \cdot y')\ j = cones\text{-}map\ y'\ \chi'\ j$ **by** *auto*
qed
hence $f' \cdot y' = ?\psi\ (cones\text{-}map\ y'\ \chi')$
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* $f' \cdot y'$]
by (*elim S.in-homE*, *auto*)
moreover have $?f \cdot y' = ?\psi\ (cones\text{-}map\ y'\ \chi')$
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* $?f \cdot y'$]
by (*elim S.in-homE*, *auto*)
ultimately show $f' \cdot y' = ?f \cdot y'$ **by** *auto*
qed
qed
qed
qed

qed
have *limit-cone a* (*?χ a*) **..**
thus *?thesis* **by** *auto*
qed
qed

end

context *set-category*
begin

A set category has an equalizer for any parallel pair of arrows.

lemma *has-equalizers*:
shows *has-equalizers*
proof (*unfold has-equalizers-def*)
have $\bigwedge$*f0 f1*. *par f0 f1* $\Longrightarrow$ $\exists$ *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*
apply *unfold-locales* **using** *par* **by** *auto*
interpret *PP*: *diagram-in-set-category J.comp S 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 {*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 *ide-a*: *ide ?a* **using** *1* **by** *auto*
have *set-a*: *set ?a* = *img* ‘ {*e*. *e* ∈ *hom unity* (*dom f0*) ∧ *f0* · *e* = *f1* · *e*}
using *1* **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*)
proof −
have *set ?a* = *img* ‘ {*e*. *e* ∈ *hom unity* (*dom f0*) ∧ *f0* · *e* = *f1* · *e*}
using *img-point-in-Univ set-a* **by** *blast*
thus *?thesis*
using *imageE img-point-elem-set mem-Collect-eq subsetI* **by** *auto*
qed
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* = *λt. 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 ∃ *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*
hence *PP.has-as-limit ?a*
using *ide-a PP.limits-are-sets-of-cones* **by** *simp*
from *this* **obtain** *ε* **where** *ε: limit-cone J.comp S PP.map ?a ε* **by** *auto*
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 have *PP.has-as-equalizer (ε J.Zero)*
using *ε* **by** *simp*
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* **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.

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* $f\ A \equiv \lambda x.\ \mathit{if}\ x \in A\ \mathit{then}\ f\ x\ \mathit{else}\ unity$

lemma *extensionalI* [*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
proof
  fix f
  assume f: $f \in extensional\ A$
  have *1*: $\forall x.\ x \notin A \longrightarrow f\ x = unity$ **using** *f extensional-def* **by** *fast*
  hence $\forall x.\ x \notin B \longrightarrow f\ x = unity$ **using** *assms* **by** *auto*
  thus $f \in extensional\ B$ **using** *extensional-def* **by** *blast*
qed

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* +
  discrete-diagram J S D +
  diagram-in-set-category J S D
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *S* :: *'s comp* (**infixr** $\cdot$ *55*)
and *D* :: *'j* ⇒ *'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* ≡ *Collect J.arr*

definition *funToCone*
where *funToCone F* ≡ λ*j*. *if J.arr j then S.mkPoint* (*D j*) (*F j*) *else S.null*

definition *coneToFun*
where *coneToFun* χ ≡ λ*j*. *if J.arr j then S.img* (χ *j*) *else S.unity*

lemma *funToCone-mapsto*:
shows *funToCone* ∈ *S.PiE I* (*S.set o D*) → *cones S.unity*
proof
  fix *F*
  assume *F*: *F* ∈ *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 *1*: *S.ide* (*S.mkIde S.Univ*) **by** *simp*
  have *cone S.unity* (*funToCone F*)
  proof
    show ⋀*j*. ¬*J.arr j* ⟹ *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 *3*: *S.dom* (*funToCone F j*) = *U.map* (*J.dom j*)
      using *2 j U.map-simp* **by** *auto*
    show *4*: *S.cod* (*funToCone F j*) = *D* (*J.cod j*)
      using *2 j is-discrete* **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 3 j is-discrete U.map-simp S.arr-dom-iff-arr S.comp-arr-dom U.preserves-arr
    by (metis J.ide-char)
  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
    proof
      fix x
      show x ∉ I ⟹ coneToFun χ x = S.unity
        using coneToFun-def by simp
    qed
  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.is-extensional 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 ∈ I ⟹ 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* $\equiv \exists\pi.\ \pi \in$ *PiE I* (λ-. *Univ*) $\rightarrow$ *Univ* $\wedge$ *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: $\pi \in$ *PiE I* (λ-. *Univ*) $\rightarrow$ *Univ* $\wedge$ *inj-on* π (*PiE I* (λ-. *Univ*))
using *admits-tupling-def* **by** *metis*
have $\pi \in$ *PiE* I' (λ-. *Univ*) $\rightarrow$ *Univ*
proof
fix *f*
assume *f*: $f \in$ *PiE* I' (λ-. *Univ*)
have $f \in$ *PiE I* (λ-. *Univ*)
using *assms*(*2*) *f extensional-def* [*of* I'] *terminal-unity extensional-monotone* **by** *auto*
thus π $f \in$ *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* **by** *auto*
ultimately show *?thesis* **using** π *assms*(*2*) **by** *blast*
qed
ultimately show *?thesis* **using** *admits-tupling-def* **by** *metis*
qed

