Category Theory with Adjunctions and Limits

Eugene W. Stark

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

September 13, 2023

Abstract

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

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

Contents

Chapter 1

Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Revision Notes

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

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

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

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

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

Chapter 2

Category

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

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

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

To represent the partiality of the composition operation of a category, we assume that the composition for a category has a unique zero element, which we call *null*, and we consider arrows to be "composable" if and only if their composite is non-null. Functors and natural transformations are required to map arrows to arrows and be "extensional" in the sense that they map non-arrows to null. This is so that equality of functors and natural transformations coincides with their extensional equality as functions in HOL. The fact that we co-opt an element of the arrow type to serve as *null* means that it is not possible to define a category whose arrows exhaust the elements of a given type. This presents a disadvantage in some situations. For example, we cannot construct a discrete category whose arrows are directly identified with the set of *all* elements of a given type *′a*; instead, we must pass to a larger type (such as *′a option*) so that there is an element available for use as *null*. The presence of *null*, however, is crucial to our being able to define a system of introduction and elimination rules that can be applied automatically to establish that a given expression denotes an arrow. Without *null*, we would be able to define an introduction rule to infer, say, that the composition of composable arrows is composable, but not an elimination rule to infer that arrows are composable from the fact that their composite is an arrow. Having the ability to do both is critical to the usability of the theory.

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

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

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

lemma *null-eqI*:
assumes $\bigwedge t.\ OP\ n\ t = n \wedge OP\ t\ n = n$
shows $n = null$
⟨*proof*⟩

lemma *null-is-zero* [*simp*]:
shows $OP\ null\ t = null$ **and** $OP\ t\ null = null$
⟨*proof*⟩

end

2.1 Partial Composition

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

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

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

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

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

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

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

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

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

lemma *domains-null*:
shows $domains\ null = \{\}$
⟨*proof*⟩

lemma *codomains-null*:
shows $codomains\ null = \{\}$
⟨*proof*⟩

lemma *self-domain-iff-ide*:
shows $a \in domains\ a \longleftrightarrow ide\ a$
⟨*proof*⟩

lemma *self-codomain-iff-ide*:
shows $a \in codomains\ a \longleftrightarrow ide\ a$
⟨*proof*⟩

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 $\neg\ arr\ null$
⟨*proof*⟩

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

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

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

lemma *arrI*:
assumes $«f : a \rightarrow b»$
shows *arr f*
⟨*proof*⟩

lemma *ide-in-hom* [*intro*]:
shows $ide\ a \longleftrightarrow «a : a \rightarrow a»$
⟨*proof*⟩

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 $f \cdot a = f$
⟨*proof*⟩

lemma *comp-ide-arr*:
assumes *ide b* **and** *seq b f*
shows $b \cdot f = f$
⟨*proof*⟩

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 $dom\ null = null$
⟨*proof*⟩

lemma *cod-null* [*simp*]:
shows $cod\ null = null$
⟨*proof*⟩

lemma *dom-in-domains*:
assumes $domains\ f \neq \{\}$
shows $dom\ f \in domains\ f$
⟨*proof*⟩

lemma *cod-in-codomains*:
assumes $codomains\ f \neq \{\}$

shows *cod f* ∈ *codomains f*
⟨*proof*⟩

end

2.2 Categories

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

locale *category* = *partial-composition* +
assumes *ext*: $g \cdot f \neq null \Longrightarrow seq\ g\ f$
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 $(h \cdot g) \cdot f = h \cdot g \cdot f$
⟨*proof*⟩

lemma *match-4*:
assumes *seq g f* **and** *seq h g*
shows $seq\ h\ (g \cdot f)$
⟨*proof*⟩

lemma *domains-comp*:
assumes *seq g f*
shows $domains\ (g \cdot f) = domains\ f$
⟨*proof*⟩

lemma *codomains-comp*:
assumes *seq g f*

shows *codomains* $(g \cdot f)$ = *codomains g*
⟨*proof*⟩

lemma *has-domain-iff-arr*:
shows *domains f* $\neq$ {} $\longleftrightarrow$ *arr f*
⟨*proof*⟩

lemma *has-codomain-iff-arr*:
shows *codomains f* $\neq$ {} $\longleftrightarrow$ *arr f*
⟨*proof*⟩

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*⟩

lemma *codomain-unique*:
assumes $b \in$ *codomains f* **and** $b' \in$ *codomains f*
shows $b = b'$
⟨*proof*⟩

lemma *domains-simp*:
assumes *arr f*
shows *domains f* = {*dom f*}
⟨*proof*⟩

lemma *codomains-simp*:
assumes *arr f*
shows *codomains f* = {*cod f*}
⟨*proof*⟩

lemma *domains-char*:
shows *domains f* = (*if arr f then* {*dom f*} *else* {})
⟨*proof*⟩

lemma *codomains-char*:
shows *codomains f* = (*if arr f then* {*cod f*} *else* {})
⟨*proof*⟩

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* $\longleftrightarrow$ «*f* : *dom f* $\rightarrow$ *cod f*»
⟨*proof*⟩

lemma *in-homI* [*intro*]:
assumes *arr f* **and** *dom f* = *a* **and** *cod f* = *b*
shows «*f* : *a* → *b*»
⟨*proof*⟩

lemma *in-homE* [*elim*]:
assumes «*f* : *a* → *b*»
and *arr f* ⟹ *dom f* = *a* ⟹ *cod f* = *b* ⟹ *T*
shows *T*
⟨*proof*⟩

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* · *f* is an arrow, but also that if *g* · *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*
⟨*proof*⟩

lemma *arr-cod-iff-arr*:
shows *arr* (*cod f*) ⟷ *arr f*
⟨*proof*⟩

lemma *arr-dom* [*simp*]:
assumes *arr f*
shows *arr* (*dom f*)
⟨*proof*⟩

lemma *arr-cod* [*simp*]:
assumes *arr f*
shows *arr* (*cod f*)
⟨*proof*⟩

lemma *seqI* [*simp*]:
assumes *arr f* **and** *arr g* **and** *dom g* = *cod f*
shows *seq g f*
⟨*proof*⟩

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

lemma *seqI'* [*intro*]:
assumes «$f : a \to b$» **and** «$g : b \to c$»
shows *seq g f*
⟨*proof*⟩

lemma *compatible-iff-seq*:
shows *domains g* ∩ *codomains f* ≠ {} ⟷ *seq g f*
⟨*proof*⟩

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
⟨*proof*⟩

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

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 «$g \cdot f : a \to c$»
⟨*proof*⟩

lemma *comp-in-homE* [*elim*]:
assumes «$g \cdot f : a \to c$»
obtains b **where** «$f : a \to b$» **and** «$g : b \to c$»
⟨*proof*⟩

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 $f \cdot a = f$
⟨*proof*⟩

lemma *comp-cod-arr*:
assumes *arr f* **and** *cod f* = b
shows $b \cdot f = f$
⟨*proof*⟩

lemma *ide-char*:
shows *ide a* ⟷ *arr a* ∧ *dom a* = *a* ∧ *cod a* = *a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ide-dom* [*simp*]:
assumes *arr f*
shows *ide* (*dom f*)
⟨*proof*⟩

lemma *ide-cod* [*simp*]:
assumes *arr f*
shows *ide* (*cod f*)
⟨*proof*⟩

lemma *dom-eqI*:
assumes *ide a* **and** *seq f a*
shows *dom f* = *a*
⟨*proof*⟩

lemma *cod-eqI*:
assumes *ide b* **and** *seq b f*
shows *cod f* = *b*
⟨*proof*⟩

lemma *dom-eqI'*:
assumes *a* ∈ *domains f*
shows *a* = *dom f*
⟨*proof*⟩

lemma *cod-eqI'*:
assumes *a* ∈ *codomains f*
shows *a* = *cod f*
⟨*proof*⟩

lemma *ide-char'*:
shows *ide a* ⟷ *arr a* ∧ (*dom a* = *a* ∨ *cod a* = *a*)
⟨*proof*⟩

lemma *dom-dom*:
shows *dom* (*dom f*) = *dom f*

⟨*proof*⟩

lemma *cod-cod*:
shows *cod* (*cod f*) = *cod f*
⟨*proof*⟩

lemma *dom-cod*:
shows *dom* (*cod f*) = *cod f*
⟨*proof*⟩

lemma *cod-dom*:
shows *cod* (*dom f*) = *dom f*
⟨*proof*⟩

lemma *dom-comp* [*simp*]:
assumes *seq g f*
shows *dom* ($g \cdot f$) = *dom f*
⟨*proof*⟩

lemma *cod-comp* [*simp*]:
assumes *seq g f*
shows *cod* ($g \cdot f$) = *cod g*
⟨*proof*⟩

lemma *comp-ide-self* [*simp*]:
assumes *ide a*
shows $a \cdot a = a$
⟨*proof*⟩

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
⟨*proof*⟩

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 $f \cdot g \cdot h = k \cdot l \cdot h$
⟨*proof*⟩

lemma *comp-reduce*:
assumes $f \cdot g = k$ **and** *seq f g* **and** *seq g h*
shows $f \cdot g \cdot h = k \cdot h$

⟨*proof*⟩

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

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

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

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

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

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

end

end

Chapter 3

EpiMonoIso

theory *EpiMonoIso*
imports *Category*
begin

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

context *category*
begin

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

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** $\bigwedge g\ g'.\ [\![seq\ g\ f;\ seq\ g'\ f;\ g \cdot f = g' \cdot f]\!] \Longrightarrow g = g'$
shows *epi f*
⟨*proof*⟩

lemma *epi-implies-arr*:
assumes *epi f*
shows *arr f*
⟨*proof*⟩

lemma *epiE* [*elim*]:
assumes *epi f*
and *seq g f* **and** $seq\ g'\ f$ **and** $g \cdot f = g' \cdot f$
shows $g = g'$
⟨*proof*⟩

lemma *monoI* [*intro*]:
assumes *arr g* **and** $\bigwedge f\ f'.\ [\![seq\ g\ f;\ seq\ g\ f';\ g \cdot f = g \cdot f']\!] \Longrightarrow f = f'$

shows *mono g*
⟨*proof*⟩

lemma *mono-implies-arr*:
assumes *mono f*
shows *arr f*
⟨*proof*⟩

lemma *monoE* [*elim*]:
assumes *mono g*
and *seq g f* **and** *seq g f'* **and** $g \cdot f = g \cdot f'$
shows $f' = f$
⟨*proof*⟩

definition *inverse-arrows*
where *inverse-arrows f g* ≡ *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*
⟨*proof*⟩

lemma *inverse-arrowsE* [*elim*]:
assumes *inverse-arrows f g*
and ⟦ *ide* $(g \cdot f)$; *ide* $(f \cdot g)$ ⟧ $\Longrightarrow T$
shows T
⟨*proof*⟩

lemma *inverse-arrows-sym*:
shows *inverse-arrows f g* ⟷ *inverse-arrows g f*
⟨*proof*⟩

lemma *ide-self-inverse*:
assumes *ide a*
shows *inverse-arrows a a*
⟨*proof*⟩

lemma *inverse-arrow-unique*:
assumes *inverse-arrows f g* **and** *inverse-arrows f g'*
shows $g = g'$
⟨*proof*⟩

lemma *inverse-arrows-compose*:
assumes *seq g f* **and** *inverse-arrows f f'* **and** *inverse-arrows g g'*
shows *inverse-arrows* $(g \cdot f)$ $(f' \cdot g')$
⟨*proof*⟩

definition *section*
where *section f* ≡ ∃ *g*. *ide* $(g \cdot f)$

lemma *sectionI* [*intro*]:
assumes *ide* $(g \cdot f)$
shows *section f*
⟨*proof*⟩

lemma *sectionE* [*elim*]:
assumes *section f*
obtains *g* **where** *ide* $(g \cdot f)$
⟨*proof*⟩

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

lemma *retractionI* [*intro*]:
assumes *ide* $(g \cdot f)$
shows *retraction g*
⟨*proof*⟩

lemma *retractionE* [*elim*]:
assumes *retraction g*
obtains *f* **where** *ide* $(g \cdot f)$
⟨*proof*⟩

lemma *section-is-mono*:
assumes *section g*
shows *mono g*
⟨*proof*⟩

lemma *retraction-is-epi*:
assumes *retraction g*
shows *epi g*
⟨*proof*⟩

lemma *section-retraction-compose*:
assumes *ide* $(e \cdot m)$ **and** *ide* $(e' \cdot m')$ **and** *seq* m' m
shows *ide* $((e \cdot e') \cdot (m' \cdot m))$
⟨*proof*⟩

lemma *sections-compose* [*intro*]:
assumes *section m* **and** *section* m' **and** *seq* m' m
shows *section* $(m' \cdot m)$
⟨*proof*⟩

lemma *retractions-compose* [*intro*]:
assumes *retraction e* **and** *retraction* e' **and** *seq* e' e
shows *retraction* $(e' \cdot e)$
⟨*proof*⟩

lemma *monos-compose* [*intro*]:
assumes *mono m* **and** *mono m'* **and** *seq m' m*
shows *mono* $(m' \cdot m)$
⟨*proof*⟩

lemma *epis-compose* [*intro*]:
assumes *epi e* **and** *epi e'* **and** *seq e' e*
shows *epi* $(e' \cdot e)$
⟨*proof*⟩