lemma *has-products-iff-admits-tupling*:
fixes I :: *'i set*
shows *has-products* $I \longleftrightarrow I \neq$ *UNIV* $\wedge$ *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: $I \neq$ *UNIV* **using** *has-products has-products-def* **by** *auto*
interpret J: *discrete-category* I ⟨*SOME* x. $x \notin I$⟩
using I *someI-ex* [*of* $\lambda x.\ x \notin I$] **by** (*unfold-locales*, *auto*)
let *?D* = λi. *mkIde Univ*
interpret D: *discrete-diagram-from-map* I S *?D* ⟨*SOME* j. $j \notin I$⟩
using J.*not-arr-null* J.*arr-char*
by (*unfold-locales*, *auto*)
interpret D: *discrete-diagram-in-set-category* J.*comp* S 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* $\Pi D \wedge (\exists\varphi$. *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* $\rightarrow$ *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** *blast*
let $?\pi$ = *img o* (*inv-into* (*hom unity* ΠD) φ) *o* D.*funToCone*
have *1*: D.*funToCone* $\in$ *PiE* I (*set o* D.*map*) $\rightarrow$ 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* $\rightarrow$ *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*
proof (*intro inj-onI*)
fix A :: *'a set* **and** B :: *'b set* **and** C :: *'c set* **and** D :: *'d set*
and F :: $'a \Rightarrow 'b$ **and** G :: $'b \Rightarrow 'c$ **and** H :: $'c \Rightarrow 'd$
assume *a1*: $F \in A \to B \wedge G \in B \to C \wedge H \in C \to D \wedge$
inj-on F A $\wedge$ *inj-on G B* $\wedge$ *inj-on H C*
fix $a\ a'$
assume *a*: $a \in A$ **and** *a'*: $a' \in A$ **and** *eq*: $(H\ o\ G\ o\ F)\ a = (H\ o\ G\ o\ F)\ a'$
have $H\ (G\ (F\ a)) = H\ (G\ (F\ a'))$ **using** *eq* **by** *simp*
moreover have $G\ (F\ a) \in C \wedge G\ (F\ a') \in C$ **using** *a a' a1* **by** *auto*
ultimately have $G\ (F\ a) = G\ (F\ a')$ **using** *a1 inj-onD* **by** *metis*
moreover have $F\ a \in B \wedge F\ a' \in B$ **using** *a a' a1* **by** *auto*
ultimately have $F\ a = F\ a'$ **using** *a1 inj-onD* **by** *metis*
thus $a = a'$ **using** *a a' a1 inj-onD* **by** *metis*
qed
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 $?\pi \in (PiE\ I\ (\lambda x.\ Univ)) \to Univ \wedge inj\text{-}on\ ?\pi\ (PiE\ I\ (\lambda x.\ Univ))$
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: $\pi \in$ (*PiE I* ($\lambda x.$ *Univ*)) $\rightarrow$ *Univ* $\wedge$ *inj-on* π (*PiE I* ($\lambda x.$ *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 I D* $\cap$ *extensional I*) of all π d where $d \in d \in J \rightarrow_E$ *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 $\bigwedge J\ D.$ *discrete-diagram J S D* $\wedge$ *Collect* (*partial-magma.arr J*) $= I$
    $\Longrightarrow \exists \Pi D.$ *has-as-product J D* ΠD
proof
  fix J :: *'i comp* **and** D
  assume D: *discrete-diagram J S D* $\wedge$ *Collect* (*partial-magma.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 D* **..**
  let *?*ΠD = *mkIde* (π ‘ *PiE I* (*set o D*))
  have *0*: *ide ?*ΠD
  proof −
    have *set o D* $\in I \rightarrow$ *Pow Univ*
      using *Pow-iff incl-in-def o-apply elem-set-implies-incl-in*
          set-subset-Univ subsetI
      by (*metis* (*mono-tags*, *lifting*) *Pi-I'*)
    hence π ‘ *PiE I* (*set o D*) $\subseteq$ *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* **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*: $\pi \in$ *PiE I* (*set o D*) $\rightarrow$ *set ?*ΠD $\wedge$ *inj-on* π (*PiE I* (*set o D*))
  proof −
    have *PiE I* (*set o D*) $\subseteq$ *PiE I* ($\lambda x.$ *Univ*)
      using *set-subset-Univ elem-set-implies-incl-in elem-set-implies-set-eq-singleton*
          incl-in-def PiE-mono
      by (*metis comp-apply subsetI*)

```
  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 ∈ set ?ΠD
  have y ∈ π ` (PiE I (set o D)) using y set-ΠD by auto
  thus inv-into (PiE I (set o D)) π y ∈ PiE I (set o D)
    using inv-into-into [of y π PiE I (set o D)] by simp
qed
have 3: ⋀x. x ∈ set ?ΠD ⟹ π (inv-into (PiE I (set o D)) π x) = x
  using set-ΠD by (simp add: f-inv-into-f)
have 4: ⋀d. d ∈ PiE I (set o D) ⟹ inv-into (PiE I (set o D)) π (π d) = d
  using 1 by auto
have 5: D.I = I
  using D by auto
have bij-betw ?φ (D.cones unity) (hom unity ?ΠD)
proof (intro bij-betwI)
  show ?φ ∈ D.cones unity → hom unity ?ΠD
  proof
    fix χ
    assume χ: χ ∈ D.cones unity
    show ?φ χ ∈ hom unity ?ΠD
      using χ 0 1 5 D.coneToFun-mapsto mkPoint-in-hom [of ?ΠD]
      by (simp, blast)
  qed
  show ?φ' ∈ hom unity ?ΠD → D.cones unity
  proof
    fix x
    assume x: x ∈ hom unity ?ΠD
    hence img x ∈ set ?ΠD
      using img-point-elem-set by blast
    hence inv-into (PiE I (set o D)) π (img x) ∈ Pi I (set ∘ D) ∩ local.extensional I
      using 2 by blast
    thus ?φ' x ∈ D.cones unity
      using 5 D.funToCone-mapsto by auto
  qed
  show ⋀x. x ∈ hom unity ?ΠD ⟹ ?φ (?φ' x) = x
  proof −
    fix x
    assume x: x ∈ hom unity ?ΠD
    show ?φ (?φ' x) = x
    proof −
      have D.coneToFun (D.funToCone (inv-into (PiE I (set o D)) π (img x)))
                = inv-into (PiE I (set o D)) π (img x)
        using x 1 5 img-point-elem-set set-ΠD D.coneToFun-funToCone by force
      hence π (D.coneToFun (D.funToCone (inv-into (PiE I (set o D)) π (img x))))
                = img x
        using x 3 img-point-elem-set set-ΠD 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: $\chi \in D.cones\ unity$
show $?\varphi'\ (?\varphi\ \chi) = \chi$
proof −
have *img* (*mkPoint* $?\Pi D$ (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* $?\Pi D$ (π (*D.coneToFun* χ))))
= *D.coneToFun* χ
using χ *D.coneToFun-mapsto 4 5* **by** (*metis PiE*)
hence *D.funToCone* (*inv-into* (*PiE I* (*set o D*)) π
(*img* (*mkPoint* $?\Pi D$ (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*) $?\varphi$) (*hom unity* $?\Pi D$) (*D.cones unity*)
using *bij-betw-inv-into* **by** *blast*
hence $\exists\varphi$. *bij-betw* φ (*hom unity* $?\Pi D$) (*D.cones unity*) **by** *blast*
hence *D.has-as-limit* $?\Pi D$
using ‹*ide* $?\Pi D$› *D.limits-are-sets-of-cones* **by** *simp*
from *this* **obtain** χ **where** χ: *limit-cone J S D* $?\Pi D\ \chi$ **by** *blast*
interpret χ: *limit-cone J S D* $?\Pi D\ \chi$ **using** χ **by** *auto*
interpret *P*: *product-cone J S D* $?\Pi D\ \chi$
using χ *D.product-coneI* **by** *blast*
have *product-cone J S D* $?\Pi D\ \chi$ **..**
thus *has-as-product J D* $?\Pi D$
using *has-as-product-def* **by** *auto*
qed
thus $I \neq UNIV\ \wedge$
($\forall$ *J D. discrete-diagram J S D* $\wedge$ *Collect* (*partial-magma.arr J*) = *I*
$\longrightarrow$ ($\exists\Pi D$. *has-as-product J D* ΠD))
using *ex-*π **by** *blast*
qed
qed

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* :: $'j$) $\longleftrightarrow$ ($\forall I$ :: $'j$ *set*. $I \neq UNIV \longrightarrow$ *admits-tupling* I)
proof
assume *has-limits*: *has-limits* (*undefined* :: $'j$)

show $\forall I :: 'j\ set.\ I \neq UNIV \longrightarrow admits\text{-}tupling\ I$
using *has-limits has-products-if-has-limits has-products-iff-admits-tupling* **by** *blast*
next
assume *admits-tupling*: $\forall I :: 'j\ set.\ I \neq UNIV \longrightarrow admits\text{-}tupling\ I$
show *has-limits* (*undefined* :: *'j*)
proof –
have *1*: $\bigwedge I :: 'j\ set.\ I \neq UNIV \Longrightarrow has\text{-}products\ I$
using *admits-tupling has-products-iff-admits-tupling* **by** *auto*
have $\bigwedge J :: 'j\ comp.\ category\ J \Longrightarrow has\text{-}products\ (Collect\ (partial\text{-}magma.arr\ J))$
proof –
fix *J* :: *'j comp*
assume *J*: *category J*
interpret *J*: *category J* **using** *J* **by** *auto*
have *Collect J.arr* $\neq$ *UNIV* **using** *J.not-arr-null* **by** *blast*
thus *has-products* (*Collect J.arr*)
using *1* **by** *simp*
qed
hence $\bigwedge J :: 'j\ comp.\ category\ J \Longrightarrow has\text{-}limits\text{-}of\text{-}shape\ J$
proof –
fix *J* :: *'j comp*
assume *J*: *category J*
interpret *J*: *category J* **using** *J* **by** *auto*
show *has-limits-of-shape J*
proof –
have *Collect J.arr* $\neq$ *UNIV* **using** *J.not-arr-null* **by** *fast*
moreover have *Collect J.ide* $\neq$ *UNIV* **using** *J.not-arr-null* **by** *blast*
ultimately show *?thesis*
using *1 has-limits-if-has-products J.category-axioms* **by** *metis*
qed
qed
thus *has-limits* (*undefined* :: *'j*)
using *has-limits-def* **by** *metis*
qed
qed

end

18.9 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** $\cdot_J$ *55*)
and A :: $'a$ *comp* (**infixr** $\cdot_A$ *55*)
and B :: $'b$ *comp* (**infixr** $\cdot_B$ *55*)
and D :: $'j * 'a \Rightarrow 'b$
begin

notation *J.in-hom* (≪- : - $\rightarrow_J$ -≫)
notation *JxA.comp* (**infixr** $\cdot_{JxA}$ *55*)
notation *JxA.in-hom* (≪- : - $\rightarrow_{JxA}$ -≫)

A choice of limit cone for each diagram $D\ (-,\ a)$, where a is an object of A, extends to a functor L: $A \rightarrow B$, 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 $\forall a.\ A.ide\ a \longrightarrow limit\text{-}cone\ J\ B\ (\lambda j.\ D\ (j,\ a))\ (l\ a)\ (\chi\ a)$
shows $\bigwedge a.\ A.ide\ a \Longrightarrow L\ l\ \chi\ a = l\ a$
proof –
  let $?L = L\ l\ \chi$
  let $?P = P\ l\ \chi$
  fix a
  assume a: $A.ide\ a$
  interpret χa: $limit\text{-}cone\ J\ B\ \langle\lambda j.\ D\ (j,\ a)\rangle\ \langle l\ a\rangle\ \langle\chi\ a\rangle$ **using** *a assms* **by** *auto*
  have *Pa*: $?P\ a = (\lambda f.\ f \in B.hom\ (l\ a)\ (l\ a)\ \wedge$
      $diagram.cones\text{-}map\ J\ B\ (\lambda j.\ D\ (j,\ a))\ f\ (\chi\ a) = \chi\ a)$
    using *a vcomp-ide-dom* $\chi a.natural\text{-}transformation\text{-}axioms$ **by** *simp*
  have $?P\ a\ (?L\ a)$ **using** *assms a L-arr* $[of\ l\ \chi\ a]$ **by** *fastforce*
  moreover have $?P\ a\ (l\ a)$
  proof –
    have $?P\ a\ (l\ a) \longleftrightarrow l\ a \in B.hom\ (l\ a)\ (l\ a) \wedge \chi a.D.cones\text{-}map\ (l\ a)\ (\chi\ a) = \chi\ a$
      using *Pa* **by** *meson*
    thus *?thesis*
      using *a* $\chi a.ide\text{-}apex$ $\chi a.cone\text{-}axioms$ $\chi a.D.cones\text{-}map\text{-}ide$ $[of\ \chi\ a\ l\ a]$ **by** *force*
  qed
  moreover have $\exists!f.\ ?P\ a\ f$
    using *a Pa* $\chi a.is\text{-}universal$ $\chi a.cone\text{-}axioms$ **by** *force*
  ultimately show $?L\ a = l\ a$ **by** *blast*
qed

lemma *chosen-limits-induce-functor*:
assumes $\forall a.\ A.ide\ a \longrightarrow limit\text{-}cone\ J\ B\ (\lambda j.\ D\ (j,\ a))\ (l\ a)\ (\chi\ a)$
shows $functor\ A\ B\ (L\ l\ \chi)$
proof –
  let $?L = L\ l\ \chi$
  let $?P = \lambda a.\ \lambda f.\ \ll f : l\ (A.dom\ a) \rightarrow_B l\ (A.cod\ a)\gg \wedge$
      $diagram.cones\text{-}map\ J\ B\ (\lambda j.\ D\ (j,\ A.cod\ a))\ f\ (\chi\ (A.cod\ a))$
        $= vertical\text{-}composite.map\ J\ B\ (\chi\ (A.dom\ a))\ (\lambda j.\ D\ (j,\ a))$
  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': $\ll a' : A.cod\ a \rightarrow_A A.cod\ a'\gg$ **using** 1 **by** *auto*
    have $a'a$: $A.seq\ a'\ a$ **using** 1 **by** *auto*
    interpret $\chi\text{-}dom\text{-}a$: $limit\text{-}cone\ J\ B\ \langle\lambda j.\ D\ (j,\ A.dom\ a)\rangle\ \langle l\ (A.dom\ a)\rangle\ \langle\chi\ (A.dom\ a)\rangle$
      using *a assms* **by** *auto*
    interpret $\chi\text{-}cod\text{-}a$: $limit\text{-}cone\ J\ B\ \langle\lambda j.\ D\ (j,\ A.cod\ a)\rangle\ \langle l\ (A.cod\ a)\rangle\ \langle\chi\ (A.cod\ a)\rangle$
      using $a'a$ *assms* **by** *auto*

interpret χ-*dom-a'a*: *limit-cone J B* ‹$\lambda j.\ D\ (j,\ A.dom\ (a' \cdot_A a))$› ‹$l\ (A.dom\ (a' \cdot_A a))$›
‹$\chi\ (A.dom\ (a' \cdot_A a))$›
using *a'a assms* **by** *auto*
interpret χ-*cod-a'a*: *limit-cone J B* ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$› ‹$l\ (A.cod\ (a' \cdot_A a))$›
‹$\chi\ (A.cod\ (a' \cdot_A a))$›
using *a'a assms* **by** *auto*
interpret *Da*: *natural-transformation J B* ‹$\lambda j.\ D\ (j,\ A.dom\ a)$› ‹$\lambda j.\ D\ (j,\ A.cod\ a)$›
‹$\lambda j.\ D\ (j,\ a)$›
using *a fixing-arr-gives-natural-transformation-2* **by** *simp*
interpret *Da'*: *natural-transformation J B* ‹$\lambda j.\ D\ (j,\ A.cod\ a)$› ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$›
‹$\lambda j.\ D\ (j,\ a')$›
using *a a'a fixing-arr-gives-natural-transformation-2* **by** *fastforce*
interpret *Da'o*χ-*cod-a*: *vertical-composite J B*
χ-*cod-a.A.map* ‹$\lambda j.\ D\ (j,\ A.cod\ a)$› ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$›
‹$\chi\ (A.cod\ a)$› ‹$\lambda j.\ D\ (j,\ a')$›..
interpret *Da'o*χ-*cod-a*: *cone J B* ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$› ‹$l\ (A.cod\ a)$› *Da'o*$\chi$-*cod-a.map*
..
interpret *Da'a*: *natural-transformation J B*
‹$\lambda j.\ D\ (j,\ A.dom\ (a' \cdot_A a))$› ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$›
‹$\lambda j.\ D\ (j,\ a' \cdot_A a)$›
using *a'a fixing-arr-gives-natural-transformation-2* [*of* $a' \cdot_A a$] **by** *auto*
interpret *Da'ao*χ-*dom-a'a*:
vertical-composite J B χ-*dom-a'a.A.map* ‹$\lambda j.\ D\ (j,\ A.dom\ (a' \cdot_A a))$›
‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$› ‹$\chi\ (A.dom\ (a' \cdot_A a))$›
‹$\lambda j.\ D\ (j,\ a' \cdot_A a)$› ..
interpret *Da'ao*χ-*dom-a'a*: *cone J B* ‹$\lambda j.\ D\ (j,\ A.cod\ (a' \cdot_A a))$›
‹$l\ (A.dom\ (a' \cdot_A a))$› *Da'ao*$\chi$-*dom-a'a.map* ..
show $?L\ (a' \cdot_A a) = ?L\ a' \cdot_B ?L\ a$
proof −
have $?P\ (a' \cdot_A a)\ (?L\ (a' \cdot_A a))$ **using** *assms a'a L-arr* [*of* $l\ \chi\ a' \cdot_A a$] **by** *fastforce*
moreover have $?P\ (a' \cdot_A a)\ (?L\ a' \cdot_B ?L\ a)$
proof
have *La*: $\ll ?L\ a : l\ (A.dom\ a) \rightarrow_B l\ (A.cod\ a)\gg$
using *assms a L-arr* **by** *fast*
moreover have *La'*: $\ll ?L\ a' : l\ (A.cod\ a) \rightarrow_B l\ (A.cod\ a')\gg$
using *assms a a' L-arr* [*of* $l\ \chi\ a'$] **by** *auto*
ultimately have *seq*: $B.seq\ (?L\ a')\ (?L\ a)$ **by** (*elim B.in-homE*, *auto*)
thus *La'-La*: $\ll ?L\ a' \cdot_B ?L\ a : l\ (A.dom\ (a' \cdot_A a)) \rightarrow_B l\ (A.cod\ (a' \cdot_A a))\gg$
using *a a' 1 La La'* **by** (*intro B.comp-in-homI*, *auto*)
show χ-*cod-a'a.D.cones-map* $(?L\ a' \cdot_B ?L\ a)\ (\chi\ (A.cod\ (a' \cdot_A a)))$
= *Da'ao*χ-*dom-a'a.map*
proof −
have χ-*cod-a'a.D.cones-map* $(?L\ a' \cdot_B ?L\ a)\ (\chi\ (A.cod\ (a' \cdot_A a)))$
= (χ-*cod-a'a.D.cones-map* $(?L\ a)$ *o* χ-*cod-a'a.D.cones-map* $(?L\ a')$)
$(\chi\ (A.cod\ a'))$
proof −
have χ-*cod-a'a.D.cones-map* $(?L\ a' \cdot_B ?L\ a)\ (\chi\ (A.cod\ (a' \cdot_A a))) =$
restrict (χ-*cod-a'a.D.cones-map* $(?L\ a) \circ \chi$-*cod-a'a.D.cones-map* $(?L\ a')$)

(χ-*cod-a'a.D.cones* (*B.cod* (*?L a'*)))
(χ (*A.cod* ($a' \cdot_A a$)))
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'*) $\in$ 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'o*χ-*cod-a.map*
proof −
have *?P a'* (*?L a'*) **using** *assms a' L-arr* [*of l* χ *a'*] **by** *fast*
moreover have
?P a' = (λ*f*. *f* $\in$ *B.hom* (*l* (*A.cod a*)) (*l* (*A.cod a'*)) $\wedge$
χ-*cod-a'a.D.cones-map f* (χ (*A.cod a'*)) = *Da'o*χ-*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*)))
(λ*j*. *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* λ*j*. *D* (*j*, *A.cod a*) λ*j*. *D* (*j*, *A.cod* ($a' \cdot_A a$))
λ*j*. *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*)) (λ*j*. *D* (*j*, *a*)))
(λ*j*. *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* (λ*j*. *D* (*j*, *a*)) (λ*j*. *D* (*j*, *a'*)))
using *a'a Da.natural-transformation-axioms Da'.natural-transformation-axioms*
χ-*dom-a.natural-transformation-axioms*
vcomp-assoc [*of J B* χ-*dom-a.A.map* λ*j*. *D* (*j*, *A.dom a*) χ (*A.dom a*)
λ*j*. *D* (*j*, *A.cod a*) λ*j*. *D* (*j*, *a*)
λ*j*. *D* (*j*, *A.cod a'*) λ*j*. *D* (*j*, *a'*)]
by *auto*

also have
... = *vertical-composite.map J B* (χ (*A.dom* ($a' \cdot_A a$))) (λj. *D* (j, $a' \cdot_A a$))
using *a'a preserves-comp-2* **by** *simp*
finally show *?thesis* **by** *auto*
qed
qed
moreover have $\exists!f$. *?P* ($a' \cdot_A a$) f
using χ*-cod-a'a.is-universal*
[*of l* (*A.dom* ($a' \cdot_A a$))
vertical-composite.map J B (χ (*A.dom* ($a' \cdot_A a$))) (λj. *D* (j, $a' \cdot_A a$))]
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** $\cdot_A$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *55*)
and *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *D* :: *'j* ⇒ (*'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** $\cdot_{JxA}$ *55*)
notation *JxA.in-hom* («- : - $\rightarrow_{JxA}$ -»)

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* (τ j)
shows *at a* τ j = *A-B.Map* (τ 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* [*of a*] **by** (*unfold-locales*, *simp*)
show *at a* χ*.A.map* = *Xa.map*
  using *a x Curry.uncurry-def E.map-def Xa.is-extensional* **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 χ*a*: *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 χ*a*: *cone J B* ‹*at a D*› ‹*A-B.Map x a*› ‹*at a* χ› **..**
  show *cone-*χ*a*: *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*›

$\langle at\ a\ D\rangle$ $\langle at\ a'\ D\rangle$ **..**

interpret *Da'a*: *natural-transformation J B* $\langle at\ (A.dom\ a)\ D\rangle$ $\langle at\ (A.cod\ a')\ D\rangle$ $\langle at\ (a' \cdot_A a)\ D\rangle$
using *assms functor-at-arr-is-transformation* [*of D* $a' \cdot_A a$] *functor-axioms* **by** *simp*
show $at\ (a' \cdot_A a)\ D = Da'oDa.map$
proof (*intro NaturalTransformation.eqI*)
show *natural-transformation J B* $(at\ (A.dom\ a)\ D)$ $(at\ (A.cod\ a')\ D)$ $Da'oDa.map$ **..**
show *natural-transformation J B* $(at\ (A.dom\ a)\ D)$ $(at\ (A.cod\ a')\ D)$ $(at\ (a' \cdot_A a)\ D)$ **..**
show $\bigwedge j.\ J.ide\ j \Longrightarrow at\ (a' \cdot_A a)\ D\ j = Da'oDa.map\ j$
proof −
fix *j*
assume *j*: *J.ide j*
interpret *Dj*: *functor A B* $\langle A\text{-}B.Map\ (D\ j)\rangle$
using *j preserves-ide A-B.ide-char* **by** *simp*
show $at\ (a' \cdot_A a)\ D\ j = Da'oDa.map\ j$
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*: $f \in A\text{-}B.hom\ x'\ x$
shows $cones\text{-}map\ f\ \chi = \chi' \longleftrightarrow$
$(\forall a.\ A.ide\ a \longrightarrow diagram.cones\text{-}map\ J\ B\ (at\ a\ D)\ (A\text{-}B.Map\ f\ a)\ (at\ a\ \chi) = at\ a\ \chi')$
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'* $\langle cones\text{-}map\ f\ \chi\rangle$
using *x' f assms*(*1*) *cones-map-mapsto* **by** *blast*
interpret *Fun-x*: *functor A B* $\langle A\text{-}B.Map\ x\rangle$ **using** *x A-B.ide-char* **by** *simp*
interpret *Fun-x'*: *functor A B* $\langle A\text{-}B.Map\ x'\rangle$ **using** *x' A-B.ide-char* **by** *simp*
show $cones\text{-}map\ f\ \chi = \chi' \Longrightarrow$
$(\forall a.\ A.ide\ a \longrightarrow diagram.cones\text{-}map\ J\ B\ (at\ a\ D)\ (A\text{-}B.Map\ f\ a)\ (at\ a\ \chi) = at\ a\ \chi')$
proof −
assume χ': $cones\text{-}map\ f\ \chi = \chi'$
have $\bigwedge a.\ A.ide\ a \Longrightarrow diagram.cones\text{-}map\ J\ B\ (at\ a\ D)\ (A\text{-}B.Map\ f\ a)\ (at\ a\ \chi) = at\ a\ \chi'$
proof −
fix *a*
assume *a*: *A.ide a*
interpret *Da*: *diagram J B* $\langle at\ a\ D\rangle$ **using** *a at-ide-is-diagram* **by** *auto*
interpret χa: *cone J B* $\langle at\ a\ D\rangle$ $\langle A\text{-}B.Map\ x\ a\rangle$ $\langle at\ a\ \chi\rangle$
using *a assms*(*1*) *cone-at-ide-is-cone* **by** *simp*
interpret $\chi' a$: *cone J B* $\langle at\ a\ D\rangle$ $\langle A\text{-}B.Map\ x'\ a\rangle$ $\langle at\ a\ \chi'\rangle$
using *a assms*(*2*) *cone-at-ide-is-cone* **by** *simp*
have *1*: $\ll A\text{-}B.Map\ f\ a : A\text{-}B.Map\ x'\ a \rightarrow_B A\text{-}B.Map\ x\ a\gg$
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.is-extensional χfa.is-extensional* [*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 NaturalTransformation.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* [*of J.dom j*] *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* [*of cones-map f χ j*] *A-B.cod-char J.ide-in-hom*
χf.preserves-hom [*of j J.dom j J.cod j*]
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 NaturalTransformation.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-χ*f-j Cod-*χ*f-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 $\chi' j$: *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 $\bigwedge$*a. A.ide a* $\Longrightarrow$ *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-*χ*a*: *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-*χ*a* **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* $\in$ *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*.is-extensional* **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-*χ*a 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* $\chi' j$*.is-extensional* **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
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 $A\text{-}B.Map\ a\ x$ for the diagram in C obtained by evaluating D at x.

lemma *cone-is-limit-if-pointwise-limit*:
assumes *cone-χ*: $cone\ x\ \chi$
and $\forall\, a.\ A.ide\ a \longrightarrow diagram.limit\text{-}cone\ J\ B\ (at\ a\ D)\ (A\text{-}B.Map\ x\ a)\ (at\ a\ \chi)$
shows $limit\text{-}cone\ x\ \chi$
proof –
  interpret χ: $cone\ J\ A\text{-}B.comp\ D\ x\ \chi$ **using** *assms* **by** *auto*
  have x: $A\text{-}B.ide\ x$ **using** $\chi.ide\text{-}apex$ **by** *auto*
  show $limit\text{-}cone\ x\ \chi$
  proof
    fix $x'\ \chi'$
    assume *cone-χ'*: $cone\ x'\ \chi'$
    interpret χ': $cone\ J\ A\text{-}B.comp\ D\ x'\ \chi'$ **using** *cone-χ'* **by** *auto*
    have x': $A\text{-}B.ide\ x'$ **using** $\chi'.ide\text{-}apex$ **by** *auto*

The universality of the limit cone $at\ a\ \chi$ yields, for each object a of A, a unique arrow fa that transforms $at\ a\ \chi$ to $at\ a\ \chi'$.

    have *EU*: $\bigwedge a.\ A.ide\ a \Longrightarrow$
        $\exists!fa.\ fa \in B.hom\ (A\text{-}B.Map\ x'\ a)\ (A\text{-}B.Map\ x\ a) \wedge$
            $diagram.cones\text{-}map\ J\ B\ (at\ a\ D)\ fa\ (at\ a\ \chi) = at\ a\ \chi'$
    proof –
      fix a
      assume a: $A.ide\ a$
      interpret *Da*: $diagram\ J\ B\ \langle at\ a\ D\rangle$ **using** *a at-ide-is-diagram* **by** *auto*
      interpret χa: $limit\text{-}cone\ J\ B\ \langle at\ a\ D\rangle\ \langle A\text{-}B.Map\ x\ a\rangle\ \langle at\ a\ \chi\rangle$
        using *assms(2) a* **by** *auto*
      interpret $\chi' a$: $cone\ J\ B\ \langle at\ a\ D\rangle\ \langle A\text{-}B.Map\ x'\ a\rangle\ \langle at\ a\ \chi'\rangle$
        using *a cone-χ' cone-at-ide-is-cone* **by** *auto*
      have $Da.cone\ (A\text{-}B.Map\ x'\ a)\ (at\ a\ \chi')$ **..**
      thus $\exists!fa.\ fa \in B.hom\ (A\text{-}B.Map\ x'\ a)\ (A\text{-}B.Map\ x\ a) \wedge$
          $Da.cones\text{-}map\ fa\ (at\ a\ \chi) = at\ a\ \chi'$
        using $\chi a.is\text{-}universal$ **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 $\exists!f.\ \langle\!\langle f : x' \rightarrow_{[A,B]} x\rangle\!\rangle \wedge cones\text{-}map\ f\ \chi = \chi'$
    proof
      let $?P = \lambda a\ fa.\ \langle\!\langle fa : A\text{-}B.Map\ x'\ a \rightarrow_B A\text{-}B.Map\ x\ a\rangle\!\rangle \wedge$
          $diagram.cones\text{-}map\ J\ B\ (at\ a\ D)\ fa\ (at\ a\ \chi) = at\ a\ \chi'$
      have *AaPa*: $\bigwedge a.\ A.ide\ a \Longrightarrow ?P\ a\ (THE\ fa.\ ?P\ a\ fa)$
      proof –
        fix a

assume *a*: *A.ide a*
have $\exists!$*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*:
$\bigwedge$*a. A.ide a* $\Longrightarrow$ $\ll$*THE fa. ?P a fa* : *A-B.Map x' a* $\rightarrow_B$ *A-B.Map x a*$\gg$
using *AaPa* **by** *blast*
have *AaPa-map*:
$\bigwedge$*a. A.ide a* $\Longrightarrow$
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* = λ*a. if A.ide a then (THE fa. ?P a fa) else B.null*
interpret *Fun-x*: *functor A B* ‹λ*a. A-B.Map x a*›
using *x A-B.ide-char* **by** *simp*
interpret *Fun-x'*: *functor A B* ‹λ*a. 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*
‹λ*a. A-B.Map x' a*› ‹λ*a. A-B.Map x a*› *?Fun-f*
proof
fix *a*
assume *a*: *A.ide a*
show $\ll$*?Fun-f a* : *A-B.Map x' a* $\rightarrow_B$ *A-B.Map x a*$\gg$ **using** *a AaPa* **by** *simp*
next
fix *a*
assume *a*: *A.arr a*

$$\begin{tikzcd}
x_{\mathrm{dom}\,a} \arrow[r, "x_a"] \arrow[d, "\chi_{\mathrm{dom}\,a}"'] \arrow[dr, phantom, "(A)"] & x_{\mathrm{cod}\,a} \arrow[d, "\chi_{\mathrm{cod}\,a}"] \\
D_{\mathrm{dom}\,a} \arrow[r, "D_a"] \arrow[dr, phantom, "(B)"] & D_{\mathrm{cod}\,a} \\
x'_{\mathrm{dom}\,a} \arrow[r, "x'_a"'] \arrow[u, "\chi'_{\mathrm{dom}\,a}"] \arrow[uu, bend left=80, "f_{\mathrm{dom}\,a}", "(C)"'] & x'_{\mathrm{cod}\,a} \arrow[u, "x'_{\mathrm{cod}\,a}"'] \arrow[uu, bend right=80, "f_{\mathrm{cod}\,a}"']
\end{tikzcd}$$

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*: $\ll$*?x-a* : *?x-dom-a* $\rightarrow_B$ *?x-cod-a*$\gg$
using *a x A-B.ide-char* **by** *auto*
have *x-dom-a*: *B.ide ?x-dom-a* **using** *a* **by** *simp*
have *x-cod-a*: *B.ide ?x-cod-a* **using** *a* **by** *simp*
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
      have x′-dom-a: B.ide ?x′-dom-a using a by simp
      have x′-cod-a: B.ide ?x′-cod-a using a by simp
      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*) χ and *at* (*A.cod a*) χ' 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.is-extensional* *χ-cod-a.cone-axioms* *x-a* **by** *force*
moreover have
$J.arr\ j \Longrightarrow Dao\chi\text{-}dom\text{-}a.map\ j = D\text{-}cod\text{-}a.cones\text{-}map\ ?x\text{-}a\ (at\ (A.cod\ a)\ \chi)\ j$
proof −
assume j: $J.arr\ j$
have $Dao\chi\text{-}dom\text{-}a.map\ j = at\ a\ D\ j \cdot_B at\ (A.dom\ a)\ \chi\ (J.dom\ j)$
using *j Daoχ-dom-a.map-simp-2* **by** *simp*
also have ... $= A\text{-}B.Map\ (D\ j)\ a \cdot_B A\text{-}B.Map\ (\chi\ (J.dom\ j))\ (A.dom\ a)$
using *a j at-simp* **by** *simp*
also have ... $= A\text{-}B.Map\ (A\text{-}B.comp\ (D\ j)\ (\chi\ (J.dom\ j)))\ a$
using *a j A-B.Map-comp*
by (*metis* (*no-types*, *lifting*) *A.comp-arr-dom* *χ.is-natural-1*
χ.preserves-reflects-arr)
also have ... $= A\text{-}B.Map\ (A\text{-}B.comp\ (\chi\ (J.cod\ j))\ (\chi.A.map\ j))\ a$
using *a j χ.naturality* **by** *simp*
also have ... $= A\text{-}B.Map\ (\chi\ (J.cod\ j))\ (A.cod\ a) \cdot_B A\text{-}B.Map\ x\ a$
using *a j x A-B.Map-comp*
by (*metis* (*no-types*, *lifting*) *A.comp-cod-arr* *χ.A.map-simp* *χ.is-natural-2*
χ.preserves-reflects-arr)
also have ... $= at\ (A.cod\ a)\ \chi\ (J.cod\ j) \cdot_B A\text{-}B.Map\ x\ a$
using *a j at-simp* **by** *simp*
also have ... $= at\ (A.cod\ a)\ \chi\ j \cdot_B A\text{-}B.Map\ x\ a$
using *a j χ-cod-a.is-natural-2* *χ-cod-a.A.map-simp*
by (*metis* *J.arr-cod-iff-arr* *J.cod-cod*)
also have ... $= D\text{-}cod\text{-}a.cones\text{-}map\ ?x\text{-}a\ (at\ (A.cod\ a)\ \chi)\ j$
using *a j x χ-cod-a.cone-axioms* *preserves-cod* **by** *simp*
finally show *?thesis* **by** *blast*
qed
ultimately show $Dao\chi\text{-}dom\text{-}a.map\ j = D\text{-}cod\text{-}a.cones\text{-}map\ ?x\text{-}a\ (at\ (A.cod\ a)\ \chi)\ j$
by *blast*
qed
have B: $Dao\chi'\text{-}dom\text{-}a.map = D\text{-}cod\text{-}a.cones\text{-}map\ ?x'\text{-}a\ (at\ (A.cod\ a)\ \chi')$
proof
fix j
have
$\neg J.arr\ j \Longrightarrow Dao\chi'\text{-}dom\text{-}a.map\ j = D\text{-}cod\text{-}a.cones\text{-}map\ ?x'\text{-}a\ (at\ (A.cod\ a)\ \chi')\ j$
using *Daoχ'-dom-a.is-extensional* *χ'-cod-a.cone-axioms* *x'-a* **by** *force*
moreover have
$J.arr\ j \Longrightarrow Dao\chi'\text{-}dom\text{-}a.map\ j = D\text{-}cod\text{-}a.cones\text{-}map\ ?x'\text{-}a\ (at\ (A.cod\ a)\ \chi')\ j$
proof −
assume j: $J.arr\ j$
have $Dao\chi'\text{-}dom\text{-}a.map\ j = at\ a\ D\ j \cdot_B at\ (A.dom\ a)\ \chi'\ (J.dom\ j)$
using *j Daoχ'-dom-a.map-simp-2* **by** *simp*
also have ... $= A\text{-}B.Map\ (D\ j)\ a \cdot_B A\text{-}B.Map\ (\chi'\ (J.dom\ j))\ (A.dom\ a)$
using *a j at-simp* **by** *simp*
also have ... $= A\text{-}B.Map\ (A\text{-}B.comp\ (D\ j)\ (\chi'\ (J.dom\ j)))\ a$
using *a j A-B.Map-comp*
by (*metis* (*no-types*, *lifting*) *A.comp-arr-dom* *χ'.is-natural-1*
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′.is-natural-2
                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.is-natural-2 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 NaturalTransformation.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 1: ⋀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
        have D-cod-a.cone ?x-dom-a Daoχ-dom-a.map ..
        thus D-cod-a.cone ?x′-dom-a (D-cod-a.cones-map ?f-dom-a Daoχ-dom-a.map)
          using f-dom-a 1 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* $\cdot_B$ *at* (*A.dom a*) χ *j*) $\cdot_B$ *?f-dom-a*
using *j Dao*χ*-dom-a.map-simp-ide* **by** *simp*
also have ... = *at a D j* $\cdot_B$ *at* (*A.dom a*) χ *j* $\cdot_B$ *?f-dom-a*
using *B.comp-assoc* **by** *simp*
also have ... = *at a D j* $\cdot_B$ *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* $\cdot_B$ *at* (*A.dom a*) χ' *j*
using *a AaPa A.ide-dom* **by** *presburger*
also have ... = *Dao*χ'*-dom-a.map j*
using *j Dao*χ'*-dom-a.map-simp-ide* **by** *simp*
finally show
D-cod-a.cones-map ?f-dom-a Daoχ*-dom-a.map j* = *Dao*χ'*-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 *Dao*χ'*-cod-a*, thus they are equal by the universality of *at* (*A.cod a*) χ.

have $\exists!$*fa*. «*fa* : *?x'-dom-a* $\rightarrow_B$ *?x-cod-a*» $\wedge$
D-cod-a.cones-map fa (*at* (*A.cod a*) χ) = *Dao*χ'*-dom-a.map*
using *Dao*χ'*-dom-a.cone-axioms a* χ*-cod-a.is-universal* [*of ?x'-dom-a Dao*χ'*-dom-a.map*]
by *fast*
moreover have
?f-cod-a $\cdot_B$ *?x'-a* $\in$ *B.hom ?x'-dom-a ?x-cod-a* $\wedge$
D-cod-a.cones-map (*?f-cod-a* $\cdot_B$ *?x'-a*) (*at* (*A.cod a*) χ) = *Dao*χ'*-dom-a.map*
proof
show *?f-cod-a* $\cdot_B$ *?x'-a* $\in$ *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* $\cdot_B$ *?x'-a*) (*at* (*A.cod a*) χ) = *Dao*χ'*-dom-a.map*
proof −
have *1*: *B.arr* (*?f-cod-a* $\cdot_B$ *?x'-a*)
using *f-cod-a x'-a* **by** (*elim B.in-homE*, *auto*)
hence *D-cod-a.cones-map* (*?f-cod-a* $\cdot_B$ *?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 *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 ... = *Dao*χ'*-dom-a.map*
using *a B AaPa-map A.ide-cod* **by** *presburger*
finally show *?thesis* **by** *auto*
qed
qed

moreover have
    $?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a \in B.hom\ ?x'\text{-}dom\text{-}a\ ?x\text{-}cod\text{-}a\ \wedge$
     $D\text{-}cod\text{-}a.cones\text{-}map\ (?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a)\ (at\ (A.cod\ a)\ \chi) = Dao\chi'\text{-}dom\text{-}a.map$
proof
  show $?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a \in B.hom\ ?x'\text{-}dom\text{-}a\ ?x\text{-}cod\text{-}a$
    using *f-dom-a x-a* **by** *blast*
  show $D\text{-}cod\text{-}a.cones\text{-}map\ (?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a)\ (at\ (A.cod\ a)\ \chi) = Dao\chi'\text{-}dom\text{-}a.map$
  proof –
    have
      $D\text{-}cod\text{-}a.cones\ (B.cod\ (A\text{-}B.Map\ x\ a)) = D\text{-}cod\text{-}a.cones\ (A\text{-}B.Map\ x\ (A.cod\ a))$
      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
        $D\text{-}cod\text{-}a.cones\text{-}map\ (?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a)\ (at\ (A.cod\ a)\ \chi)$
          $= restrict\ (D\text{-}cod\text{-}a.cones\text{-}map\ ?f\text{-}dom\text{-}a\ o\ D\text{-}cod\text{-}a.cones\text{-}map\ ?x\text{-}a)$
                  $(D\text{-}cod\text{-}a.cones\ (?x\text{-}cod\text{-}a))$
                  $(at\ (A.cod\ a)\ \chi)$
      using *D-cod-a.cones-map-comp* [*of ?x-a ?f-dom-a*] *x-a* **by** *argo*
    also have ... $= D\text{-}cod\text{-}a.cones\text{-}map\ ?f\text{-}dom\text{-}a$
                  $(D\text{-}cod\text{-}a.cones\text{-}map\ ?x\text{-}a\ (at\ (A.cod\ a)\ \chi))$
      using *χ-cod-a.cone-axioms* **by** *simp*
    also have ... $= Dao\chi'\text{-}dom\text{-}a.map$
      using *A C a AaPa* **by** *argo*
    finally show *?thesis* **by** *blast*
  qed
qed
ultimately show $?f\text{-}cod\text{-}a \cdot_B ?x'\text{-}a = ?x\text{-}a \cdot_B ?f\text{-}dom\text{-}a$
  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 $?f = A\text{-}B.MkArr\ (\lambda a.\ A\text{-}B.Map\ x'\ a)\ (\lambda a.\ A\text{-}B.Map\ x\ a)\ \varphi.map$
have *f-in-hom*: $?f \in A\text{-}B.hom\ x'\ x$
proof –
  have *arr-f*: *A-B.arr ?f*
    using x' *x A-B.arr-MkArr φ.natural-transformation-axioms* **by** *simp*
  moreover have $A\text{-}B.MkIde\ (\lambda a.\ A\text{-}B.Map\ x\ a) = x$
    using *x A-B.ide-char A-B.MkArr-Map A-B.in-homE A-B.ide-in-hom* **by** *metis*
  moreover have $A\text{-}B.MkIde\ (\lambda a.\ A\text{-}B.Map\ x'\ a) = x'$
    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*: $\bigwedge a.\ A.ide\ a \Longrightarrow A\text{-}B.Map\ ?f\ a = (THE\ fa.\ ?P\ a\ fa)$
  using *f-in-hom φ.map-simp-ide* **by** *fastforce*
have *cones-map-f*: $cones\text{-}map\ ?f\ \chi = \chi'$