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

lemma *isoI* [*intro*]:
assumes *inverse-arrows f g*
shows *iso f*
⟨*proof*⟩

lemma *isoE* [*elim*]:
assumes *iso f*
obtains *g* **where** *inverse-arrows f g*
⟨*proof*⟩

lemma *ide-is-iso* [*simp*]:
assumes *ide a*
shows *iso a*
⟨*proof*⟩

lemma *iso-is-arr*:
assumes *iso f*
shows *arr f*
⟨*proof*⟩

lemma *iso-is-section*:
assumes *iso f*
shows *section f*
⟨*proof*⟩

lemma *iso-is-retraction*:
assumes *iso f*
shows *retraction f*
⟨*proof*⟩

lemma *iso-iff-mono-and-retraction*:
shows *iso f* $\longleftrightarrow$ *mono f* $\wedge$ *retraction f*
⟨*proof*⟩

lemma *iso-iff-section-and-epi*:
shows *iso f* $\longleftrightarrow$ *section f* $\wedge$ *epi f*

⟨*proof*⟩

lemma *iso-iff-section-and-retraction*:
shows *iso f* $\longleftrightarrow$ *section f* $\wedge$ *retraction f*
⟨*proof*⟩

lemma *isos-compose* [*intro*]:
assumes *iso f* **and** *iso f′* **and** *seq f′ f*
shows *iso* $(f' \cdot f)$
⟨*proof*⟩

lemma *iso-cancel-left*:
assumes *iso f* **and** $f \cdot g = f \cdot g'$ **and** *seq f g*
shows $g = g'$
⟨*proof*⟩

lemma *iso-cancel-right*:
assumes *iso g* **and** $f \cdot g = f' \cdot g$ **and** *seq f g* **and** *iso g*
shows $f = f'$
⟨*proof*⟩

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

lemma *isomorphicI* [*intro*]:
assumes *iso f*
shows *isomorphic* (*dom f*) (*cod f*)
⟨*proof*⟩

lemma *isomorphicE* [*elim*]:
assumes *isomorphic a a′*
obtains *f* **where** «*f* : *a* → *a′*» ∧ *iso f*
⟨*proof*⟩

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

lemma *iso-in-homI* [*intro*]:
assumes «*f* : *a* → *b*» **and** *iso f*
shows «*f* : *a* ≅ *b*»
⟨*proof*⟩

lemma *iso-in-homE* [*elim*]:
assumes «*f* : *a* ≅ *b*»
and ⟦«*f* : *a* → *b*»; *iso f*⟧ ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *isomorphicI′*:

assumes «$f : a \cong b$»
shows *isomorphic a b*
⟨*proof*⟩

lemma *ide-iso-in-hom*:
assumes *ide a*
shows «$a : a \cong a$»
⟨*proof*⟩

lemma *comp-iso-in-hom* [*intro*]:
assumes «$f : a \cong b$» **and** «$g : b \cong c$»
shows «$g \cdot f : a \cong c$»
⟨*proof*⟩

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

lemma *inv-is-inverse*:
assumes *iso f*
shows *inverse-arrows f* (*inv f*)
⟨*proof*⟩

lemma *iso-inv-iso* [*intro*, *simp*]:
assumes *iso f*
shows *iso* (*inv f*)
⟨*proof*⟩

lemma *inverse-unique*:
assumes *inverse-arrows f g*
shows *inv f* = *g*
⟨*proof*⟩

lemma *inv-ide* [*simp*]:
assumes *ide a*
shows *inv a* = *a*
⟨*proof*⟩

lemma *inv-inv* [*simp*]:
assumes *iso f*
shows *inv* (*inv f*) = *f*
⟨*proof*⟩

lemma *comp-arr-inv*:
assumes *inverse-arrows f g*
shows $f \cdot g$ = *dom g*
⟨*proof*⟩

lemma *comp-inv-arr*:
assumes *inverse-arrows f g*

shows *g* · *f* = *dom f*
⟨*proof*⟩

lemma *comp-arr-inv'*:
assumes *iso f*
shows *f* · *inv f* = *cod f*
⟨*proof*⟩

lemma *comp-inv-arr'*:
assumes *iso f*
shows *inv f* · *f* = *dom f*
⟨*proof*⟩

lemma *inv-in-hom* [*simp*]:
assumes *iso f* **and** «*f* : *a* → *b*»
shows «*inv f* : *b* → *a*»
⟨*proof*⟩

lemma *arr-inv* [*simp*]:
assumes *iso f*
shows *arr* (*inv f*)
⟨*proof*⟩

lemma *dom-inv* [*simp*]:
assumes *iso f*
shows *dom* (*inv f*) = *cod f*
⟨*proof*⟩

lemma *cod-inv* [*simp*]:
assumes *iso f*
shows *cod* (*inv f*) = *dom f*
⟨*proof*⟩

lemma *inv-comp*:
assumes *iso f* **and** *iso g* **and** *seq g f*
shows *inv* (*g* · *f*) = *inv f* · *inv g*
⟨*proof*⟩

lemma *isomorphic-reflexive*:
assumes *ide f*
shows *isomorphic f f*
⟨*proof*⟩

lemma *isomorphic-symmetric*:
assumes *isomorphic f g*
shows *isomorphic g f*
⟨*proof*⟩

lemma *isomorphic-transitive* [*trans*]:

assumes *isomorphic f g* **and** *isomorphic g h*
shows *isomorphic f h*
⟨*proof*⟩

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

lemma *section-retraction-of-iso*:
assumes *iso f*
shows *ide* $(g \cdot f) \Longrightarrow$ *inverse-arrows f g*
and *ide* $(f \cdot g) \Longrightarrow$ *inverse-arrows f g*
⟨*proof*⟩

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*) $h \land g =$ *inv f* $\cdot\ h$
and *iso g* $\Longrightarrow$ *seq h* (*inv g*) $\land f = h \cdot$ *inv g*
⟨*proof*⟩

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*) $\land$ *seq* (*inv f*) $h \land g \cdot$ *inv k* $=$ *inv f* $\cdot\ h$
⟨*proof*⟩

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

lemma *inv-comp-left*:
assumes *iso* $(g \cdot f)$ **and** *iso g*
shows *inv* $(g \cdot f) =$ *inv f* $\cdot$ *inv g* **and** *iso f*
⟨*proof*⟩

lemma *inv-comp-right*:
assumes *iso* $(g \cdot f)$ **and** *iso f*
shows *inv* $(g \cdot f) =$ *inv f* $\cdot$ *inv g* **and** *iso g*
⟨*proof*⟩

end

end

Chapter 4

DualCategory

theory *DualCategory*
imports *EpiMonoIso*
begin

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

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

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

lemma *comp-char* [*simp*]:
shows $g \cdot^{op} f = f \cdot g$
⟨*proof*⟩

interpretation *partial-composition comp*
⟨*proof*⟩

notation *in-hom* («- : - ← -»)

lemma *null-char* [*simp*]:
shows *null* = *C.null*
⟨*proof*⟩

lemma *ide-char* [*simp*]:
shows *ide a* ⟷ *C.ide a*
⟨*proof*⟩

lemma *domains-char*:
shows *domains f* = *C.codomains f*
⟨*proof*⟩

lemma *codomains-char*:
shows *codomains f* = *C.domains f*
⟨*proof*⟩

interpretation *category comp*
⟨*proof*⟩

lemma *is-category*:
shows *category comp* ⟨*proof*⟩

end

sublocale *dual-category* ⊆ *category comp*
⟨*proof*⟩

context *dual-category*
begin

lemma *dom-char* [*simp*]:
shows *dom f* = *C.cod f*
⟨*proof*⟩

lemma *cod-char* [*simp*]:
shows *cod f* = *C.dom f*
⟨*proof*⟩

lemma *arr-char* [*simp*]:
shows *arr f* ⟷ *C.arr f*
⟨*proof*⟩

lemma *hom-char* [*simp*]:
shows *in-hom f b a* ⟷ *C.in-hom f a b*
⟨*proof*⟩

lemma *seq-char* [*simp*]:
shows *seq g f* = *C.seq f g*
⟨*proof*⟩

lemma *iso-char* [*simp*]:
shows *iso f* ⟷ *C.iso f*
⟨*proof*⟩

end

end

Chapter 5

Concrete Categories

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

theory *ConcreteCategory*
imports *Category*
begin

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

Comp D C A h (*Comp C B A g f*) = *Comp D B A* (*Comp D C B h g*) *f*

begin

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

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

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

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

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

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

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

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

interpretation *partial-composition COMP*
  ⟨*proof*⟩

lemma *null-char*:
shows *null* = *Null*
⟨*proof*⟩

lemma *ide-char*$_{CC}$:
shows *ide f* ⟷ *Ide f*
⟨*proof*⟩

lemma *ide-MkIde* [*simp*]:
assumes *A* ∈ *Obj*
shows *ide* (*MkIde A*)

⟨*proof*⟩

lemma *in-domains-char*:
shows $a \in \mathit{domains}\ f \longleftrightarrow \mathit{Arr}\ f \wedge a = \mathit{MkIde}\ (\mathit{Dom}\ f)$
⟨*proof*⟩

lemma *in-codomains-char*:
shows $b \in \mathit{codomains}\ f \longleftrightarrow \mathit{Arr}\ f \wedge b = \mathit{MkIde}\ (\mathit{Cod}\ f)$
⟨*proof*⟩

lemma *arr-char*:
shows $\mathit{arr}\ f \longleftrightarrow \mathit{Arr}\ f$
⟨*proof*⟩

lemma arrI_{CC}:
assumes $f \neq \mathit{Null}$ **and** $\mathit{Dom}\ f \in \mathit{Obj}$ $\mathit{Cod}\ f \in \mathit{Obj}$ $\mathit{Map}\ f \in \mathit{Hom}\ (\mathit{Dom}\ f)\ (\mathit{Cod}\ f)$
shows $\mathit{arr}\ f$
⟨*proof*⟩

lemma *arrE*:
assumes $\mathit{arr}\ f$
and $[\![f \neq \mathit{Null}; \mathit{Dom}\ f \in \mathit{Obj}; \mathit{Cod}\ f \in \mathit{Obj}; \mathit{Map}\ f \in \mathit{Hom}\ (\mathit{Dom}\ f)\ (\mathit{Cod}\ f)]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *arr-MkArr* [*simp*]:
assumes $A \in \mathit{Obj}$ **and** $B \in \mathit{Obj}$ **and** $f \in \mathit{Hom}\ A\ B$
shows $\mathit{arr}\ (\mathit{MkArr}\ A\ B\ f)$
⟨*proof*⟩

lemma *MkArr-Map*:
assumes $\mathit{arr}\ f$
shows $\mathit{MkArr}\ (\mathit{Dom}\ f)\ (\mathit{Cod}\ f)\ (\mathit{Map}\ f) = f$
⟨*proof*⟩

lemma *Arr-comp*:
assumes $\mathit{arr}\ f$ **and** $\mathit{arr}\ g$ **and** $\mathit{Dom}\ g = \mathit{Cod}\ f$
shows $\mathit{Arr}\ (\mathit{COMP}\ g\ f)$
⟨*proof*⟩

lemma *Dom-comp* [*simp*]:
assumes $\mathit{arr}\ f$ **and** $\mathit{arr}\ g$ **and** $\mathit{Dom}\ g = \mathit{Cod}\ f$
shows $\mathit{Dom}\ (\mathit{COMP}\ g\ f) = \mathit{Dom}\ f$
⟨*proof*⟩

lemma *Cod-comp* [*simp*]:
assumes $\mathit{arr}\ f$ **and** $\mathit{arr}\ g$ **and** $\mathit{Dom}\ g = \mathit{Cod}\ f$
shows $\mathit{Cod}\ (\mathit{COMP}\ g\ f) = \mathit{Cod}\ g$
⟨*proof*⟩