```
      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 A-B.arr (A-B.MkArr (A-B.Dom f') (A-B.Cod f') (A-B.Map f'))
          using f' calculation by blast
        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 NaturalTransformation.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
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 $\bigwedge D.$ *diagram J comp D* $\Longrightarrow (\exists x\ \chi.$ *limit-cone J comp D x* $\chi)$
  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* = $\lambda a.$ *diagram.some-limit J B (D.at a D)*
    let *?*χ = $\lambda a.$ *diagram.some-limit-cone J B (D.at a D)*
    have *l*χ: $\bigwedge a.$ *A.ide a* $\Longrightarrow$ *diagram.limit-cone J B (D.at a D) (?l a) (?*χ *a)*
    proof −
      fix *a*
      assume *a*: *A.ide a*
      interpret *Da*: *diagram J B ‹D.at a D›*
        using *a D.at-ide-is-diagram* **by** *blast*
      show *limit-cone J B (D.at a D) (?l a) (?*χ *a)*
        using *assms(2) B.has-limits-of-shape-def Da.diagram-axioms*
              Da.limit-cone-some-limit-cone
        by *auto*
    qed

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 *l*χ *uncurry-D.chosen-limits-induce-functor [of ?l ?*χ*]* **by** *simp*
    have *L-ide*: $\bigwedge a.$ *A.ide a* $\Longrightarrow$ *?L a = ?l a*

using *uncurry-D.L-ide* [*of ?l ?χ*] *lχ* **by** *blast*
have *L-arr*: $\bigwedge a.\ A.arr\ a \Longrightarrow (\exists!f.\ ?P\ a\ f) \wedge ?P\ a\ (?L\ a)$
using *uncurry-D.L-arr* [*of ?l ?χ*] *lχ* **by** *blast*
have *L-arr-in-hom*: $\bigwedge a.\ A.arr\ a \Longrightarrow \langle\!\langle ?L\ a : ?l\ (A.dom\ a) \rightarrow_B ?l\ (A.cod\ a)\rangle\!\rangle$
using *L-arr* **by** *blast*
have *L-map*: $\bigwedge a.\ A.arr\ a \Longrightarrow uncurry\text{-}D.P\ ?l\ ?\chi\ a\ (uncurry\text{-}D.L\ ?l\ ?\chi\ a)$
using *L-arr* **by** *blast*

The functor L extends to a functor L' from JxA to B that is constant on J.

let $?L' = \lambda ja.$ *if JxA.arr ja then ?L (snd ja) else B.null*
let $?P' = \lambda ja.\ ?P\ (snd\ ja)$
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 $\bigwedge ja.\ JxA.arr\ ja \Longrightarrow (\exists!f.\ ?P'\ ja\ f) \wedge ?P'\ ja\ (?L'\ ja)$
proof −
fix *ja*
assume *ja*: *JxA.arr ja*
have *A.arr* (*snd ja*) **using** *ja* **by** *blast*
thus $(\exists!f.\ ?P'\ ja\ f) \wedge ?P'\ ja\ (?L'\ ja)$
using *ja L-arr* **by** *presburger*
qed
hence L'*-arr*: $\bigwedge ja.\ JxA.arr\ ja \Longrightarrow ?P'\ ja\ (?L'\ ja)$ **by** *blast*
have L'*-arr-in-hom*:
$\bigwedge ja.\ JxA.arr\ ja \Longrightarrow \langle\!\langle ?L'\ ja : ?l\ (A.dom\ (snd\ ja)) \rightarrow_B ?l\ (A.cod\ (snd\ ja))\rangle\!\rangle$
using L'*-arr* **by** *simp*
have L'*-ide*: $\bigwedge ja.\ [\![\ J.arr\ (fst\ ja);\ A.ide\ (snd\ ja)\ ]\!] \Longrightarrow ?L'\ ja = ?l\ (snd\ ja)$
using *L-ide lχ* **by** *force*
have L'*-arr-map*:
$\bigwedge ja.\ JxA.arr\ ja \Longrightarrow uncurry\text{-}D.P\ ?l\ ?\chi\ (snd\ ja)\ (uncurry\text{-}D.L\ ?l\ ?\chi\ (snd\ ja))$
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 $?\chi' = \lambda ja.\ ?\chi\ (snd\ ja)\ (fst\ ja)$
interpret χ': *transformation-by-components JxA.comp B ?L'* ‹*Curry.uncurry D*› $?\chi'$
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*› ‹$?\chi\ ?a$›
using *ja lχ* **by** *blast*
show $\langle\!\langle ?\chi'\ ja : ?L'\ ja \rightarrow_B Curry.uncurry\ D\ ja\rangle\!\rangle$
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)
      proof −
        have B.seq (D.at ?a D (J.cod ?j)) (D.at (A.dom ?a) D ?j)
          using j ja by auto
        moreover have B.seq (D.at (A.dom ?a) D ?j) (?χ' (JxA.dom ja))
          using j ja by fastforce
        ultimately show ?thesis using B.comp-assoc by force
      qed
      also have ... = B (D.at ?a D ?j) (?χ' (JxA.dom ja))
      proof −
        have D.at ?a D (J.cod ?j) ·B D.at (A.dom ?a) D ?j =
                Map (D (J.cod ?j)) ?a ·B Map (D ?j) (A.dom ?a)
          using ja D.at-simp by auto
        also have ... = Map (comp (D (J.cod ?j)) (D ?j)) (?a ·A A.dom ?a)
```

```
          using ja Map-comp D.preserves-hom
          by (metis (mono-tags, lifting) A.comp-arr-dom D.natural-transformation-axioms
              D.preserves-arr a j natural-transformation.is-natural-2)
        also have ... = D.at ?a D ?j
          using ja D.at-simp dom-char A.comp-arr-dom by force
        finally show ?thesis by auto
      qed
      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›
  proof
    show ide (MkIde ?L)
      using L.natural-transformation-axioms MkArr-in-hom ide-in-hom L.functor-axioms
      by blast
  qed

  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.is-extensional by simp
    moreover have J.arr j ⟹ constL.map j = Curry.curry ?L' ?L' ?L' j
    proof -
      assume j: J.arr j
      show constL.map j = Curry.curry ?L' ?L' ?L' j
      proof -
        have constL.map j = MkIde ?L using j constL.map-simp by simp
        moreover have ... = MkArr ?L ?L ?L by simp
        moreover have ... = MkArr (λa. ?L' (J.dom j, a)) (λa. ?L' (J.cod j, a))
                                (λa. ?L' (j, a))
          using j constL.value-is-ide in-homE ide-in-hom by (intro MkArr-eqI, auto)
        moreover have ... = Curry.curry ?L' ?L' ?L' j
          using j Curry.curry-def by auto
        ultimately show ?thesis by force
      qed
    qed
    ultimately show constL.map j = Curry.curry ?L' ?L' ?L' j by blast
  qed
  hence uncurry-constL: Curry.uncurry constL.map = ?L'
    using L'.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 1: Curry.curry (Curry.uncurry D) (Curry.uncurry D) (Curry.uncurry D) = D
```

```
      using Curry.curry-uncurry D.functor-axioms D.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 NaturalTransformation.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)
  proof −
    have ⋀a. A.ide a ⟹ Map (MkIde ?L) a = ?l a
      using L.functor-axioms L-ide by simp
    thus ?thesis
      using 1 2 curry-χ'.cone-axioms curry-L' D.cone-is-limit-if-pointwise-limit by simp
```

qed
thus $\exists\, x\ \chi.$ *limit-cone J comp D x* χ **by** *blast*
qed
thus $\forall\, D.$ *diagram J comp D* $\longrightarrow$ ($\exists\, x\ \chi.$ *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

18.10 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 Yoχ-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 Yoχ.cone-axioms by fastforce
have eval-at-ide: ⋀j. J.ide j ⟹ YoD.at a' (map ∘ D) j = Hom.map (a', D j)
proof -
  fix j
  assume j: J.ide j
  have YoD.at a' (map ∘ 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 ∘ D) j = Hom.map (a', D j) by auto
qed
have eval-at-arr: ⋀j. J.arr j ⟹ YoD.at a' (map ∘ χ) j = Hom.map (a', χ j)
proof -
  fix j
  assume j: J.arr j
  have YoD.at a' (map ∘ 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 ∘ 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 $\ll\psi\ (a',\ D\ j)\ (S.Fun\ (\sigma\ j)\ e) : A'.map\ j \rightarrow D\ j\gg$
  using $e\ j\ S.Fun\text{-}mapsto\ [of\ \sigma\ j]\ A'.preserves\text{-}ide\ Hom.set\text{-}map\ eval\text{-}at\text{-}ide$
    $Hom.\psi\text{-}mapsto\ [of\ A'.map\ j\ D\ j]$
  by *force*
next
fix j
assume j: $J.arr\ j$
show $\psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (\sigma\ (J.cod\ j))\ e) \cdot A'.map\ j =$
    $D\ j \cdot \psi\ (a',\ D\ (J.dom\ j))\ (S.Fun\ (\sigma\ (J.dom\ j))\ e)$
proof −
  have 1: $Y\ (D\ j)\ a' =$
      $S.mkArr\ (Hom.set\ (a',\ D\ (J.dom\ j)))\ (Hom.set\ (a',\ D\ (J.cod\ j)))$
        $(\varphi\ (a',\ D\ (J.cod\ j)) \circ C\ (D\ j) \circ \psi\ (a',\ D\ (J.dom\ j)))$
    using $j\ a'\ D.preserves\text{-}hom$
      $Y\text{-}arr\text{-}ide\ [of\ a'\ D\ j\ D\ (J.dom\ j)\ D\ (J.cod\ j)]$
    by *blast*
  have $\psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (\sigma\ (J.cod\ j))\ e) \cdot A'.map\ j =$
    $\psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (\sigma\ (J.cod\ j))\ e) \cdot a'$
    using $A'.map\text{-}simp\ j$ **by** *simp*
  also have ... $= \psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (\sigma\ (J.cod\ j))\ e)$
  proof −
    have $\psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (\sigma\ (J.cod\ j))\ e) \in C.hom\ a'\ (D\ (J.cod\ j))$
      using $a'\ e\ j\ Hom.\psi\text{-}mapsto\ [of\ A'.map\ j\ D\ (J.cod\ j)]\ A'.map\text{-}simp$
        $S.Fun\text{-}mapsto\ [of\ \sigma\ (J.cod\ j)]\ Hom.set\text{-}map\ eval\text{-}at\text{-}ide$
      by *auto*
    thus *?thesis*
      using $C.comp\text{-}arr\text{-}dom$ **by** *fastforce*
  qed
  also have ... $= \psi\ (a',\ D\ (J.cod\ j))\ (S.Fun\ (Y\ (D\ j)\ a')\ (S.Fun\ (\sigma\ (J.dom\ j))\ e))$
  proof −
    have $S.Fun\ (Y\ (D\ j)\ a')\ (S.Fun\ (\sigma\ (J.dom\ j))\ e) =$
      $(S.Fun\ (Y\ (D\ j)\ a')\ o\ S.Fun\ (\sigma\ (J.dom\ j)))\ e$
      by *simp*
    also have ... $= S.Fun\ (Y\ (D\ j)\ a' \cdot_S \sigma\ (J.dom\ j))\ e$
      using $a'\ e\ j\ Y\text{-}arr\text{-}ide(1)\ S.in\text{-}homE\ \sigma j\ eval\text{-}at\text{-}ide\ S.Fun\text{-}comp$ **by** *force*
    also have ... $= S.Fun\ (\sigma\ (J.cod\ j))\ e$
      using $a'\ j\ x\ \sigma.is\text{-}natural\text{-}2\ \sigma.A.map\text{-}simp\ S.comp\text{-}arr\text{-}dom\ J.arr\text{-}cod\text{-}iff\text{-}arr$
        $J.cod\text{-}cod\ YoD.preserves\text{-}arr\ \sigma.is\text{-}natural\text{-}1\ YoD.at\text{-}simp$
      by *auto*
    finally have
      $S.Fun\ (Y\ (D\ j)\ a')\ (S.Fun\ (\sigma\ (J.dom\ j))\ e) = S.Fun\ (\sigma\ (J.cod\ j))\ e$
      by *auto*
    thus *?thesis* **by** *simp*
  qed
  also have ... $= D\ j \cdot \psi\ (a',\ D\ (J.dom\ j))\ (S.Fun\ (\sigma\ (J.dom\ j))\ e)$
  proof −
    have $e \in S.Dom\ (\sigma\ (J.dom\ j))$
      using $e\ j$ **by** *simp*
    hence $S.Fun\ (\sigma\ (J.dom\ j))\ e \in S.Cod\ (\sigma\ (J.dom\ j))$

```
          using e j S.Fun-mapsto [of σ (J.dom j)] by auto
        hence 2: S.Fun (σ (J.dom j)) e ∈ Hom.set (a', D (J.dom j))
        proof -
          have YoD.at a' (map ∘ D) (J.dom j) = S.mkIde (Hom.set (a', D (J.dom j)))
            using a' j YoD.at-simp by (simp add: eval-at-ide)
          moreover have S.Cod (σ (J.dom j)) = Hom.set (a', D (J.dom j))
            using a' e j Hom.set-map YoD.at-simp eval-at-ide by simp
          ultimately show ?thesis
            using a' e j σj S.Fun-mapsto [of σ (J.dom j)] Hom.set-map
            by auto
        qed
        hence 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
            by (auto simp add: eval-at-ide)
          hence 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' j Hom.ψ-mapsto [of a' D (J.dom j)] by auto
          thus ?thesis
            using a' e j 2 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
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: ⋀e. e ∈ S.set x ⟹ ∃!fe. «fe : a' → a» ∧ D.cones-map fe χ = ?κ 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))
              (λe. φ (a', a) (χ.induced-arrow a' (?κ e)))
have 0: (λe. 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
hence f: «?f : x →S Hom.map (a', a)»
  using a a' x σ.ide-apex S.mkArr-in-hom [of S.set x Hom.set (a', a)]
      Hom.set-subset-Univ
  by simp
have YoD-a'.cones-map ?f (YoD.at a' (map o χ)) = σ
proof (intro NaturalTransformation.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)
```

show *S.par* (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?f*) (σ *j*)
  using *1 f j x YoD-*a'*.preserves-hom* **by** *fastforce*
show *S.Fun* (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?f*) = *S.Fun* (σ *j*)
proof
  fix *e*
  have $e \notin$ *S.set x* $\Longrightarrow$ *S.Fun* (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?f*) *e* = *S.Fun* (σ *j*) *e*
  proof –
    assume *e*: $e \notin$ *S.set x*
    have *S.Fun* (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?f*) *e* = *undefined*
      using *1 e f j x S.Fun-mapsto* **by** *fastforce*
    also have ... = *S.Fun* (σ *j*) *e*
    proof –
      have «σ *j* : *x* $\rightarrow_S$ *YoD.at* a' (*map* $\circ$ *D*) (*J.cod j*)»
        using *j* σ*.A.map-simp* **by** *force*
      thus *?thesis*
        using *e j S.Fun-mapsto* [*of* σ *j*] *extensional-arb* [*of S.Fun* (σ *j*)]
        by *fastforce*
    qed
    finally show *?thesis* **by** *auto*
  qed
  moreover have $e \in$ *S.set x* $\Longrightarrow$
                  S.Fun (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?f*) *e* = *S.Fun* (σ *j*) *e*
  proof –
    assume *e*: $e \in$ *S.set x*
    interpret κ*e*: *transformation-by-components J C* A'*.map D*
                  ‹λ*j*. ψ (a', *D j*) (*S.Fun* (σ *j*) *e*)›
      using *e* κ **by** *blast*
    interpret κ*e*: *cone J C D* a' ‹*?*κ *e*› **using** *e cone-*κ*e* **by** *simp*
    have *induced-arrow*: χ*.induced-arrow* a' (*?*κ *e*) $\in$ *C.hom* a' *a*
      using *a* a' *e ex-fe* χ*.induced-arrowI* κ*e.cone-axioms* **by** *simp*
    have *S.Fun* (*YoD.at* a' (*map o* χ) *j* $\cdot_S$ *?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* κ*e.cone-axioms* **by** *fastforce*
    also have ... = φ (a', *D j*) (ψ (a', *D j*) (*S.Fun* (σ *j*) *e*))
      using *j* κ*e.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
  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)
    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' x S.Fun-mapsto extensional-arb by fastforce
      moreover have e ∈ S.set x ⟹ S.Fun f' e = S.Fun ?f e
      proof -
        assume e: e ∈ S.set x
        have 1: «ψ (a', a) (S.Fun f' e) : a' → a»
        proof -
          have S.Fun f' e ∈ S.Cod f'
            using a a' e f' S.Fun-mapsto by auto
          hence S.Fun f' e ∈ Hom.set (a', a)
            using a a' f' Hom.set-map by auto
          thus ?thesis
            using a a' e f' S.Fun-mapsto Hom.ψ-mapsto Hom.set-map by blast
        qed
        have 2: «ψ (a', a) (S.Fun ?f e) : a' → a»
        proof -
```