lemma *Map-comp* [*simp*]:
assumes *arr f* **and** *arr g* **and** *Dom g* = *Cod f*
shows *Map* (*COMP g f*) = *Comp* (*Cod g*) (*Dom g*) (*Dom f*) (*Map g*) (*Map f*)
⟨*proof*⟩

lemma *seq-char*:
shows *seq g f* ⟷ *arr f* ∧ *arr g* ∧ *Dom g* = *Cod f*
⟨*proof*⟩

interpretation *category COMP*
⟨*proof*⟩

proposition *is-category*:
shows *category COMP*
⟨*proof*⟩

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

lemma *Dom-in-Obj*:
assumes *arr f*
shows *Dom f* ∈ *Obj*
⟨*proof*⟩

lemma *Cod-in-Obj*:
assumes *arr f*
shows *Cod f* ∈ *Obj*
⟨*proof*⟩

lemma *Map-in-Hom*:
assumes *arr f*
shows *Map f* ∈ *Hom* (*Dom f*) (*Cod f*)
⟨*proof*⟩

lemma *MkArr-in-hom*:
assumes *A* ∈ *Obj* **and** *B* ∈ *Obj* **and** *f* ∈ *Hom A B*
shows *in-hom* (*MkArr A B f*) (*MkIde A*) (*MkIde B*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cod-char*:
shows *cod f* = (*if arr f then MkIde* (*Cod f*) *else null*)
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

lemma *Dom-dom* [*simp*]:
assumes *arr f*
shows *Dom* (*dom f*) = *Dom f*
⟨*proof*⟩

lemma *Cod-dom* [*simp*]:
assumes *arr f*
shows *Cod* (*dom f*) = *Dom f*
⟨*proof*⟩

lemma *Dom-cod* [*simp*]:
assumes *arr f*
shows *Dom* (*cod f*) = *Cod f*
⟨*proof*⟩

lemma *Cod-cod* [*simp*]:
assumes *arr f*
shows *Cod* (*cod f*) = *Cod f*
⟨*proof*⟩

lemma *Map-dom* [*simp*]:
assumes *arr f*
shows *Map* (*dom f*) = *Id* (*Dom f*)
⟨*proof*⟩

lemma *Map-cod* [*simp*]:
assumes *arr f*
shows *Map* (*cod f*) = *Id* (*Cod f*)
⟨*proof*⟩

lemma *Map-ide*:
assumes *ide a*
shows *Map a* = *Id* (*Dom a*) **and** *Map a* = *Id* (*Cod a*)
⟨*proof*⟩

lemma *MkIde-Dom*:
assumes *arr a*

shows *MkIde* (*Dom a*) = *dom a*
⟨*proof*⟩

lemma *MkIde-Cod*:
assumes *arr a*
shows *MkIde* (*Cod a*) = *cod a*
⟨*proof*⟩

lemma *MkIde-Dom′* [*simp*]:
assumes *ide a*
shows *MkIde* (*Dom a*) = *a*
⟨*proof*⟩

lemma *MkIde-Cod′* [*simp*]:
assumes *ide a*
shows *MkIde* (*Cod a*) = *a*
⟨*proof*⟩

lemma *dom-MkArr* [*simp*]:
assumes *arr* (*MkArr A B F*)
shows *dom* (*MkArr A B F*) = *MkIde A*
⟨*proof*⟩

lemma *cod-MkArr* [*simp*]:
assumes *arr* (*MkArr A B F*)
shows *cod* (*MkArr A B F*) = *MkIde B*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

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 $MkArr\ (Dom\ a)\ (Dom\ b) \in Hom\ (Dom\ a)\ (Dom\ b) \rightarrow hom\ a\ b$
and $\bigwedge f.\ f \in hom\ a\ b \Longrightarrow MkArr\ (Dom\ a)\ (Dom\ b)\ (Map\ f) = f$
and $\bigwedge F.\ F \in Hom\ (Dom\ a)\ (Dom\ b) \Longrightarrow Map\ (MkArr\ (Dom\ a)\ (Dom\ b)\ F) = F$
and *bij-betw Map* $(hom\ a\ b)\ (Hom\ (Dom\ a)\ (Dom\ b))$
⟨*proof*⟩

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 $t = t'$
⟨*proof*⟩

end

sublocale *concrete-category* ⊆ *category COMP*
⟨*proof*⟩

end

Chapter 6

InitialTerminal

theory *InitialTerminal*
imports *EpiMonoIso*
begin

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

context *category*
begin

definition *initial*
where $\mathit{initial}\ a \equiv \mathit{ide}\ a \wedge (\forall\, b.\ \mathit{ide}\ b \longrightarrow (\exists!f.\ «f : a \rightarrow b»))$

definition *terminal*
where $\mathit{terminal}\ b \equiv \mathit{ide}\ b \wedge (\forall\, a.\ \mathit{ide}\ a \longrightarrow (\exists!f.\ «f : a \rightarrow b»))$

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

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

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

lemma *initial-arr-unique*:
assumes $\mathit{par}\ f\ f'$ **and** $\mathit{initial\text{-}arr}\ f$ **and** $\mathit{initial\text{-}arr}\ f'$
shows $f = f'$
$\langle \mathit{proof} \rangle$

lemma *initialI* [*intro*]:
assumes $\mathit{ide}\ a$ **and** $\bigwedge b.\ \mathit{ide}\ b \Longrightarrow \exists!f.\ «f : a \rightarrow b»$
shows *initial a*
$\langle \mathit{proof} \rangle$

lemma *initialE* [*elim*]:
assumes *initial a* **and** *ide b*
obtains *f* **where** «$f : a \rightarrow b$» **and** $\bigwedge f'$. «$f' : a \rightarrow b$» $\Longrightarrow f' = f$
⟨*proof*⟩

lemma *terminal-arr-unique*:
assumes *par f f'* **and** *terminal-arr f* **and** *terminal-arr f'*
shows $f = f'$
⟨*proof*⟩

lemma *terminalI* [*intro*]:
assumes *ide b* **and** $\bigwedge a$. *ide a* $\Longrightarrow \exists!f$. «$f : a \rightarrow b$»
shows *terminal b*
⟨*proof*⟩

lemma *terminalE* [*elim*]:
assumes *terminal b* **and** *ide a*
obtains *f* **where** «$f : a \rightarrow b$» **and** $\bigwedge f'$. «$f' : a \rightarrow b$» $\Longrightarrow f' = f$
⟨*proof*⟩

lemma *terminal-objs-isomorphic*:
assumes *terminal a* **and** *terminal b*
shows *isomorphic a b*
⟨*proof*⟩

lemma *isomorphic-to-terminal-is-terminal*:
assumes *terminal a* **and** *isomorphic a a'*
shows *terminal a'*
⟨*proof*⟩

lemma *initial-objs-isomorphic*:
assumes *initial a* **and** *initial b*
shows *isomorphic a b*
⟨*proof*⟩

lemma *isomorphic-to-initial-is-initial*:
assumes *initial a* **and** *isomorphic a a'*
shows *initial a'*
⟨*proof*⟩

lemma *point-is-mono*:
assumes *point f*
shows *mono f*
⟨*proof*⟩

end

end

Chapter 7

Functor

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

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

```
locale functor =
  A: category A +
  B: category B
for A :: 'a comp      (infixr ·_A 55)
and B :: 'b comp      (infixr ·_B 55)
and F :: 'a ⇒ 'b +
assumes is-extensional: ¬A.arr f ⟹ F f = B.null
and preserves-arr: A.arr f ⟹ B.arr (F f)
and preserves-dom [iff]: A.arr f ⟹ B.dom (F f) = F (A.dom f)
and preserves-cod [iff]: A.arr f ⟹ B.cod (F f) = F (A.cod f)
and preserves-comp [iff]: A.seq g f ⟹ F (g ·_A f) = F g ·_B F f
begin

  notation A.in-hom     («- : - →_A -»)
  notation B.in-hom     («- : - →_B -»)

  lemma preserves-hom [intro]:
  assumes «f : a →_A b»
  shows «F f : F a →_B F b»
    ⟨proof⟩
```

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*) $\longleftrightarrow$ *A.arr f*
⟨*proof*⟩

lemma *preserves-seq* [*intro*]:
assumes *A.seq g f*
shows *B.seq* (*F g*) (*F f*)
⟨*proof*⟩

lemma *preserves-ide* [*simp*]:
assumes *A.ide a*
shows *B.ide* (*F a*)
⟨*proof*⟩

lemma *preserves-iso* [*simp*]:
assumes *A.iso f*
shows *B.iso* (*F f*)
⟨*proof*⟩

lemma *preserves-isomorphic*:
assumes *A.isomorphic a b*
shows *B.isomorphic* (*F a*) (*F b*)
⟨*proof*⟩

lemma *preserves-section-retraction*:
assumes *A.ide* (*A e m*)
shows *B.ide* (*B* (*F e*) (*F m*))
⟨*proof*⟩

lemma *preserves-section*:
assumes *A.section m*
shows *B.section* (*F m*)
⟨*proof*⟩

lemma *preserves-retraction*:
assumes *A.retraction e*
shows *B.retraction* (*F e*)
⟨*proof*⟩

lemma *preserves-inverse-arrows*:
assumes *A.inverse-arrows f g*
shows *B.inverse-arrows* (*F f*) (*F g*)
⟨*proof*⟩

lemma *preserves-inv*:
assumes *A.iso f*
shows *F* (*A.inv f*) = *B.inv* (*F f*)
⟨*proof*⟩

lemma *preserves-iso-in-hom* [*intro*]:
assumes *A.iso-in-hom f a b*
shows *B.iso-in-hom* (*F f*) (*F a*) (*F b*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *reflects-isomorphic*:
assumes *A.ide f* **and** *A.ide f*′ **and** *B.isomorphic* (*F f*) (*F f*′)
shows *A.isomorphic f f*′
⟨*proof*⟩

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*
  ⟨*proof*⟩

context *embedding-functor*
begin

  lemma *reflects-ide*:
  assumes $B.ide\ (F\ f)$
  shows $A.ide\ f$
    ⟨*proof*⟩

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 $map\ f = b$
    ⟨*proof*⟩

lemma *is-functor*:
shows *functor A B map*
⟨*proof*⟩

end

sublocale *constant-functor* ⊆ *functor A B map*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *functor C C map*
⟨*proof*⟩

lemma *is-functor*:
shows *functor C C map*
⟨*proof*⟩

sublocale *fully-faithful-functor C C map*
⟨*proof*⟩

lemma *is-fully-faithful*:
shows *fully-faithful-functor C C map*
⟨*proof*⟩

end

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

sublocale *category* ⊆ *identity-functor C* ⟨*proof*⟩

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*⟩

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

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

  sublocale *functor A C* ‹*G o F*›
    ⟨*proof*⟩

  lemma *is-functor*:
  shows *functor A C* (*G o F*)
    ⟨*proof*⟩

end

lemma *comp-functor-identity* [*simp*]:
assumes *functor A B F*
shows *F o identity-functor.map A* = *F*
⟨*proof*⟩

lemma *comp-identity-functor* [*simp*]:
assumes *functor A B F*
shows *identity-functor.map B o F* = *F*
⟨*proof*⟩

lemma *faithful-functors-compose*:
assumes *faithful-functor A B F* **and** *faithful-functor B C G*
shows *faithful-functor A C* (*G o F*)
⟨*proof*⟩

lemma *full-functors-compose*:
assumes *full-functor A B F* **and** *full-functor B C G*
shows *full-functor A C* (*G o F*)
⟨*proof*⟩

lemma *fully-faithful-functors-compose*:
assumes *fully-faithful-functor A B F* **and** *fully-faithful-functor B C G*
shows *full-functor A C* (*G o F*)
⟨*proof*⟩

lemma *embedding-functors-compose*:
assumes *embedding-functor A B F* **and** *embedding-functor B C G*
shows *embedding-functor A C* (*G o F*)
⟨*proof*⟩

lemma *full-embedding-functors-compose*:
assumes *full-embedding-functor A B F* **and** *full-embedding-functor B C G*
shows *full-embedding-functor A C* (*G o F*)
⟨*proof*⟩

lemma *essentially-surjective-functors-compose*:
assumes *essentially-surjective-functor A B F* **and** *essentially-surjective-functor B C G*
shows *essentially-surjective-functor A C* (*G o F*)
⟨*proof*⟩

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

  lemma *bij-betw-arr-sets*:
  shows *bij-betw F* (*Collect B.arr*) (*Collect A.arr*)
    ⟨*proof*⟩

end

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*: ∃ *F G*. *inverse-functors A B F G*

sublocale *inverse-functors* ⊆ *isomorphic-categories A B*
  ⟨*proof*⟩

lemma *inverse-functors-sym*:
assumes *inverse-functors A B F G*
shows *inverse-functors B A G F*
⟨*proof*⟩

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*⟩

lemma *inverse-functor-unique′*:
assumes *inverse-functors C D F G* **and** *inverse-functors C D F′ G*
shows $F = F'$
⟨*proof*⟩

locale *invertible-functor* =
A: *category A* +
B: *category B* +
G: *functor A B G*
for *A* :: *′a comp* (**infixr** $\cdot_A$ *55*)
and *B* :: *′b comp* (**infixr** $\cdot_B$ *55*)
and *G* :: *′a* ⇒ *′b* +
assumes *invertible*: ∃ *F*. *inverse-functors A B F G*
begin

lemma *has-unique-inverse*:
shows ∃!*F*. *inverse-functors A B F G*
⟨*proof*⟩

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

interpretation *inverse-functors A B inv G*
⟨*proof*⟩

lemma *inv-is-inverse*:
shows *inverse-functors A B inv G* ⟨*proof*⟩

sublocale *fully-faithful-functor A B G*
⟨*proof*⟩

lemma *is-fully-faithful*:
shows *fully-faithful-functor A B G*
⟨*proof*⟩

lemma *preserves-terminal*:
assumes *A.terminal a*
shows *B.terminal* (*G a*)
⟨*proof*⟩

end

sublocale *invertible-functor* ⊆ *inverse-functors A B inv G*

⟨*proof*⟩

locale *dual-functor* =
F: *functor A B F* +
Aop: *dual-category A* +
Bop: *dual-category B*
for *A* :: *'a comp* (**infixr** $\cdot_A$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *55*)
and *F* :: *'a* ⇒ *'b*
begin

notation *Aop.comp* (**infixr** $\cdot_A{}^{op}$ *55*)
notation *Bop.comp* (**infixr** $\cdot_B{}^{op}$ *55*)

abbreviation *map*
where *map* ≡ *F*

lemma *is-functor*:
shows *functor Aop.comp Bop.comp map*
⟨*proof*⟩

end

sublocale *dual-functor* ⊆ *functor Aop.comp Bop.comp map*
⟨*proof*⟩

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

context *category*
begin

lemma *bij-induces-invertible-functor*:
assumes *bij-betw* φ *S* (*Collect arr*) **and** $n \notin S$
shows ∃ *C'*. *Collect* (*partial-composition.arr C'*) = *S* ∧
invertible-functor C' C (λ*i*. *if partial-composition.arr C' i then* φ *i else null*)
⟨*proof*⟩

corollary (**in** *category*) *finite-imp-ex-iso-nat-comp*:
assumes *finite* (*Collect arr*)
shows ∃ *C'* :: *nat comp*. *isomorphic-categories C' C*
⟨*proof*⟩

end

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

context *category*

begin

interpretation *CC*: *concrete-category* ‹*Collect ide*› *hom id* ‹λ- - - *g f*. *g* · *f*›
⟨*proof*⟩

interpretation *F*: *functor C CC.COMP*
‹λ*f*. *if arr f then CC.MkArr* (*dom f*) (*cod f*) *f else CC.null*›
⟨*proof*⟩

interpretation *G*: *functor CC.COMP C* ‹λ*F*. *if CC.arr F then CC.Map F else null*›
⟨*proof*⟩

interpretation *FG*: *inverse-functors C CC.COMP*
‹λ*F*. *if CC.arr F then CC.Map F else null*›
‹λ*f*. *if arr f then CC.MkArr* (*dom f*) (*cod f*) *f else CC.null*›
⟨*proof*⟩

theorem *is-isomorphic-to-concrete-category*:
shows *isomorphic-categories C CC.COMP*
⟨*proof*⟩

end

end

Chapter 8

Subcategory

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

theory *Subcategory*
imports *Functor*
begin

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

no-notation *C.in-hom* («- : - → -»)
notation *C.in-hom* («- : - $\rightarrow_C$ -»)

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

interpretation *partial-composition comp*
⟨*proof*⟩

lemma *null-char* [*simp*]:
shows *null* = *C.null*
⟨*proof*⟩

lemma $ideI_{SbC}$:
assumes *Arr a* **and** *C.ide a*
shows *ide a*

⟨*proof*⟩

lemma *Arr-iff-dom-in-domain*:
shows *Arr f* ⟷ *C.dom f* ∈ *domains f*
⟨*proof*⟩

lemma *Arr-iff-cod-in-codomain*:
shows *Arr f* ⟷ *C.cod f* ∈ *codomains f*
⟨*proof*⟩

lemma $arr\text{-}char_{SbC}$:
shows *arr f* ⟷ *Arr f*
⟨*proof*⟩

lemma $arrI_{SbC}$ [*intro*]:
assumes *Arr f*
shows *arr f*
⟨*proof*⟩

lemma *arrE* [*elim*]:
assumes *arr f*
shows *Arr f*
⟨*proof*⟩

interpretation *category comp*
⟨*proof*⟩

theorem *is-category*:
shows *category comp* ⟨*proof*⟩

notation *in-hom* («- : - → -»)

lemma *dom-simp*:
assumes *arr f*
shows *dom f* = *C.dom f*
⟨*proof*⟩

lemma $dom\text{-}char_{SbC}$:
shows *dom f* = (*if arr f then C.dom f else C.null*)
⟨*proof*⟩

lemma *cod-simp*:
assumes *arr f*
shows *cod f* = *C.cod f*
⟨*proof*⟩

lemma $cod\text{-}char_{SbC}$:
shows *cod f* = (*if arr f then C.cod f else C.null*)
⟨*proof*⟩

lemma *in-hom-char*$_{SbC}$:
shows «f : a → b» ⟷ *arr a* ∧ *arr b* ∧ *arr f* ∧ «f : a $\rightarrow_C$ b»
⟨*proof*⟩

lemma *ide-char*$_{SbC}$:
shows *ide a* ⟷ *arr a* ∧ *C.ide a*
⟨*proof*⟩

lemma *seq-char*$_{SbC}$:
shows *seq g f* ⟷ *arr f* ∧ *arr g* ∧ *C.seq g f*
⟨*proof*⟩

lemma *hom-char*:
shows *hom a b* = *C.hom a b* ∩ *Collect Arr*
⟨*proof*⟩

lemma *comp-char*:
shows $g \cdot f$ = (*if arr f* ∧ *arr g* ∧ *C.seq g f then* $g \cdot_C f$ *else C.null*)
⟨*proof*⟩

lemma *comp-simp*:
assumes *seq g f*
shows $g \cdot f = g \cdot_C f$
⟨*proof*⟩

lemma *inclusion-preserves-inverse*:
assumes *inverse-arrows f g*
shows *C.inverse-arrows f g*
⟨*proof*⟩

lemma *iso-char*$_{SbC}$:
shows *iso f* ⟷ *C.iso f* ∧ *arr f* ∧ *arr* (*C.inv f*)
⟨*proof*⟩

lemma *inv-char*$_{SbC}$:
assumes *iso f*
shows *inv f* = *C.inv f*
⟨*proof*⟩

lemma *inverse-arrows-char*$_{SbC}$:
shows *inverse-arrows f g* ⟷ *seq f g* ∧ *C.inverse-arrows f g*
⟨*proof*⟩

end

sublocale *subcategory* ⊆ *category comp*
⟨*proof*⟩

8.1 Full Subcategory

locale *full-subcategory* =
C: *category C*
for *C* :: *'a comp*
and *Ide* :: *'a* ⇒ *bool* +
assumes $inclusion_{FSbC}$: *Ide f* ⟹ *C.ide f*
begin

sublocale *subcategory C* λ*f. C.arr f* ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*)
⟨*proof*⟩

lemma *is-subcategory*:
shows *subcategory C* (λ*f. C.arr f* ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*))
⟨*proof*⟩

lemma $in\text{-}hom\text{-}char_{FSbC}$:
shows «*f* : *a* → *b*» ⟷ *arr a* ∧ *arr b* ∧ «*f* : *a* $\rightarrow_C$ *b*»
⟨*proof*⟩

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

lemma $iso\text{-}char_{FSbC}$:
shows *iso f* ⟷ *arr f* ∧ *C.iso f*
⟨*proof*⟩

end

8.2 Inclusion Functor

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

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

interpretation *functor S.comp C S.map*
⟨*proof*⟩

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

interpretation *faithful-functor S.comp C S.map*
⟨*proof*⟩

lemma *is-faithful-functor*:

shows *faithful-functor S.comp C S.map* ⟨*proof*⟩

interpretation *embedding-functor S.comp C S.map*
⟨*proof*⟩

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

end

sublocale *inclusion-functor* ⊆ *faithful-functor S.comp C S.map*
⟨*proof*⟩
sublocale *inclusion-functor* ⊆ *embedding-functor S.comp C S.map*
⟨*proof*⟩

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*
begin

sublocale *inclusion-functor C* ‹λ*f*. *C.arr f* ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*)› ⟨*proof*⟩

lemma *is-inclusion-functor*:
shows *inclusion-functor C* (λ*f*. *C.arr f* ∧ *Ide* (*C.dom f*) ∧ *Ide* (*C.cod f*))
⟨*proof*⟩

interpretation *full-functor S.comp C S.map*
⟨*proof*⟩

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

sublocale *full-functor S.comp C S.map*
⟨*proof*⟩
sublocale *fully-faithful-functor S.comp C S.map* ⟨*proof*⟩

end

end

Chapter 9

SetCategory

theory *SetCategory*
imports *Category Functor Subcategory*
begin

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

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

9.1 Some Lemmas about Restriction

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

lemma *restr-eqI*:
assumes $A = A'$ **and** $\bigwedge x.\ x \in A \Longrightarrow F\ x = F'\ x$
shows *restrict* $F\ A =$ *restrict* $F'\ A'$
⟨*proof*⟩

lemma *restr-eqE* [*elim*]:
assumes *restrict* $F\ A =$ *restrict* $F'\ A$ **and** $x \in A$
shows $F\ x = F'\ x$
⟨*proof*⟩

lemma *compose-eq'* [*simp*]:
shows *compose* $A\ G\ F =$ *restrict* $(G\ o\ F)\ A$
⟨*proof*⟩

9.2 Set Categories

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

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

notation *in-hom* («- : - → -»)

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

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

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

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

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

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

end

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

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

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

lemma *setp-singleton*:
assumes *terminal a*
shows *setp {a}*
⟨*proof*⟩

lemma *setp-empty*:
shows *setp {}*
⟨*proof*⟩

lemma *finite-imp-setp*:

assumes $A \subseteq Univ$ **and** *finite A*
shows *setp A*
⟨*proof*⟩

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

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

lemma *comp-arr-point*$_{SC}$:
assumes *arr f* **and** «x : *unity* → *dom f*»
shows $f \cdot x$ = *inv-into* (*hom unity* (*cod f*)) *img* (*Fun f* (*img x*))
⟨*proof*⟩

Parallel arrows that determine the same function are equal.

lemma *arr-eqI*$_{SC}$:
assumes *par f f′* **and** *Fun f* = *Fun f′*
shows $f = f'$
⟨*proof*⟩

lemma *terminal-unity*$_{SC}$:
shows *terminal unity*
⟨*proof*⟩

lemma *ide-unity* [*simp*]:
shows *ide unity*
⟨*proof*⟩

lemma *setp-set′* [*simp*]:
assumes *ide a*
shows *setp* (*set a*)
⟨*proof*⟩

lemma *inj-on-set*:
shows *inj-on set* (*Collect ide*)
⟨*proof*⟩

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* $\Rightarrow 's$
where *mkIde A* = (*if setp A then inv-into* (*Collect ide*) *set A else null*)

lemma *mkIde-set* [*simp*]:
assumes *ide a*
shows *mkIde* (*set a*) = *a*
⟨*proof*⟩

lemma *set-mkIde* [*simp*]:
assumes *setp A*

shows *set* (*mkIde A*) = *A*
⟨*proof*⟩

lemma *ide-mkIde* [*simp*]:
assumes *setp A*
shows *ide* (*mkIde A*)
⟨*proof*⟩

lemma *arr-mkIde* [*iff*]:
shows *arr* (*mkIde A*) ⟷ *setp A*
⟨*proof*⟩

lemma *dom-mkIde* [*simp*]:
assumes *setp A*
shows *dom* (*mkIde A*) = *mkIde A*
⟨*proof*⟩

lemma *cod-mkIde* [*simp*]:
assumes *setp A*
shows *cod* (*mkIde A*) = *mkIde A*
⟨*proof*⟩