have $S.Fun\ ?f\ e \in S.Cod\ ?f$
using $a\ a'\ e\ f\ S.Fun\text{-}mapsto$ **by** *force*
hence $S.Fun\ ?f\ e \in Hom.set\ (a',\ a)$
using $a\ a'\ f\ Hom.set\text{-}map$ **by** *auto*
thus *?thesis*
using $a\ a'\ e\ f'\ S.Fun\text{-}mapsto\ Hom.\psi\text{-}mapsto\ Hom.set\text{-}map$ **by** *blast*
qed
interpret χofe: *cone* $J\ C\ D\ a'\ \langle D.cones\text{-}map\ (\psi\ (a',\ a)\ (S.Fun\ ?f\ e))\ \chi\rangle$
proof –
have $D.cones\text{-}map\ (\psi\ (a',\ a)\ (S.Fun\ ?f\ e)) \in D.cones\ a \to D.cones\ a'$
using $2\ D.cones\text{-}map\text{-}mapsto\ [of\ \psi\ (a',\ a)\ (S.Fun\ ?f\ e)]$
by $(elim\ C.in\text{-}homE,\ auto)$
thus *cone* $J\ C\ D\ a'\ (D.cones\text{-}map\ (\psi\ (a',\ a)\ (S.Fun\ ?f\ e))\ \chi)$
using $\chi.cone\text{-}axioms$ **by** *blast*
qed
have $f'e$: $S.Fun\ f'\ e \in Hom.set\ (a',\ a)$
using $a\ a'\ e\ f'\ x\ S.Fun\text{-}mapsto\ [of\ f']\ Hom.set\text{-}map$ **by** *fastforce*
have fe: $S.Fun\ ?f\ e \in Hom.set\ (a',\ a)$
using $e\ f$ **by** $(elim\ S.in\text{-}homE,\ auto)$
have A: $\bigwedge h\ j.\ h \in C.hom\ a'\ a \Longrightarrow J.arr\ j \Longrightarrow$
$S.Fun\ (YoD.at\ a'\ (map\ o\ \chi)\ j)\ (\varphi\ (a',\ a)\ h)$
$= \varphi\ (a',\ D\ (J.cod\ j))\ (\chi\ j \cdot h)$
proof –
fix $h\ j$
assume j: $J.arr\ j$
assume h: $h \in C.hom\ a'\ a$
have $S.Fun\ (YoD.at\ a'\ (map\ o\ \chi)\ j) = S.Fun\ (Y\ (\chi\ j)\ a')$
using $a'\ j\ YoD.at\text{-}simp\ Y\text{-}def\ Yo\chi.preserves\text{-}reflects\text{-}arr\ [of\ j]$
by *simp*
also have ... $= restrict\ (\varphi\ (a',\ D\ (J.cod\ j)) \circ C\ (\chi\ j) \circ \psi\ (a',\ a))$
$(Hom.set\ (a',\ a))$
proof –
have $S.arr\ (Y\ (\chi\ j)\ a') \wedge$
$Y\ (\chi\ j)\ a' = S.mkArr\ (Hom.set\ (a',\ a))\ (Hom.set\ (a',\ D\ (J.cod\ j)))$
$(\varphi\ (a',\ D\ (J.cod\ j)) \circ C\ (\chi\ j) \circ \psi\ (a',\ a))$
using $a'\ j\ \chi.preserves\text{-}hom\ [of\ j\ J.dom\ j\ J.cod\ j]$
$Y\text{-}arr\text{-}ide\ [of\ a'\ \chi\ j\ a\ D\ (J.cod\ j)]\ \chi.A.map\text{-}simp$
by *auto*
thus *?thesis*
using $S.Fun\text{-}mkArr$ **by** *metis*
qed
finally have $S.Fun\ (YoD.at\ a'\ (map\ o\ \chi)\ j)$
$= restrict\ (\varphi\ (a',\ D\ (J.cod\ j)) \circ C\ (\chi\ j) \circ \psi\ (a',\ a))$
$(Hom.set\ (a',\ a))$
by *auto*
hence $S.Fun\ (YoD.at\ a'\ (map\ o\ \chi)\ j)\ (\varphi\ (a',\ a)\ h)$
$= (\varphi\ (a',\ D\ (J.cod\ j)) \circ C\ (\chi\ j) \circ \psi\ (a',\ a))\ (\varphi\ (a',\ a)\ h)$
using $a\ a'\ h\ Hom.\varphi\text{-}mapsto$ **by** *auto*
also have ... $= \varphi\ (a',\ D\ (J.cod\ j))\ (\chi\ j \cdot h)$

```
    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 3: 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
    have 4: 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
    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*
ultimately show *?thesis*
using *a a′ 3 4 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*
ultimately show *?thesis*
using *a a′ 3 4 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
qed
ultimately have ∃!*f*. ≪*f* : *x* $\rightarrow_S$ *Hom.map* (*a′*, *a*)≫ ∧
YoD-a′.cones-map f (*YoD.at a′* (*map o* χ)) = σ
using *ex1I* [*of* λ*f*. *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* $\rightarrow_S$ *Cop-S.Map* (*map a*) *a′*≫ ∧
YoD-a′.cones-map f (*YoD.at a′* (*map o* χ)) = σ

using *a a′ Y-def* [*of a*] **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 19

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** $\cdot_C$ *55*)
and $Arr :: 'a \Rightarrow bool$ +
assumes *inclusion*: $Arr\ f \Longrightarrow C.arr\ f$
and *dom-closed*: $Arr\ f \Longrightarrow Arr\ (C.dom\ f)$
and *cod-closed*: $Arr\ f \Longrightarrow Arr\ (C.cod\ f)$
and *comp-closed*: $[\![\ Arr\ f;\ Arr\ g;\ C.cod\ f = C.dom\ g\ ]\!] \Longrightarrow Arr\ (g \cdot_C f)$
begin

no-notation *C.in-hom* $(\langle\!\langle \text{-} : \text{-} \rightarrow \text{-} \rangle\!\rangle)$
notation *C.in-hom* $(\langle\!\langle \text{-} : \text{-} \rightarrow_C \text{-} \rangle\!\rangle)$

definition *comp* (**infixr** $\cdot$ *55*)
where $g \cdot f = (if\ Arr\ f \wedge Arr\ g \wedge C.cod\ f = C.dom\ g\ then\ g \cdot_C f\ else\ C.null)$

interpretation *partial-magma comp*
proof
show $\exists!n.\ \forall f.\ n \cdot f = n \wedge f \cdot n = n$
proof
show *1*: $\forall f.\ C.null \cdot f = C.null \wedge f \cdot C.null = C.null$
by (*metis C.comp-null*(*1*) *C.ex-un-null comp-def*)
show $\bigwedge n.\ \forall f.\ n \cdot f = n \wedge f \cdot n = n \Longrightarrow n = C.null$
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.comp-null*(*1*) *C.ex-un-null comp-def*)
  thus *?thesis* **using** *ex-un-null* **by** (*metis comp-null*(*2*))
qed

lemma *ideI*:
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
      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
      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*:
shows *arr f* $\longleftrightarrow$ *Arr f*
proof
show *Arr f* $\Longrightarrow$ *arr f*
using *arr-def comp-def Arr-iff-dom-in-domain Arr-iff-cod-in-codomain* **by** *auto*
show *arr f* $\Longrightarrow$ *Arr f*
proof –
assume *f*: *arr f*
obtain *a* **where** *a*: $a \in$ *domains f* $\vee$ $a \in$ *codomains f*
using *f arr-def* **by** *auto*
have $f \cdot a \neq C.null \vee a \cdot f \neq C.null$
using *a domains-def codomains-def null-char* **by** *auto*
thus *Arr f*
using *comp-def* **by** *metis*
qed
qed

lemma *arrI* [*intro*]:
assumes *Arr f*
shows *arr f*
using *assms arr-char* **by** *simp*

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

interpretation *category comp*
using *comp-def null-char inclusion comp-closed dom-closed cod-closed*
apply *unfold-locales*
apply *auto*[*1*]
apply (*metis Arr-iff-dom-in-domain Arr-iff-cod-in-codomain arr-char arr-def emptyE*)
proof –
fix *f g h*
assume *gf*: *seq g f* **and** *hg*: *seq h g*
show *1*: *seq* $(h \cdot g)$ *f*
using *gf hg inclusion comp-closed comp-def* **by** *auto*
show $(h \cdot g) \cdot f = h \cdot g \cdot f$
using *gf hg 1 C.not-arr-null inclusion comp-def arr-char*
by (*metis* (*full-types*) *C.cod-comp C.comp-assoc*)
next
fix *f g h*
assume *hg*: *seq h g* **and** *hgf*: *seq* $(h \cdot g)$ *f*
show *seq g f*
using *hg hgf comp-def null-char inclusion arr-char comp-closed*

by (*metis* (*full-types*) *C.dom-comp*)
next
fix *f g h*
assume *hgf*: *seq h* (*g* · *f*) **and** *gf*: *seq g f*
show *seq h g*
using *hgf gf comp-def null-char arr-char comp-closed*
by (*metis C.seqE C.ext C.match-2*)
qed

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

notation *in-hom* (≪- : - → -≫)

lemma *dom-simp* [*simp*]:
assumes *arr f*
shows *dom f* = *C.dom f*
proof −
have *ide* (*C.dom f*)
using *assms ideI*
by (*meson C.ide-dom arr-char dom-closed inclusion*)
moreover have *f* · *C.dom f* ≠ *null*
using *assms inclusion comp-def null-char dom-closed not-arr-null C.comp-arr-dom arr-char*
by *auto*
ultimately show *?thesis*
using *dom-eqI ext* **by** *blast*
qed

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

lemma *cod-simp* [*simp*]:
assumes *arr f*
shows *cod f* = *C.cod f*
proof −
have *ide* (*C.cod f*)
using *assms ideI*
by (*meson C.ide-cod arr-char cod-closed inclusion*)
moreover have *C.cod f* · *f* ≠ *null*
using *assms inclusion comp-def null-char cod-closed not-arr-null C.comp-cod-arr arr-char*
by *auto*
ultimately show *?thesis*
using *cod-eqI ext* **by** *blast*
qed

lemma *cod-char*:
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*:
shows $\ll f : a \rightarrow b\gg \longleftrightarrow arr\ a \wedge arr\ b \wedge arr\ f \wedge \ll f : a \rightarrow_C b\gg$
  using *inclusion arr-char 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*:
shows *ide a* $\longleftrightarrow$ *arr a* $\wedge$ *C.ide a*
  using *ide-in-hom C.ide-in-hom in-hom-char* **by** *simp*

lemma *seq-char*:
shows *seq g f* $\longleftrightarrow$ *arr f* $\wedge$ *arr g* $\wedge$ *C.seq g f*
proof
  show *arr f* $\wedge$ *arr g* $\wedge$ *C.seq g f* $\Longrightarrow$ *seq g f*
    using *arr-char dom-char cod-char* **by** (*intro seqI*, *auto*)
  show *seq g f* $\Longrightarrow$ *arr f* $\wedge$ *arr g* $\wedge$ *C.seq g f*
    apply (*elim seqE*, *auto*)
    using *inclusion arr-char* **by** *auto*
qed

lemma *hom-char*:
shows *hom a b* = *C.hom a b* $\cap$ *Collect Arr*
proof
  show *hom a b* $\subseteq$ *C.hom a b* $\cap$ *Collect Arr*
    using *in-hom-char* **by** *auto*
  show *C.hom a b* $\cap$ *Collect Arr* $\subseteq$ *hom a b*
    using *arr-char dom-char cod-char* **by** *force*
qed

lemma *comp-char*:
shows $g \cdot f$ = (*if arr f* $\wedge$ *arr g* $\wedge$ *C.seq g f then* $g \cdot_C f$ *else C.null*)
  using *arr-char 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 seq-char* **by** *metis*

lemma *inclusion-preserves-inverse*:
assumes *inverse-arrows f g*
shows *C.inverse-arrows f g*
  using *assms ide-char comp-simp arr-char*
  by (*intro C.inverse-arrowsI*, *auto*)

lemma *iso-char*:
shows *iso f* $\longleftrightarrow$ *C.iso f* $\wedge$ *arr f* $\wedge$ *arr* (*C.inv f*)
proof
  assume *f*: *iso f*
  show *C.iso f* $\wedge$ *arr f* $\wedge$ *arr* (*C.inv f*)