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

lemma *Fun-mapsto*:
assumes *arr f*
shows *Fun f* ∈ *extensional* (*set* (*dom f*)) ∩ (*set* (*dom f*) → *set* (*cod f*))
⟨*proof*⟩

Identities of *S* correspond to restrictions of the identity function.

lemma *Fun-ide*:
assumes *ide a*
shows *Fun a* = *restrict* (λ*x*. *x*) (*set a*)
⟨*proof*⟩

lemma *Fun-mkIde* [*simp*]:
assumes *setp A*
shows *Fun* (*mkIde A*) = *restrict* (λ*x*. *x*) *A*
⟨*proof*⟩

Composition in (·) corresponds to extensional function composition.

lemma *Fun-comp* [*simp*]:
assumes *seq g f*
shows *Fun* (*g* · *f*) = *restrict* (*Fun g o Fun f*) (*set* (*dom f*))
⟨*proof*⟩

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

definition *mkArr* :: *'s set* ⇒ *'s set* ⇒ (*'s* ⇒ *'s*) ⇒ *'s*
where *mkArr A B F* = (*if setp A* ∧ *setp B* ∧ *F* ∈ *A* → *B*

then (*THE f*. *f* ∈ *hom* (*mkIde A*) (*mkIde B*) ∧ *Fun f* = *restrict F A*)
else null)

Each function $F \in set\ a \rightarrow set\ b$ 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** $F \in set\ a \rightarrow set\ b$
shows ∃!*f*. «*f* : *a* → *b*» ∧ *Fun f* = *restrict F* (*set a*)
⟨*proof*⟩

lemma *mkArr-in-hom*:
assumes *setp A* **and** *setp B* **and** $F \in A \rightarrow B$
shows «*mkArr A B F* : *mkIde A* → *mkIde B*»
⟨*proof*⟩

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*) ⟷ *setp A* ∧ *setp B* ∧ $F \in A \rightarrow B$
⟨*proof*⟩

lemma *arr-mkArrI* [*intro*]:
assumes *setp A* **and** *setp B* **and** $F \in A \rightarrow B$
shows *arr* (*mkArr A B F*)
⟨*proof*⟩

lemma *Fun-mkArr'*:
assumes *arr* (*mkArr A B F*)
shows «*mkArr A B F* : *mkIde A* → *mkIde B*»
and *Fun* (*mkArr A B F*) = *restrict F A*
⟨*proof*⟩

lemma *mkArr-Fun*:
assumes *arr f*
shows *mkArr* (*set* (*dom f*)) (*set* (*cod f*)) (*Fun f*) = *f*
⟨*proof*⟩

lemma *dom-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)
shows *dom* (*mkArr A B F*) = *mkIde A*

⟨*proof*⟩

lemma *cod-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)
shows *cod* (*mkArr A B F*) = *mkIde B*
⟨*proof*⟩

lemma *Fun-mkArr* [*simp*]:
assumes *arr* (*mkArr A B F*)
shows *Fun* (*mkArr A B F*) = *restrict F A*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *mkArr-restrict-eq*:
assumes *arr* (*mkArr A B F*)
shows *mkArr A B* (*restrict F A*) = *mkArr A B F*
⟨*proof*⟩

lemma *mkArr-restrict-eq′*:
assumes *arr* (*mkArr A B* (*restrict F A*))
shows *mkArr A B* (*restrict F A*) = *mkArr A B F*
⟨*proof*⟩

lemma *mkIde-as-mkArr* [*simp*]:
assumes *setp A*
shows *mkArr A A* ($\lambda x.\ x$) = *mkIde A*
⟨*proof*⟩

lemma *comp-mkArr*:
assumes *arr* (*mkArr A B F*) **and** *arr* (*mkArr B C G*)
shows *mkArr B C G* · *mkArr A B F* = *mkArr A C* ($G \circ F$)
⟨*proof*⟩

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

lemma *terminal-char1*:
shows *terminal t* ⟷ *ide t* ∧ (∃!*x*. *x* ∈ *set t*)
⟨*proof*⟩

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

lemma *terminal-char2*:
shows *terminal t* ⟷ *ide t* ∧ *set t* = {*t*}
⟨*proof*⟩

end

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

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

notation *in-hom* («- : - → -»)

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

sublocale *set-category-given-img S some-img setp*
⟨*proof*⟩

end

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

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

abbreviation *setp*
where *setp* ≡ λ*A*. *A* ⊆ *Univ*

lemma *is-set-category*:
shows *set-category S* (λ*A*. *A* ⊆ *Collect terminal*)
⟨*proof*⟩

end

context *set-category*
begin

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

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

lemma *Fun-incl*:
assumes *incl f*
shows *Fun f* = (λ*x* ∈ *set* (*dom f*). *x*)
⟨*proof*⟩

lemma *ex-incl-iff-subset*:
assumes *ide a* **and** *ide b*
shows (∃*f*. «*f* : *a* → *b*» ∧ *incl f*) ⟷ *set a* ⊆ *set b*
⟨*proof*⟩

end

9.3 Categoricity

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

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

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

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

lemma ψ-φ:
assumes $t \in S.Univ$
shows $\psi\ (\varphi\ t) = t$
$\langle proof \rangle$

lemma φ-ψ:
assumes $t' \in S'.Univ$
shows $\varphi\ (\psi\ t') = t'$
$\langle proof \rangle$

lemma ψ-*img*-φ-*img*:
assumes $A \subseteq S.Univ$
shows $\psi\ `\ \varphi\ `\ A = A$
$\langle proof \rangle$

lemma φ-*img*-ψ-*img*:
assumes $A' \subseteq S'.Univ$
shows $\varphi\ `\ \psi\ `\ A' = A'$
$\langle proof \rangle$

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

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

lemma *set*-Φo:
assumes $S.ide\ a$
shows $S'.set\ (\Phi o\ a) = \varphi\ `\ S.set\ a$
$\langle proof \rangle$

lemma Φo-*preserves-ide*:
assumes $S.ide\ a$
shows $S'.ide\ (\Phi o\ a)$
$\langle proof \rangle$

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

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

lemma Φa-*mapsto*:
assumes $S.arr\ f$
shows $\Phi a\ f \in S'.set\ (\Phi o\ (S.dom\ f)) \rightarrow S'.set\ (\Phi o\ (S.cod\ f))$
$\langle proof \rangle$

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

lemma Φa-*comp*:
assumes *gf*: $S.seq\ g\ f$
shows $\Phi a\ (g \cdot f) = restrict\ (\Phi a\ g\ o\ \Phi a\ f)\ (S'.set\ (\Phi o\ (S.dom\ f)))$

$\langle proof \rangle$

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

definition Φ
where $\Phi\ f = (\mathit{if}\ S.arr\ f\ \mathit{then}$
$S'.mkArr\ (S'.set\ (\Phi o\ (S.dom\ f)))\ (S'.set\ (\Phi o\ (S.cod\ f)))\ (\Phi a\ f)$
$\mathit{else}\ S'.null)$

lemma Φ-*in-hom*:
assumes $S.arr\ f$
shows $\Phi\ f \in S'.hom\ (\Phi o\ (S.dom\ f))\ (\Phi o\ (S.cod\ f))$
$\langle proof \rangle$

lemma Φ-*ide* [*simp*]:
assumes $S.ide\ a$
shows $\Phi\ a = \Phi o\ a$
$\langle proof \rangle$

lemma *set-dom-*Φ:
assumes $S.arr\ f$
shows $S'.set\ (S'.dom\ (\Phi\ f)) = \varphi\ `\ (S.set\ (S.dom\ f))$
$\langle proof \rangle$

lemma Φ-*comp*:
assumes $S.seq\ g\ f$
shows $\Phi\ (g \cdot f) = \Phi\ g \cdot' \Phi\ f$
$\langle proof \rangle$

interpretation Φ: *functor* $S\ S'\ \Phi$
$\langle proof \rangle$

lemma Φ-*is-functor*:
shows *functor* $S\ S'\ \Phi\ \langle proof \rangle$

lemma *Fun-*Φ:
assumes $S.arr\ f$ **and** $x \in S.set\ (S.dom\ f)$
shows $S'.Fun\ (\Phi\ f)\ (\varphi\ x) = \Phi a\ f\ (\varphi\ x)$
$\langle proof \rangle$

lemma Φ-*acts-elementwise*:
assumes $S.ide\ a$
shows $S'.set\ (\Phi\ a) = \Phi\ `\ S.set\ a$
$\langle proof \rangle$

lemma Φ-*preserves-incl*:
assumes $S.incl\ m$
shows $S'.incl\ (\Phi\ m)$
$\langle proof \rangle$

lemma ψ-*respects-sets*:
assumes $A' \subseteq S'.Univ$
shows *setp* $(\psi$ ‘ $A') \longleftrightarrow$ *setp'* A'
⟨*proof*⟩

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

interpretation *INV*: *two-set-categories-bij-betw-Univ* S' *setp'* S *setp* ψ
⟨*proof*⟩

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
⟨*proof*⟩

The functors Φ and Ψ are inverses.

lemma *Fun*-Ψ:
assumes $S'.arr\ f'$ **and** $x' \in S'.set\ (S'.dom\ f')$
shows $S.Fun\ (\Psi\ f')\ (\psi\ x') = \Psi a\ f'\ (\psi\ x')$
⟨*proof*⟩

lemma Ψo-Φo:
assumes $S.ide\ a$
shows $\Psi o\ (\Phi o\ a) = a$
⟨*proof*⟩

lemma $\Phi\Psi$:
assumes $S.arr\ f$
shows $\Psi\ (\Phi\ f) = f$
⟨*proof*⟩

lemma Φo-Ψo:
assumes $S'.ide\ a'$
shows $\Phi o\ (\Psi o\ a') = a'$
⟨*proof*⟩

lemma $\Psi\Phi$:
assumes $S'.arr\ f'$
shows $\Phi\ (\Psi\ f') = f'$
⟨*proof*⟩

lemma *inverse-functors*-Φ-Ψ:
shows *inverse-functors* S S' Ψ Φ

⟨*proof*⟩

lemma *are-isomorphic*:
shows $\exists\,\Phi.$ *invertible-functor S S′* $\Phi \wedge (\forall\, m.$ *S.incl m* $\longrightarrow$ *S′.incl* $(\Phi\ m))$
⟨*proof*⟩

end

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

theorem *set-category-is-categorical*:
assumes *set-category S setp* **and** *set-category S′ setp′*
and *bij-betw* φ (*set-category-data.Univ S*) (*set-category-data.Univ S′*)
and $\bigwedge A.\ A \subseteq$ *set-category-data.Univ S* $\Longrightarrow$ *setp′* $(\varphi$ ‘ $A) \longleftrightarrow$ *setp A*
shows $\exists\,\Phi.$ *invertible-functor S S′* $\Phi \wedge$
$(\forall\, m.$ *set-category.incl S setp m* $\longrightarrow$ *set-category.incl S′ setp′* $(\Phi\ m))$
⟨*proof*⟩

9.4 Further Properties of Set Categories

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

context *set-category*
begin

abbreviation *Dom*
where *Dom f* $\equiv$ *set* (*dom f*)

abbreviation *Cod*
where *Cod f* $\equiv$ *set* (*cod f*)

9.4.1 Initial Object

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

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

lemma *initial-empty*:
shows *initial empty*
⟨*proof*⟩

9.4.2 Identity Arrows

Identity arrows correspond to restrictions of the identity function.

lemma *ide-char*$_{SC}$:
assumes *arr f*
shows *ide f* ⟷ *Dom f* = *Cod f* ∧ *Fun f* = (λ*x* ∈ *Dom f*. *x*)
⟨*proof*⟩

lemma *ideI*:
assumes *arr f* **and** *Dom f* = *Cod f* **and** ⋀*x*. *x* ∈ *Dom f* ⟹ *Fun f x* = *x*
shows *ide f*
⟨*proof*⟩

9.4.3 Inclusions

lemma *ide-implies-incl*:
assumes *ide a*
shows *incl a*
⟨*proof*⟩

definition *incl-in* :: *'s* ⇒ *'s* ⇒ *bool*
where *incl-in a b* = (*ide a* ∧ *ide b* ∧ *set a* ⊆ *set b*)

abbreviation *incl-of*
where *incl-of a b* ≡ *mkArr* (*set a*) (*set b*) (λ*x*. *x*)

lemma *elem-set-implies-set-eq-singleton*:
assumes *a* ∈ *set b*
shows *set a* = {*a*}
⟨*proof*⟩

lemma *elem-set-implies-incl-in*:
assumes *a* ∈ *set b*
shows *incl-in a b*
⟨*proof*⟩

lemma *incl-incl-of* [*simp*]:
assumes *incl-in a b*
shows *incl* (*incl-of a b*)
and «*incl-of a b* : *a* → *b*»
⟨*proof*⟩

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'*
⟨*proof*⟩

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* · *f*)
⟨*proof*⟩

9.4.4 Image Factorization

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

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

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

lemma *ide-img* [*simp*]:
assumes *arr f*
shows *ide* (*img f*)
⟨*proof*⟩

lemma *set-img* [*simp*]:
assumes *arr f*
shows *set* (*img f*) = *Img f*
⟨*proof*⟩

lemma *img-point-in-Univ*:
assumes «*x* : *unity* → *a*»
shows *img x* ∈ *Univ*
⟨*proof*⟩

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

lemma *img-point-elem-set*:
assumes «*x* : *unity* → *a*»
shows *img x* ∈ *set a*
⟨*proof*⟩

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

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

lemma *corestr-in-hom*:
assumes *arr f*
shows «*corestr f* : *dom f* → *img f*»
⟨*proof*⟩

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*⟩

lemma *Fun-corestr*:
assumes *arr f*
shows *Fun* (*corestr f*) = *Fun f*
⟨*proof*⟩

9.4.5 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* ∈ *set a*
shows «*mkPoint a t* : *unity* → *a*»
⟨*proof*⟩

lemma *Fun-mkPoint*:
assumes *ide a* **and** *t* ∈ *set a*
shows *Fun* (*mkPoint a t*) = (λ- ∈ {*unity*}. *t*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *setp-img-points*:
assumes *ide a*
shows *setp* (*img* ‘ *hom unity a*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *comp-arr-mkPoint*:
assumes *arr f* **and** t ∈ *Dom f*
shows f · *mkPoint* (*dom f*) t = *mkPoint* (*cod f*) (*Fun f t*)
⟨*proof*⟩

lemma *comp-arr-point*$_{SSC}$:
assumes *arr f* **and** «x : *unity* → *dom f*»
shows f · x = *mkPoint* (*cod f*) (*Fun f* (*img* x))
⟨*proof*⟩

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*⟩

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

lemma *arr-eqI'*$_{SC}$:
assumes *par f f'* **and** ⋀x. «x : *unity* → *dom f*» ⟹ $f \cdot x = f' \cdot x$
shows $f = f'$
⟨*proof*⟩

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

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

lemma *mkArr'-in-hom*:
assumes *ide a* **and** *ide b* **and** F ∈ *hom unity a* → *hom unity b*
shows «*mkArr' a b F* : a → b»
⟨*proof*⟩

lemma *comp-point-mkArr'*:
assumes *ide a* **and** *ide b* **and** F ∈ *hom unity a* → *hom unity b*
shows ⋀x. «x : *unity* → a» ⟹ *mkArr' a b F* · x = $F\ x$
⟨*proof*⟩

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

lemma *terminal-char3*:
assumes $\exists!x.$ «$x : unity \to a$»
shows *terminal a*
⟨*proof*⟩

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 $\exists!f.$ «$f : a \to b$» $\wedge$ ($\forall x.$ «$x : unity \to a$» $\longrightarrow f \cdot x = F\ x$)
⟨*proof*⟩

9.4.6 The 'Determines Same Function' Relation on Arrows

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

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

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 $Fun\ f = Fun\ f' \longleftrightarrow corestr\ f = corestr\ f'$
⟨*proof*⟩

9.4.7 Retractions, Sections, and Isomorphisms

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

lemma *retraction-if-Img-eq-Cod*:
assumes *arr g* **and** $Img\ g = Cod\ g$
shows *retraction g*
and $ide\ (g \cdot mkArr\ (Cod\ g)\ (Dom\ g)\ (inv\text{-}into\ (Dom\ g)\ (Fun\ g)))$
⟨*proof*⟩

lemma *retraction-char*:
shows $retraction\ g \longleftrightarrow arr\ g \wedge Img\ g = Cod\ g$
⟨*proof*⟩

Every corestriction is a retraction.

lemma *retraction-corestr*:

assumes *arr f*
shows *retraction* (*corestr f*)
⟨*proof*⟩

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*⟩

lemma *section-char*:
shows *section f* ⟷ *arr f* ∧ (*Dom f* = {} ⟶ *Cod f* = {}) ∧ *inj-on* (*Fun f*) (*Dom f*)
⟨*proof*⟩

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

lemma *section-retraction-char*:
shows *ide* (*g* · *f*) ⟷ *antipar f g* ∧ *compose* (*Dom f*) (*Fun g*) (*Fun f*) = (λ*x* ∈ *Dom f*. *x*)
⟨*proof*⟩

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

lemma *inverse-arrows-char*:
shows *inverse-arrows f g* ⟷
antipar f g ∧ *compose* (*Dom f*) (*Fun g*) (*Fun f*) = (λ*x* ∈ *Dom f*. *x*)
∧ *compose* (*Dom g*) (*Fun f*) (*Fun g*) = (λ*y* ∈ *Dom g*. *y*)
⟨*proof*⟩

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

lemma *iso-char*:
shows *iso f* ⟷ *arr f* ∧ *bij-betw* (*Fun f*) (*Dom f*) (*Cod f*)
⟨*proof*⟩

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*⟩

lemma *Fun-inv*:
assumes *iso f*
shows *Fun* (*inv f*) = *restrict* (*inv-into* (*Dom f*) (*Fun f*)) (*Cod f*)
⟨*proof*⟩

9.4.8 Monomorphisms and Epimorphisms

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

lemma *mono-char*:
shows *mono f* $\longleftrightarrow$ *arr f* $\wedge$ *inj-on* (*Fun f*) (*Dom f*)
⟨*proof*⟩

Inclusions are monomorphisms.

lemma *mono-imp-incl*:
assumes *incl f*
shows *mono f*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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** $\exists t\ t'.\ t \in Univ \wedge t' \in Univ \wedge t \neq t'$
shows *retraction f*
⟨*proof*⟩

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*⟩

end

9.5 Concrete Set Categories

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

locale *concrete-set-category* = *set-category S setp*
  for *S* :: *'s comp* (**infixr** $\cdot_S$ *55*)
  and *setp* :: *'s set* ⇒ *bool*
  and *U* :: *'a set*
  and ι :: *'a* ⇒ *'s* +
  assumes *UP-mapsto*: ι ∈ *U* → *Univ*
  and *inj-UP*: *inj-on* ι *U*
begin

  abbreviation *UP*
  where *UP* ≡ ι

  abbreviation *DN*
  where *DN* ≡ *inv-into U UP*

  lemma *DN-mapsto*:
  shows *DN* ∈ *UP* ' *U* → *U*
    ⟨*proof*⟩

  lemma *DN-UP* [*simp*]:
  assumes *x* ∈ *U*
  shows *DN* (*UP x*) = *x*
    ⟨*proof*⟩

  lemma *UP-DN* [*simp*]:
  assumes *t* ∈ *UP* ' *U*
  shows *UP* (*DN t*) = *t*
    ⟨*proof*⟩

  lemma *bij-UP*:
  shows *bij-betw UP U* (*UP* ' *U*)
    ⟨*proof*⟩

  lemma *bij-DN*:
  shows *bij-betw DN* (*UP* ' *U*) *U*
    ⟨*proof*⟩

end

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

and U :: $'a$ *set*
and UP :: $'a \Rightarrow 's$

9.6 Sub-Set Categories

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

locale *sub-set-category* =
S: *set-category* +
fixes *ssetp* :: $'a$ *set* $\Rightarrow$ *bool*
assumes *ssetp-singleton*: $\bigwedge t.\ t \in S.Univ \Longrightarrow ssetp\ \{t\}$
and *subset-closed*: $\bigwedge B\ A.\ [\![B \subseteq A;\ ssetp\ A]\!] \Longrightarrow ssetp\ B$
and *union-closed*: $\bigwedge A\ B.\ [\![ssetp\ A;\ ssetp\ B]\!] \Longrightarrow ssetp\ (A \cup B)$
and *containment*: $\bigwedge A.\ ssetp\ A \Longrightarrow setp\ A$
begin

sublocale *full-subcategory* *S* ‹$\lambda a.\ S.ide\ a \wedge ssetp\ (S.set\ a)$›
⟨*proof*⟩

lemma *is-full-subcategory*:
shows *full-subcategory* *S* $(\lambda a.\ S.ide\ a \wedge ssetp\ (S.set\ a))$
⟨*proof*⟩

lemma *ide-char*$_{SSC}$:
shows $ide\ a \longleftrightarrow S.ide\ a \wedge ssetp\ (S.set\ a)$
⟨*proof*⟩

lemma *terminal-unity*$_{SSC}$:
shows *terminal* $S.unity$
⟨*proof*⟩

lemma *terminal-char*:
shows $terminal\ t \longleftrightarrow S.terminal\ t$
⟨*proof*⟩

sublocale *set-category comp ssetp*
⟨*proof*⟩

lemma *is-set-category*:
shows *set-category comp ssetp*
⟨*proof*⟩

end

end

Chapter 10

SetCat

theory *SetCat*
imports *SetCategory ConcreteCategory*
begin

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

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

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

lemma *finite-imp-Setp*: *finite A* ⟹ *Setp A*
⟨*proof*⟩

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

interpretation *S*: *concrete-category* ‹*Collect Setp*› ‹λ*A B*. *extensional A* ∩ (*A* → *B*)›
‹λ*A*. λ*x* ∈ *A*. *x*› ‹λ*C B A g f*. *compose A g f*›
⟨*proof*⟩

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

lemma *is-category*:
shows *category comp*
⟨*proof*⟩

lemma *MkArr-expansion*:
assumes *S.arr f*
shows $f = S.MkArr\ (S.Dom\ f)\ (S.Cod\ f)\ (\lambda x \in S.Dom\ f.\ S.Map\ f\ x)$
⟨*proof*⟩

lemma *arr-char*:
shows $S.arr\ f \longleftrightarrow f \neq S.Null \land Setp\ (S.Dom\ f) \land Setp\ (S.Cod\ f) \land$
$S.Map\ f \in extensional\ (S.Dom\ f) \cap (S.Dom\ f \rightarrow S.Cod\ f)$
⟨*proof*⟩

lemma *terminal-char*:
shows $S.terminal\ a \longleftrightarrow (\exists\, x.\ a = S.MkIde\ \{x\})$
⟨*proof*⟩

definition *IMG* :: $'e\ setcat.arr \Rightarrow\ 'e\ setcat.arr$
where $IMG\ f = S.MkIde\ (S.Map\ f\ `\ S.Dom\ f)$

interpretation *S*: *set-category-data comp IMG*
⟨*proof*⟩

lemma *terminal-unity*:
shows *S.terminal S.unity*
⟨*proof*⟩

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

definition *arr-of* :: $'e \Rightarrow\ 'e\ setcat.arr$
where $arr\text{-}of\ x \equiv S.MkIde\ \{x\}$

definition *elem-of* :: $'e\ setcat.arr \Rightarrow\ 'e$
where $elem\text{-}of\ t \equiv the\text{-}elem\ (S.Dom\ t)$

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

lemma *arr-of-mapsto*:
shows $arr\text{-}of \in UNIV \rightarrow S.Univ$
⟨*proof*⟩

lemma *elem-of-mapsto*:
shows $elem\text{-}of \in Univ \rightarrow UNIV$
⟨*proof*⟩

lemma *elem-of-arr-of* [*simp*]:
shows *elem-of* (*arr-of* x) = x
⟨*proof*⟩

lemma *arr-of-elem-of* [*simp*]:
assumes $t \in S.Univ$
shows *arr-of* (*elem-of* t) = t
⟨*proof*⟩

lemma *inj-arr-of*:
shows *inj arr-of*
⟨*proof*⟩

lemma *bij-arr-of*:
shows *bij-betw arr-of UNIV S.Univ*
⟨*proof*⟩

lemma *bij-elem-of*:
shows *bij-betw elem-of S.Univ UNIV*
⟨*proof*⟩

lemma *elem-of-img-arr-of-img* [*simp*]:
shows *elem-of* ‘ *arr-of* ‘ $A = A$
⟨*proof*⟩

lemma *arr-of-img-elem-of-img* [*simp*]:
assumes $A \subseteq S.Univ$
shows *arr-of* ‘ *elem-of* ‘ $A = A$
⟨*proof*⟩

lemma *Dom-terminal*:
assumes *S.terminal t*
shows *S.Dom t* = {*elem-of t*}
⟨*proof*⟩