```
  proof -
    have 1: inverse-arrows f (inv f)
      using f inv-is-inverse by auto
    have 2: C.inverse-arrows f (inv f)
      using 1 inclusion-preserves-inverse by blast
    moreover have arr (inv f)
      using 1 iso-is-arr iso-inv-iso by blast
    moreover have inv f = C.inv f
      using 1 2 C.inv-is-inverse C.inverse-arrow-unique by blast
    ultimately show ?thesis using f 2 iso-is-arr by auto
  qed
  next
  assume f: C.iso f ∧ arr f ∧ arr (C.inv f)
  show iso f
  proof
    have 1: C.inverse-arrows f (C.inv f)
      using f C.inv-is-inverse by blast
    show inverse-arrows f (C.inv f)
    proof
      have 2: C.inv f · f = C.inv f ·C f ∧ f · C.inv f = f ·C C.inv f
        using f 1 comp-char by fastforce
      have 3: antipar f (C.inv f)
        using f C.seqE seqI by simp
      show ide (C.inv f · f)
        using 1 2 3 ide-char apply (elim C.inverse-arrowsE) by simp
      show ide (f · C.inv f)
        using 1 2 3 ide-char apply (elim C.inverse-arrowsE) by simp
    qed
  qed
qed

lemma inv-char:
assumes iso f
shows inv f = C.inv f
proof -
  have C.inverse-arrows f (inv f)
  proof
    have 1: inverse-arrows f (inv f)
      using assms iso-char inv-is-inverse by blast
    show C.ide (inv f ·C f)
    proof -
      have inv f · f = inv f ·C f
        using assms 1 inv-in-hom iso-char [of f] comp-char [of inv f f] seq-char by auto
      thus ?thesis
        using 1 ide-char arr-char by force
    qed
    show C.ide (f ·C inv f)
    proof -
      have f · inv f = f ·C inv f
```

using *assms 1 inv-in-hom iso-char* [*of f*] *comp-char* [*of f inv f*] *seq-char* **by** *auto*
thus *?thesis*
using *1 ide-char arr-char* **by** *force*
qed
qed
thus *?thesis* **using** *C.inverse-arrow-unique C.inv-is-inverse* **by** *blast*
qed

end

sublocale *subcategory* ⊆ *category comp*
using *is-category* **by** *auto*

19.1 Full Subcategory

locale *full-subcategory* =
C: *category C*
for *C* :: *'a comp*
and *Ide* :: *'a* ⇒ *bool* +
assumes *inclusion*: *Ide f* ⟹ *C.ide f*

sublocale *full-subcategory* ⊆ *subcategory C* λ*f*. *C.arr f* ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*)
by (*unfold-locales*; *simp*)

context *full-subcategory*
begin

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

lemma *iso-char*:
shows *iso f* ⟷ *arr f* ∧ *C.iso f*
proof
assume *f*: *iso f*
obtain *g* **where** *g*: *inverse-arrows f g* **using** *f* **by** *blast*
show *arr f* ∧ *C.iso f*
proof −
have *C.inverse-arrows f g*
using *g* **apply** (*elim inverse-arrowsE*, *intro C.inverse-arrowsI*)
using *comp-simp ide-char arr-char* **by** *auto*
thus *?thesis*
using *f iso-is-arr* **by** *blast*
qed
next
assume *f*: *arr f* ∧ *C.iso f*
obtain *g* **where** *g*: *C.inverse-arrows f g* **using** *f* **by** *blast*
have *inverse-arrows f g*
proof
show *ide* (*comp g f*)
using *f g*
by (*metis* (*no-types*, *lifting*) *C.seqE C.ide-compE C.inverse-arrowsE*

```
        arr-char dom-simp ide-dom comp-def)
    show ide (comp f g)
      using f g C.inverse-arrows-sym [of f g]
      by (metis (no-types, lifting) C.seqE C.ide-compE C.inverse-arrowsE
          arr-char dom-simp ide-dom comp-def)
  qed
  thus iso f by auto
qed

end
```

19.2 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 ⇒ bool
begin

  interpretation functor S.comp C S.map
    using S.map-def S.arr-char S.inclusion S.dom-char S.cod-char
          S.dom-closed S.cod-closed S.comp-closed S.arr-char 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 ⊆ faithful-functor S.comp C S.map
  using is-faithful-functor by auto
```

sublocale *inclusion-functor* ⊆ *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* ⇒ *bool*

sublocale *full-inclusion-functor* ⊆
inclusion-functor C λf. C.arr f ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*) **..**

context *full-inclusion-functor*
begin

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 S.in-hom-char S.map-simp*)

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

end

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

end

Chapter 20

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*: ... $= (f \cdot_C \varepsilon\ (C.dom\ f)) \cdot_C C.inv\ (\varepsilon\ (C.dom\ f))$
    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 $\varepsilon\ (C.cod\ f) \cdot_C F\ (G\ f) \cdot_C C.inv\ (\varepsilon\ (C.dom\ f)) = f$ **by** *blast*
  show $C.inv\ (\varepsilon\ (C.cod\ f)) \cdot_C f \cdot_C \varepsilon\ (C.dom\ f) = F\ (G\ f)$
    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 $C.inv\ (\varepsilon\ (C.cod\ f)) \in C.hom\ (C.cod\ f)\ (F\ (G\ (C.cod\ f))) \wedge$
        $C.iso\ (C.inv\ (\varepsilon\ (C.cod\ f)))$
      using *par C.iso-inv-iso* **by** *auto*
    moreover have *1*: $\varepsilon\ (C.dom\ f) \in C.hom\ (F\ (G\ (C.dom\ f)))\ (C.dom\ f) \wedge$
                $C.iso\ (\varepsilon\ (C.dom\ f))$
      using *par* **by** *auto*
    ultimately have *2*: $f \cdot_C \varepsilon\ (C.dom\ f) = f' \cdot_C \varepsilon\ (C.dom\ f)$
      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 $C.epi\ (\varepsilon\ (C.dom\ f))$
        using *1 par C.iso-is-retraction C.retraction-is-epi* **by** *blast*
      thus *?thesis* **using** *2 par* **by** *auto*
    qed
  qed
qed

lemma *G-is-essentially-surjective*:
shows *essentially-surjective-functor C D G*
proof
  fix b
  assume *b*: $D.ide\ b$
  have $C.ide\ (F\ b) \wedge D.isomorphic\ (G\ (F\ b))\ b$
  proof
    show $C.ide\ (F\ b)$ **using** *b* **by** *simp*
    show $D.isomorphic\ (G\ (F\ b))\ b$
    proof (*unfold D.isomorphic-def*)
      have $\ll D.inv\ (\eta\ b) : G\ (F\ b) \rightarrow_D b\gg \wedge D.iso\ (D.inv\ (\eta\ b))$
        using *b D.iso-inv-iso* **by** *auto*
      thus $\exists f.\ \ll f : G\ (F\ b) \rightarrow_D b\gg \wedge D.iso\ f$ **by** *blast*

    qed
  qed
  thus $\exists\, a.\ C.ide\ a \wedge D.isomorphic\ (G\ a)\ b$
    by *blast*
qed

interpretation ε*-inv*: *inverse-transformation* $C\ C\ \langle F\ o\ G\rangle\ C.map\ \varepsilon$ **..**
interpretation η*-inv*: *inverse-transformation* $D\ D\ D.map\ \langle G\ o\ F\rangle\ \eta$ **..**
interpretation *GF*: *equivalence-of-categories* $D\ C\ G\ F\ \varepsilon$*-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: $\ll g : G\ a \rightarrow_D G\ a'\gg$
  show $\exists f.\ \ll f : a \rightarrow_C a'\gg \wedge G\ f = g$
  proof
    have εa: $C.inverse\text{-}arrows\ (\varepsilon\ a)\ (C.inv\ (\varepsilon\ a))$
      using *a C.inv-is-inverse* **by** *auto*
    have $\varepsilon a'$: $C.inverse\text{-}arrows\ (\varepsilon\ a')\ (C.inv\ (\varepsilon\ a'))$
      using a' *C.inv-is-inverse* **by** *auto*
    let $?f = \varepsilon\ a' \cdot_C F\ g \cdot_C C.inv\ (\varepsilon\ a)$
    have f: $\ll ?f : a \rightarrow_C a'\gg$
      using $a\ a'\ g\ \varepsilon a\ \varepsilon a'\ \varepsilon$*.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 $F\ (G\ ?f) = C.inv\ (\varepsilon\ a') \cdot_C ?f \cdot_C \varepsilon\ a$
          using *f C-arr-expansion(2)* [*of ?f*] **by** *auto*
        also have ... $= (C.inv\ (\varepsilon\ a') \cdot_C \varepsilon\ a') \cdot_C F\ g \cdot_C C.inv\ (\varepsilon\ a) \cdot_C \varepsilon\ a$
          using $a\ a'\ f\ g$ *C.comp-assoc* **by** *fastforce*
        also have ... $= F\ g$
          using $a\ a'\ g\ \varepsilon a\ \varepsilon a'$ *C.comp-inv-arr C.comp-arr-dom C.comp-cod-arr* **by** *auto*
        finally show *?thesis* **by** *blast*
      qed
      moreover have $D.par\ (G\ (\varepsilon\ a' \cdot_C F\ g \cdot_C C.inv\ (\varepsilon\ a)))\ g$

using *f g* **by** *fastforce*
ultimately show *?thesis* **using** *f g F.is-faithful* **by** *blast*
qed
ultimately show $\ll ?f : a \rightarrow_C a' \gg \wedge G\ ?f = g$ **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** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and *G* :: $'c \Rightarrow 'd$ +
assumes *induces-equivalence*: $\exists F\ \eta\ \varepsilon.$ *equivalence-of-categories C D F G* $\eta\ \varepsilon$
begin

notation *C.in-hom* $(\ll - : - \rightarrow_C - \gg)$
notation *D.in-hom* $(\ll - : - \rightarrow_D - \gg)$

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* ⊆ *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 ⊆ 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
```

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

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* $D\ D\ D.map\ G\ o\ F\ \eta$ +
ε: *natural-isomorphism* $C\ C\ F\ o\ G\ C.map\ \varepsilon$
for C :: $'c$ *comp* (**infixr** $\cdot_C$ *55*)
and D :: $'d$ *comp* (**infixr** $\cdot_D$ *55*)
and F :: $'d \Rightarrow 'c$
and G :: $'c \Rightarrow 'd$
and η :: $'d \Rightarrow 'd$
and ε :: $'c \Rightarrow 'c$

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 $D.inverse\text{-}arrows\ (\eta\ (G\ a))\ (G\ (\varepsilon\ a))$
proof
show $D.ide\ (G\ (\varepsilon\ a) \cdot_D \eta\ (G\ a))$
using *assms triangle-G* $G\varepsilon o\eta G.map\text{-}simp\text{-}ide$ **by** *fastforce*
thus $D.ide\ (\eta\ (G\ a) \cdot_D G\ (\varepsilon\ a))$
using *assms* $D.section\text{-}retraction\text{-}of\text{-}iso$ [*of* $G\ (\varepsilon\ a)\ \eta\ (G\ a)$] **by** *auto*
qed

lemma *triangle-F'*:
assumes $D.ide\ b$
shows $C.inverse\text{-}arrows\ (F\ (\eta\ b))\ (\varepsilon\ (F\ b))$
proof
show $C.ide\ (\varepsilon\ (F\ b) \cdot_C F\ (\eta\ b))$
using *assms triangle-F* $\varepsilon FoF\eta.map\text{-}simp\text{-}ide$ **by** *auto*
thus $C.ide\ (F\ (\eta\ b) \cdot_C \varepsilon\ (F\ b))$
using *assms* $C.section\text{-}retraction\text{-}of\text{-}iso$ [*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-equivalence*:
assumes *adjoint-equivalence* $C\ D\ F\ G\ \eta\ \varepsilon$
shows *adjoint-equivalence* $D\ C\ G\ F\ (inverse\text{-}transformation.map\ C\ C\ (C.map)\ \varepsilon)$