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

lemma *IMG-point*:
assumes «p : *S.unity* → a»
shows *IMG* ∈ *S.hom S.unity a* → *S.Univ*
and *IMG p* = (*arr-of o S.Map p o elem-of*) *S.unity*
⟨*proof*⟩

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

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

lemma *MkElem-in-hom*:
assumes *S.arr f* **and** $x \in S.Dom\ f$
shows «*MkElem* (*arr-of x*) (*S.dom f*) : *S.unity* → *S.dom f*»
⟨*proof*⟩

lemma *MkElem-IMG*:
assumes $p \in S.hom\ S.unity\ a$
shows *MkElem* (*IMG p*) *a* = *p*
⟨*proof*⟩

lemma *inj-IMG*:
assumes *S.ide a*
shows *inj-on IMG* (*S.hom S.unity a*)
⟨*proof*⟩

lemma *set-char*:
assumes *S.ide a*
shows *S.set a* = *arr-of* ' *S.Dom a*
⟨*proof*⟩

lemma *Map-via-comp*:
assumes *S.arr f*
shows *S.Map f* = (λ*x* ∈ *S.Dom f*. *S.Map* (*f* · *MkElem* (*arr-of x*) (*S.dom f*)) *U*)
⟨*proof*⟩

lemma *arr-eqI'*:
assumes *S.par f f'* **and** ⋀*t*. «*t* : *S.unity* → *S.dom f*» ⟹ *f* · *t* = *f'* · *t*
shows *f* = *f'*
⟨*proof*⟩

lemma *Setp-elem-of-img*:
assumes *A* ∈ *S.set* ' *Collect S.ide*
shows *Setp* (*elem-of* ' *A*)
⟨*proof*⟩

lemma *set-MkIde-elem-of-img*:
assumes *A* ⊆ *S.Univ* **and** *S.ide* (*S.MkIde* (*elem-of* ' *A*))
shows *S.set* (*S.MkIde* (*elem-of* ' *A*)) = *A*
⟨*proof*⟩

lemma *set-img-Collect-ide-iff*:
shows *A* ∈ *S.set* ' *Collect S.ide* ⟷ *A* ⊆ *S.Univ* ∧ *Setp* (*elem-of* ' *A*)
⟨*proof*⟩

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* (λ*A*. *A* ⊆ *S.Univ* ∧ *Setp* (*elem-of* ' *A*))

⟨*proof*⟩

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

corollary *is-concrete-set-category*:
shows *concrete-set-category comp* $(\lambda A.\ A \subseteq S.Univ \wedge Setp\ (elem\text{-}of\ `\ A))$ *UNIV arr-of*
⟨*proof*⟩

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

corollary *set-category-iso-SetCat*:
fixes S :: *'s comp* **and** φ :: $'s \Rightarrow 'e$
assumes *set-category S* $\mathcal{S}$
and *bij-betw* φ (*set-category-data.Univ S*) *UNIV*
and $\bigwedge A.\ \mathcal{S}\ A \longleftrightarrow A \subseteq set\text{-}category\text{-}data.Univ\ S \wedge (arr\text{-}of \circ \varphi)\ `\ A \in S.set\ `\ Collect\ S.ide$
shows $\exists\,\Phi.$ *invertible-functor S comp* Φ
$\wedge\ (\forall\, m.\ set\text{-}category.incl\ S\ \mathcal{S}\ m$
$\longrightarrow set\text{-}category.incl\ comp\ (\lambda A.\ A \in S.set\ `\ Collect\ S.ide)\ (\Phi\ m))$
⟨*proof*⟩

sublocale *category comp*
⟨*proof*⟩
sublocale *set-category comp* ‹$\lambda A.\ A \subseteq Collect\ S.terminal \wedge Setp\ (elem\text{-}of\ `\ A)$›
⟨*proof*⟩
interpretation *concrete-set-category comp* ‹$\lambda A.\ A \subseteq Collect\ S.terminal \wedge Setp\ (elem\text{-}of\ `$
$A)$›
UNIV arr-of
⟨*proof*⟩

end

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

context *setcat*
begin

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

interpretation *set-category comp* ‹$\lambda A.\ A \subseteq Collect\ terminal \wedge Setp\ (elem\text{-}of\ `\ A)$›
⟨*proof*⟩
interpretation *concrete-set-category comp* ‹$\lambda A.\ A \subseteq Univ \wedge Setp\ (elem\text{-}of\ `\ A)$› *UNIV arr-of*
⟨*proof*⟩

definition *set-of-ide* :: $'e\ setcat.arr \Rightarrow 'e\ set$
where *set-of-ide a* $\equiv$ *elem-of* ‘ *set a*

definition *ide-of-set* :: $'e$ *set* $\Rightarrow$ $'e$ *setcat.arr*
where *ide-of-set A* $\equiv$ *mkIde* (*arr-of* ‘ *A*)

lemma *bij-betw-ide-set*:
shows *set-of-ide* $\in$ *Collect ide* $\to$ *Collect Setp*
and *ide-of-set* $\in$ *Collect Setp* $\to$ *Collect ide*
and [*simp*]: *ide a* $\Longrightarrow$ *ide-of-set* (*set-of-ide a*) = *a*
and [*simp*]: *Setp A* $\Longrightarrow$ *set-of-ide* (*ide-of-set A*) = *A*
and *bij-betw set-of-ide* (*Collect ide*) (*Collect Setp*)
and *bij-betw ide-of-set* (*Collect Setp*) (*Collect ide*)
⟨*proof*⟩

definition *fun-of-arr* :: $'e$ *setcat.arr* $\Rightarrow$ $'e$ $\Rightarrow$ $'e$
where *fun-of-arr f* $\equiv$ *restrict* (*elem-of o Fun f o arr-of*) (*elem-of* ‘*Dom f*)

definition *arr-of-fun* :: $'e$ *set* $\Rightarrow$ $'e$ *set* $\Rightarrow$ ($'e$ $\Rightarrow$ $'e$) $\Rightarrow$ $'e$ *setcat.arr*
where *arr-of-fun A B F* $\equiv$ *mkArr* (*arr-of* ‘ *A*) (*arr-of* ‘ *B*) (*arr-of o F o elem-of*)

lemma *bij-betw-hom-fun*:
shows *fun-of-arr* $\in$ *hom a b* $\to$ *extensional* (*set-of-ide a*) $\cap$ (*set-of-ide a* $\to$ *set-of-ide b*)
and ⟦*Setp A*; *Setp B*⟧ $\Longrightarrow$ *arr-of-fun A B* $\in$ (*A* $\to$ *B*) $\to$ *hom* (*ide-of-set A*) (*ide-of-set B*)
and *f* $\in$ *hom a b* $\Longrightarrow$ *arr-of-fun* (*set-of-ide a*) (*set-of-ide b*) (*fun-of-arr f*) = *f*
and ⟦*Setp A*; *Setp B*; *F* $\in$ *A* $\to$ *B*; *F* $\in$ *extensional A*⟧ $\Longrightarrow$ *fun-of-arr* (*arr-of-fun A B F*) =
F
and ⟦*ide a*; *ide b*⟧ $\Longrightarrow$ *bij-betw fun-of-arr* (*hom a b*)
(*extensional* (*set-of-ide a*) $\cap$ (*set-of-ide a* $\to$ *set-of-ide b*))
and ⟦*Setp A*; *Setp B*⟧ $\Longrightarrow$
bij-betw (*arr-of-fun A B*)
(*extensional A* $\cap$ (*A* $\to$ *B*)) (*hom* (*ide-of-set A*) (*ide-of-set B*))
⟨*proof*⟩

lemma *fun-of-arr-ide*:
assumes *ide a*
shows *fun-of-arr a* = *restrict id* (*elem-of* ‘ *Dom a*)
⟨*proof*⟩

lemma *arr-of-fun-id*:
assumes *Setp A*
shows *arr-of-fun A A* (*restrict id A*) = *ide-of-set A*
⟨*proof*⟩

lemma *fun-of-arr-comp*:
assumes *f* $\in$ *hom a b* **and** *g* $\in$ *hom b c*
shows *fun-of-arr* (*comp g f*) = *restrict* (*fun-of-arr g* $\circ$ *fun-of-arr f*) (*set-of-ide a*)
⟨*proof*⟩

lemma *arr-of-fun-comp*:
assumes *Setp A* **and** *Setp B* **and** *Setp C*
and *F* $\in$ *extensional A* $\cap$ (*A* $\to$ *B*) **and** *G* $\in$ *extensional B* $\cap$ (*B* $\to$ *C*)

shows *arr-of-fun A C* (*G o F*) = *comp* (*arr-of-fun B C G*) (*arr-of-fun A B F*)
⟨*proof*⟩

end

When there is no restriction on the sets that determine objects, the resulting set category is replete. This is the normal use case, which we want to streamline as much as possible, so it is useful to introduce a special locale for this purpose.

locale *replete-setcat* =
fixes *elem-type* :: *'e itself*
begin

interpretation *SC*: *setcat elem-type* ‹λ-. *True*›
⟨*proof*⟩

definition *comp*
where *comp* ≡ *SC.comp*

definition *arr-of*
where *arr-of* ≡ *SC.arr-of*

definition *elem-of*
where *elem-of* ≡ *SC.elem-of*

sublocale *replete-set-category comp*
⟨*proof*⟩

lemma *is-replete-set-category*:
shows *replete-set-category comp*
⟨*proof*⟩

lemma *is-set-category*$_{RSC}$:
shows *set-category comp* (λ*A*. *A* ⊆ *Univ*)
⟨*proof*⟩

sublocale *concrete-set-category comp setp UNIV arr-of*
⟨*proof*⟩

lemma *is-concrete-set-category*:
shows *concrete-set-category comp setp UNIV arr-of*
⟨*proof*⟩

lemma *bij-arr-of*:
shows *bij-betw arr-of UNIV Univ*
⟨*proof*⟩

lemma *bij-elem-of*:
shows *bij-betw elem-of Univ UNIV*
⟨*proof*⟩

end

end

Chapter 11

ProductCategory

theory *ProductCategory*
imports *Category EpiMonoIso*
begin

This theory defines the product of two categories *C1* and *C2*, which is the category *C* whose arrows are ordered pairs consisting of an arrow of *C1* and an arrow of *C2*, with composition defined componentwise. As the ordered pair (*C1.null*, *C2.null*) is available to serve as *C.null*, we may directly identify the arrows of the product category *C* with ordered pairs, leaving the type of arrows of *C* transparent.

```
locale product-category =
  C1: category C1 +
  C2: category C2
for C1 :: 'a1 comp      (infixr ·1 55)
and C2 :: 'a2 comp      (infixr ·2 55)
begin

  type-synonym ('aa1, 'aa2) arr = 'aa1 * 'aa2

  notation C1.in-hom      («- : - →1 -»)
  notation C2.in-hom      («- : - →2 -»)

  abbreviation (input) Null :: ('a1, 'a2) arr
  where Null ≡ (C1.null, C2.null)

  abbreviation (input) Arr :: ('a1, 'a2) arr ⇒ bool
  where Arr f ≡ C1.arr (fst f) ∧ C2.arr (snd f)

  abbreviation (input) Ide :: ('a1, 'a2) arr ⇒ bool
  where Ide f ≡ C1.ide (fst f) ∧ C2.ide (snd f)

  abbreviation (input) Dom :: ('a1, 'a2) arr ⇒ ('a1, 'a2) arr
  where Dom f ≡ (if Arr f then (C1.dom (fst f), C2.dom (snd f)) else Null)

  abbreviation (input) Cod :: ('a1, 'a2) arr ⇒ ('a1, 'a2) arr
```

where *Cod f* ≡ (*if Arr f then* (*C1.cod* (*fst f*), *C2.cod* (*snd f*)) *else Null*)

definition *comp* :: ('*a1*, '*a2*) *arr* ⇒ ('*a1*, '*a2*) *arr* ⇒ ('*a1*, '*a2*) *arr*
where *comp g f* = (*if Arr f* ∧ *Arr g* ∧ *Cod f* = *Dom g then*
(*C1* (*fst g*) (*fst f*), *C2* (*snd g*) (*snd f*))
else Null)

notation *comp* (**infixr** · *55*)

lemma *not-Arr-Null*:
shows ¬*Arr Null*
⟨*proof*⟩

interpretation *partial-composition comp*
⟨*proof*⟩

notation *in-hom* («- : - → -»)