```
                          (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.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.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.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.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 NaturalTransformation.eqI)
      show natural-transformation C D G G G
        using G.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'
                η.components-are-iso ε.components-are-iso 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 NaturalTransformation.eqI)
      show natural-transformation D C F F F
        using F.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.components-are-iso ε.components-are-iso G.preserves-ide
              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 D C F +
  essentially-surjective-functor D C F
  for C :: 'c comp      (infixr ·C 55)
  and D :: 'd comp      (infixr ·D 55)
  and F :: 'd ⇒ 'c
begin

  notation C.in-hom      («- : - →C -»)
  notation D.in-hom      («- : - →D -»)

  lemma is-left-adjoint-functor:
  shows left-adjoint-functor D C F
```

proof
  fix y
  assume y: $C.ide\ y$
  let $?x = SOME\ x.\ D.ide\ x \wedge (\exists\, e.\ C.iso\ e \wedge \ll e : F\ x \rightarrow_C y \gg)$
  let $?e = SOME\ e.\ C.iso\ e \wedge \ll e : F\ ?x \rightarrow_C y \gg$
  have $\exists\, x\ e.\ C.iso\ e \wedge$ *terminal-arrow-from-functor* $D\ C\ F\ x\ y\ e$
  proof −
    have $\exists\, x.\ C.iso\ ?e \wedge$ *terminal-arrow-from-functor* $D\ C\ F\ x\ y\ ?e$
    proof −
      have x: $D.ide\ ?x \wedge (\exists\, e.\ C.iso\ e \wedge \ll e : F\ ?x \rightarrow_C y \gg)$
      proof −
        obtain x **where** x: $D.ide\ x \wedge C.isomorphic\ (F\ x)\ y$
          using *y essentially-surjective D.isomorphic-def* **by** *blast*
        obtain e **where** e: $C.iso\ e \wedge \ll e : F\ x \rightarrow_C y \gg$
          using *y x* **by** *auto*
        hence $\exists\, x.\ D.ide\ x \wedge (\exists\, e.\ C.iso\ e \wedge \ll e : F\ x \rightarrow_C y \gg)$
          using *x* **by** *auto*
        thus $D.ide\ ?x \wedge (\exists\, e.\ C.iso\ e \wedge \ll e : F\ ?x \rightarrow_C y \gg)$
          using *someI-ex* $[of\ \lambda x.\ D.ide\ x \wedge (\exists\, e.\ C.iso\ e \wedge \ll e : F\ x \rightarrow_C y \gg)]$ **by** *blast*
      qed
      hence e: $C.iso\ ?e \wedge \ll ?e : F\ ?x \rightarrow_C y \gg$
        using *someI-ex* $[of\ \lambda e.\ C.iso\ e \wedge \ll e : F\ ?x \rightarrow_C y \gg]$ **by** *blast*
      interpret *arrow-from-functor* $D\ C\ F\ ?x\ y\ ?e$
        using *x e* **by** (*unfold-locales*, *simp*)
      interpret *terminal-arrow-from-functor* $D\ C\ F\ ?x\ y\ ?e$
      proof
        fix $x'\ f$
        assume *1*: *arrow-from-functor* $D\ C\ F\ x'\ y\ f$
        interpret f: *arrow-from-functor* $D\ C\ F\ x'\ y\ f$
          using *1* **by** *simp*
        have f: $\ll f: F\ x' \rightarrow_C y \gg$
          by (*meson f.arrow*)
        show $\exists !g.$ *is-coext* $x'\ f\ g$
        proof
          let $?g = SOME\ g.\ \ll g : x' \rightarrow_D ?x \gg \wedge F\ g = C.inv\ ?e \cdot_C f$
          have g: $\ll ?g : x' \rightarrow_D ?x \gg \wedge F\ ?g = C.inv\ ?e \cdot_C f$
          proof −
            have $\exists\, g.\ \ll g : x' \rightarrow_D ?x \gg \wedge F\ g = C.inv\ ?e \cdot_C f$
              using *f e x f.arrow*
              by (*meson C.comp-in-homI C.inv-in-hom is-full*)
            thus *?thesis*
              using *someI-ex* $[of\ \lambda g.\ \ll g : x' \rightarrow_D ?x \gg \wedge F\ g = C.inv\ ?e \cdot_C f]$ **by** *blast*
          qed
          show *1*: *is-coext* $x'\ f\ ?g$
          proof −
            have $\ll ?g : x' \rightarrow_D ?x \gg$
              using *g* **by** *simp*
            moreover have $?e \cdot_C F\ ?g = f$
            proof −

```
            have ?e ·C F ?g = ?e ·C C.inv ?e ·C f
              using g by simp
            also have ... = (?e ·C C.inv ?e) ·C f
              using e f C.inv-in-hom by (metis C.comp-assoc)
            also have ... = f
            proof −
              have ?e ·C C.inv ?e = y
                using e C.comp-arr-inv [of ?e] C.inv-is-inverse by auto
              thus ?thesis
                using f C.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' →D ?x» ∧ ?e ·C F g' = f using g' is-coext-def by simp
          have 3: «?g : x' →D ?x» ∧ ?e ·C F ?g = f using 1 is-coext-def by simp
          have F g' = F ?g
            using e 2 3 C.iso-is-section C.section-is-mono C.monoE by blast
          moreover have D.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 D C F x y e by blast
qed

lemma is-equivalence-functor:
shows equivalence-functor D C F
proof
  interpret left-adjoint-functor D C F
    using is-left-adjoint-functor by blast
  interpret equivalence-of-categories C D F G η ε
  proof
    show 1: ⋀a. C.ide a ⟹ C.iso (ε a)
    proof −
      fix a
```

assume *a*: *C.ide a*
interpret *εa*: *terminal-arrow-from-functor D C F* ‹*G a*› *a* ‹*ε a*›
  using *a φψ.has-terminal-arrows-from-functor* [*of a*] **by** *blast*
have *C.retraction* (*ε a*)
proof −
  obtain *b φ* **where** *φ*: *D.ide b* ∧ *C.iso φ* ∧ «*φ*: *F b* $\rightarrow_C$ *a*»
    using *a essentially-surjective* **by** *blast*
  interpret *φ*: *arrow-from-functor D C F b a φ*
    using *φ* **by** (*unfold-locales*, *simp*)
  let *?g* = *εa.the-coext b φ*
  have *1*: «*?g* : *b* $\rightarrow_D$ *G a*» ∧ *ε a* $\cdot_C$ *F ?g* = *φ*
    using *φ.arrow-from-functor-axioms εa.the-coext-prop* [*of b φ*] **by** *simp*
  have *a* = (*ε a* $\cdot_C$ *F ?g*) $\cdot_C$ *C.inv φ*
    using *a 1 φ C.comp-cod-arr ε.preserves-hom* [*of a a a*]
        C.invert-side-of-triangle(*2*) [*of ε a* $\cdot_C$ *F ?g a φ*]
     by *auto*
  also have ... = *ε a* $\cdot_C$ *F ?g* $\cdot_C$ *C.inv φ*
  proof −
    have *C.seq* (*ε a*) (*F ?g*)
      using *a 1 ε.preserves-hom* [*of a a a*] **by** *fastforce*
    moreover have *C.seq* (*F ?g*) (*C.inv φ*)
      using *a 1 φ C.inv-in-hom* [*of φ F b a*] **by** *blast*
    ultimately show *?thesis* **using** *C.comp-assoc* **by** *auto*
  qed
  finally have ∃*f*. *C.ide* (*ε a* $\cdot_C$ *f*)
    using *a* **by** *metis*
  thus *?thesis*
    unfolding *C.retraction-def* **by** *blast*
qed
moreover have *C.mono* (*ε a*)
proof
  show *C.arr* (*ε a*)
    using *a* **by** *simp*
  show ⋀*f f′*. *C.seq* (*ε a*) *f* ∧ *C.seq* (*ε a*) *f′* ∧ *ε a* $\cdot_C$ *f* = *ε a* $\cdot_C$ *f′* ⟹ *f* = *f′*
  proof −
    fix *f f′*
    assume *ff′*: *C.seq* (*ε a*) *f* ∧ *C.seq* (*ε a*) *f′* ∧ *ε a* $\cdot_C$ *f* = *ε a* $\cdot_C$ *f′*
    have *f*: «*f* : *C.dom f* $\rightarrow_C$ *F* (*G a*)»
      using *a ff′ ε.preserves-hom* [*of a a a*] **by** *fastforce*
    have *f′*: «*f′* : *C.dom f′* $\rightarrow_C$ *F* (*G a*)»
      using *a ff′ ε.preserves-hom* [*of a a a*] **by** *fastforce*
    have *par*: *C.par f f′*
      using *f f′ ff′ C.dom-comp* [*of ε a f*] **by** *force*
    obtain *b′ φ* **where** *φ*: *D.ide b′* ∧ *C.iso φ* ∧ «*φ*: *F b′* $\rightarrow_C$ *C.dom f*»
      using *par essentially-surjective C.ide-dom* [*of f*] **by** *blast*
    have *1*: *ε a* $\cdot_C$ *f* $\cdot_C$ *φ* = *ε a* $\cdot_C$ *f′* $\cdot_C$ *φ*
    proof −
      have *ε a* $\cdot_C$ *f* $\cdot_C$ *φ* = (*ε a* $\cdot_C$ *f*) $\cdot_C$ *φ*
      proof −

have $C.seq\ f\ \varphi$ **using** *par* φ **by** *auto*
moreover have $C.seq\ (\varepsilon\ a)\ f$ **using** *ff'* **by** *blast*
ultimately show *?thesis* **using** *C.comp-assoc* **by** *auto*
qed
also have ... $= (\varepsilon\ a \cdot_C f') \cdot_C \varphi$
using *ff'* **by** *argo*
also have ... $= \varepsilon\ a \cdot_C f' \cdot_C \varphi$
proof −
have $C.seq\ f'\ \varphi$ **using** *par* φ **by** *auto*
moreover have $C.seq\ (\varepsilon\ a)\ f'$ **using** *ff'* **by** *blast*
ultimately show *?thesis* **using** *C.comp-assoc* **by** *auto*
qed
finally show *?thesis* **by** *blast*
qed
obtain g **where** g: $\ll g : b' \rightarrow_D G\ a\gg \wedge F\ g = f \cdot_C \varphi$
using $a\ f\ \varphi$ *is-full* [*of* $G\ a\ b'\ f \cdot_C \varphi$] **by** *auto*
obtain g' **where** g': $\ll g' : b' \rightarrow_D G\ a\gg \wedge F\ g' = f' \cdot_C \varphi$
using $a\ f'$ *par* φ *is-full* [*of* $G\ a\ b'\ f' \cdot_C \varphi$] **by** *auto*
interpret $f\varphi$: *arrow-from-functor* $D\ C\ F\ b'\ a$ ‹$\varepsilon\ a \cdot_C f \cdot_C \varphi$›
using $a\ \varphi\ f$ *ε.preserves-hom* [*of* $a\ a\ a$]
by (*unfold-locales*, *fastforce*)
interpret $f'\varphi$: *arrow-from-functor* $D\ C\ F\ b'\ a$ ‹$\varepsilon\ a \cdot_C f' \cdot_C \varphi$›
using $a\ \varphi\ f'$ *par* *ε.preserves-hom* [*of* $a\ a\ a$]
by (*unfold-locales*, *fastforce*)
have *εa.is-coext* b' $(\varepsilon\ a \cdot_C f \cdot_C \varphi)$ g
unfolding *εa.is-coext-def* **using** *g 1* **by** *auto*
moreover have *εa.is-coext* b' $(\varepsilon\ a \cdot_C f' \cdot_C \varphi)$ 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 [*of* $b'\ \varepsilon\ a \cdot_C f \cdot_C \varphi\ g$]
εa.the-coext-unique [*of* $b'\ \varepsilon\ a \cdot_C f' \cdot_C \varphi\ g'$]
by *auto*
hence $f \cdot_C \varphi = f' \cdot_C \varphi$
using $g\ g'$ *is-faithful* **by** *argo*
thus $f = f'$
using $\varphi\ f\ f'$ *par C.iso-is-retraction C.retraction-is-epi*
C.epiE [*of* $\varphi\ f\ f'$]
by *auto*
qed
qed
ultimately show $C.iso\ (\varepsilon\ a)$
using *C.iso-iff-mono-and-retraction* **by** *simp*
qed
interpret ε: *natural-isomorphism* $C\ C$ ‹$F\ o\ G$› *C.map* ε
using *1* **by** (*unfold-locales*, *auto*)
interpret εF: *natural-isomorphism* $D\ C$ ‹$F\ o\ G\ o\ F$› F ‹$\varepsilon\ o\ F$›
using *ε.components-are-iso* **by** (*unfold-locales*, *simp*)
show $\bigwedge a.\ D.ide\ a \Longrightarrow D.iso\ (\eta\ a)$

proof −
fix a
assume a: $D.ide\ a$
have 1: $C.iso\ ((\varepsilon\ o\ F)\ a)$
using a ε.*components-are-iso* **by** *simp*
moreover have $(\varepsilon\ o\ F)\ a \cdot_C (F\ o\ \eta)\ a = F\ a$
using a $\eta\varepsilon$.*triangle-F* $\varepsilon FoF\eta$.*map-simp-ide* **by** *simp*
ultimately have $C.inverse\text{-}arrows\ ((\varepsilon\ o\ F)\ a)\ ((F\ o\ \eta)\ a)$
using a C.*section-retraction-of-iso* **by** *simp*
hence $C.iso\ ((F\ o\ \eta)\ a)$
using C.*iso-inv-iso* **by** *blast*
thus $D.iso\ (\eta\ a)$
using a *reflects-iso* [*of* η a] **by** *fastforce*
qed
qed

interpret *adjoint-equivalence* $C\ D\ F\ G\ \eta\ \varepsilon$ **..**
interpret ε': *inverse-transformation* $C\ C\ \langle F\ o\ G\rangle\ C.map\ \varepsilon$ **..**
interpret η': *inverse-transformation* $D\ D\ D.map\ \langle G\ o\ F\rangle\ \eta$ **..**
interpret E: *adjoint-equivalence* $D\ C\ G\ F\ \varepsilon'.map\ \eta'.map$
using *adjoint-equivalence-axioms dual-equivalence* **by** *blast*
have *equivalence-of-categories* $D\ C\ G\ F\ \varepsilon'.map\ \eta'.map$ **..**
thus $\exists\, G\ \eta\ \varepsilon.$ *equivalence-of-categories* $D\ C\ G\ F\ \eta\ \varepsilon$ **by** *blast*
qed

end

sublocale *fully-faithful-and-essentially-surjective-functor* $\subseteq$ *equivalence-functor* $D\ C\ F$
using *is-equivalence-functor* **by** *blast*

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].