lemma *null-char* [*simp*]:
shows *null* = *Null*
⟨*proof*⟩

lemma *ide-Ide*:
assumes *Ide a*
shows *ide a*
⟨*proof*⟩

lemma *has-domain-char*:
shows *domains f* ≠ {} ⟷ *Arr f*
⟨*proof*⟩

lemma *has-codomain-char*:
shows *codomains f* ≠ {} ⟷ *Arr f*
⟨*proof*⟩

lemma *arr-char* [*iff*]:
shows *arr f* ⟷ *Arr f*
⟨*proof*⟩

lemma $arrI_{PC}$ [*intro*]:
assumes *C1.arr f1* **and** *C2.arr f2*
shows *arr* (*f1*, *f2*)
⟨*proof*⟩

lemma *arrE*:
assumes *arr f*
and *C1.arr* (*fst f*) ∧ *C2.arr* (*snd f*) ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *seqI*$_{PC}$ [*intro*]:
assumes *C1.seq g1 f1* ∧ *C2.seq g2 f2*
shows *seq* (*g1*, *g2*) (*f1*, *f2*)
⟨*proof*⟩

lemma *seqE*$_{PC}$ [*elim*]:
assumes *seq g f*
and *C1.seq* (*fst g*) (*fst f*) ⟹ *C2.seq* (*snd g*) (*snd f*) ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *seq-char* [*iff*]:
shows *seq g f* ⟷ *C1.seq* (*fst g*) (*fst f*) ∧ *C2.seq* (*snd g*) (*snd f*)
⟨*proof*⟩

lemma *Dom-comp*:
assumes *seq g f*
shows *Dom* (*g* · *f*) = *Dom f*
⟨*proof*⟩

lemma *Cod-comp*:
assumes *seq g f*
shows *Cod* (*g* · *f*) = *Cod g*
⟨*proof*⟩

theorem *is-category*:
shows *category comp*
⟨*proof*⟩

sublocale *category comp*
⟨*proof*⟩

lemma *dom-char*:
shows *dom f* = *Dom f*
⟨*proof*⟩

lemma *dom-simp* [*simp*]:
assumes *arr f*
shows *dom f* = (*C1.dom* (*fst f*), *C2.dom* (*snd f*))
⟨*proof*⟩

lemma *cod-char*:
shows *cod f* = *Cod f*
⟨*proof*⟩

lemma *cod-simp* [*simp*]:
assumes *arr f*
shows *cod f* = (*C1.cod* (*fst f*), *C2.cod* (*snd f*))

⟨*proof*⟩

lemma *in-homI*$_{PC}$ [*intro*, *simp*]:
assumes «*fst f*: *fst a* $\rightarrow_1$ *fst b*» **and** «*snd f*: *snd a* $\rightarrow_2$ *snd b*»
shows «*f*: *a* → *b*»
⟨*proof*⟩

lemma *in-homE*$_{PC}$ [*elim*]:
assumes «*f*: *a* → *b*»
and «*fst f*: *fst a* $\rightarrow_1$ *fst b*» ⟹ «*snd f*: *snd a* $\rightarrow_2$ *snd b*» ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *ide-char*$_{PC}$ [*iff*]:
shows *ide f* ⟷ *Ide f*
⟨*proof*⟩

lemma *comp-char*:
shows *g* · *f* = (*if C1.arr* (*C1* (*fst g*) (*fst f*)) ∧ *C2.arr* (*C2* (*snd g*) (*snd f*)) *then*
(*C1* (*fst g*) (*fst f*), *C2* (*snd g*) (*snd f*))
else Null)
⟨*proof*⟩

lemma *comp-simp* [*simp*]:
assumes *C1.seq* (*fst g*) (*fst f*) **and** *C2.seq* (*snd g*) (*snd f*)
shows *g* · *f* = (*fst g* $\cdot_1$ *fst f*, *snd g* $\cdot_2$ *snd f*)
⟨*proof*⟩

lemma *iso-char* [*iff*]:
shows *iso f* ⟷ *C1.iso* (*fst f*) ∧ *C2.iso* (*snd f*)
⟨*proof*⟩

lemma *isoI*$_{PC}$ [*intro*, *simp*]:
assumes *C1.iso* (*fst f*) **and** *C2.iso* (*snd f*)
shows *iso f*
⟨*proof*⟩

lemma *isoD*:
assumes *iso f*
shows *C1.iso* (*fst f*) **and** *C2.iso* (*snd f*)
⟨*proof*⟩

lemma *inv-simp* [*simp*]:
assumes *iso f*
shows *inv f* = (*C1.inv* (*fst f*), *C2.inv* (*snd f*))
⟨*proof*⟩

end

end

Chapter 12

NaturalTransformation

theory *NaturalTransformation*
imports *Functor*
begin

12.1 Definition of a Natural Transformation

As is the case for functors, the "object-free" definition of category makes it possible to view natural transformations as functions on arrows. In particular, a natural transformation between functors F and G from A to B can be represented by the map that takes each arrow f of A to the diagonal of the square in B corresponding to the transformation of $F\,f$ to $G\,f$. The images of the identities of A under this map are the usual components of the natural transformation. This representation exhibits natural transformations as a kind of generalization of functors, and in fact we can directly identify functors with identity natural transformations. However, functors are still necessary to state the defining conditions for a natural transformation, as the domain and codomain of a natural transformation cannot be recovered from the map on arrows that represents it.

Like functors, natural transformations preserve arrows and map non-arrows to null. Natural transformations also "preserve" domain and codomain, but in a more general sense than functors. The naturality conditions, which express the two ways of factoring the diagonal of a commuting square, are degenerate in the case of an identity transformation.

locale *natural-transformation* =
  A: *category A* +
  B: *category B* +
  F: *functor A B F* +
  G: *functor A B G*
for A :: *'a comp* (**infixr** $\cdot_A$ *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 $\tau\ (A.cod\ f) \cdot_B F\ f = G\ f \cdot_B \tau\ (A.dom\ f)$
⟨*proof*⟩

The following fact for natural transformations provides us with the same advantages as the corresponding fact for functors.

lemma *preserves-reflects-arr* [*iff*]:
shows $B.arr\ (\tau\ f) \longleftrightarrow A.arr\ f$
⟨*proof*⟩

lemma *preserves-hom* [*intro*]:
assumes «$f : a \rightarrow_A b$»
shows «$\tau\ f : F\ a \rightarrow_B G\ b$»
⟨*proof*⟩

lemma *preserves-comp-1*:
assumes $A.seq\ f'\ f$
shows $\tau\ (f' \cdot_A f) = G\ f' \cdot_B \tau\ f$
⟨*proof*⟩

lemma *preserves-comp-2*:
assumes $A.seq\ f'\ f$
shows $\tau\ (f' \cdot_A f) = \tau\ f' \cdot_B F\ f$
⟨*proof*⟩

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*⟩

lemma *functor-implies-equals-cod*:
assumes *functor* $A\ B\ \tau$
shows $G = \tau$
⟨*proof*⟩

end

12.2 Components of a Natural Transformation

The values taken by a natural transformation on identities are the *components* of the transformation. We have the following basic technique for proving two natural transformations equal: show that they have the same components.

lemma *eqI*:
assumes *natural-transformation A B F G* σ **and** *natural-transformation A B F G* σ'
and $\bigwedge a.$ *partial-composition.ide A a* $\Longrightarrow \sigma\ a = \sigma'\ a$
shows $\sigma = \sigma'$
⟨*proof*⟩

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*
    ⟨*proof*⟩

  lemma *is-natural-transformation*:
  shows *natural-transformation A B F G map*
    ⟨*proof*⟩

end

sublocale *transformation-by-components* ⊆ *natural-transformation A B F G map*
  ⟨*proof*⟩

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*⟩

12.3 Functors as Natural Transformations

A functor is a special case of a natural transformation, in the sense that the same map that defines the functor also defines an identity natural transformation.

lemma *functor-is-transformation* [*simp*]:
assumes *functor A B F*
shows *natural-transformation A B F F F*
⟨*proof*⟩

sublocale *functor* ⊆ *as-nat-trans*: *natural-transformation A B F F F*
⟨*proof*⟩

12.4 Constant Natural Transformations

A constant natural transformation is one whose components are all the same arrow.

locale *constant-transformation* =
A: *category A* +
B: *category B* +
F: *constant-functor A B B.dom g* +
G: *constant-functor A B B.cod g*
for *A* :: *'a comp* (**infixr** $\cdot_A$ *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*
⟨*proof*⟩

lemma *is-natural-transformation*:
shows *natural-transformation A B F.map G.map map*
⟨*proof*⟩

lemma *is-functor-if-value-is-ide*:
assumes *B.ide g*
shows *functor A B map*
⟨*proof*⟩

end

sublocale *constant-transformation* ⊆ *natural-transformation A B F.map G.map map*
⟨*proof*⟩

context *constant-transformation*
begin

lemma *equals-dom-if-value-is-ide*:
assumes *B.ide g*
shows *map* = *F.map*
⟨*proof*⟩

lemma *equals-cod-if-value-is-ide*:
assumes *B.ide g*
shows *map* = *G.map*
⟨*proof*⟩

end

12.5 Vertical Composition

Vertical composition is a way of composing natural transformations σ: $F \to G$ and τ: $G \to H$, between parallel functors F, G, and H to obtain a natural transformation from F to H. The composite is traditionally denoted by $\tau\ o\ \sigma$, however in the present setting this notation is misleading because it is horizontal composite, rather than vertical composite, that coincides with composition of natural transformations as functions on arrows.

locale *vertical-composite* =
A: *category A* +
B: *category B* +
F: *functor A B F* +
G: *functor A B G* +
H: *functor A B H* +
σ: *natural-transformation A B F G* σ +
τ: *natural-transformation A B G H* τ
for *A* :: *'a comp* (**infixr** $\cdot_A$ *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*)
⟨*proof*⟩

lemma *map-simp-ide*:
assumes *A.ide a*
shows *map a* = τ *a* $\cdot_B$ σ *a*
⟨*proof*⟩

lemma *map-simp-1*:
assumes *A.arr f*
shows *map f* = τ (*A.cod f*) $\cdot_B$ σ *f*
⟨*proof*⟩

lemma *map-simp-2*:
assumes *A.arr f*
shows *map f* = τ *f* $\cdot_B$ σ (*A.dom f*)
⟨*proof*⟩

lemma *is-natural-transformation*:
shows *natural-transformation A B F H map*
⟨*proof*⟩

end

sublocale *vertical-composite* ⊆ *natural-transformation A B F H map*
⟨*proof*⟩

Functors are the identities for vertical composition.

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

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

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*⟩

12.6 Natural Isomorphisms

A natural isomorphism is a natural transformation each of whose components is an isomorphism. Equivalently, a natural isomorphism is a natural transformation that is invertible with respect to vertical composition.

locale *natural-isomorphism* = *natural-transformation* $A\ B\ F\ G\ \tau$
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

lemma *inv-naturality*:
assumes $A.arr\ f$
shows $F\ f \cdot_B B.inv\ (\tau\ (A.dom\ f)) = B.inv\ (\tau\ (A.cod\ f)) \cdot_B G\ f$
⟨*proof*⟩

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 $B.iso\ (\tau\ f)$
⟨*proof*⟩

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* ⊆ *as-nat-iso*: *natural-isomorphism* $A\ B\ F\ F\ F$
⟨*proof*⟩

definition *naturally-isomorphic*
where *naturally-isomorphic* $A\ B\ F\ G = (\exists \tau.\ \textit{natural-isomorphism}\ A\ B\ F\ G\ \tau)$

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*⟩

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*⟩

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* ⇒ *'b*
and *G* :: *'a* ⇒ *'b*
and *τ* :: *'a* ⇒ *'b*
begin

interpretation *τ'*: *transformation-by-components A B G F* ‹*λa. B.inv* (*τ a*)›
⟨*proof*⟩

definition *map*
where *map* = *τ'.map*

lemma *map-ide-simp* [*simp*]:
assumes *A.ide a*
shows *map a* = *B.inv* (*τ a*)
⟨*proof*⟩

lemma *map-simp*:
assumes *A.arr f*
shows *map f* = *B.inv* (*τ* (*A.cod f*)) $\cdot_B$ *G f*
⟨*proof*⟩

lemma *is-natural-transformation*:
shows *natural-transformation A B G F map*
⟨*proof*⟩

lemma *inverts-components*:
assumes *A.ide a*
shows *B.inverse-arrows* (*τ a*) (*map a*)
⟨*proof*⟩

end

sublocale *inverse-transformation* ⊆ *natural-transformation A B G F map*
⟨*proof*⟩

sublocale *inverse-transformation* ⊆ *natural-isomorphism A B G F map*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩
sublocale *inverse-transformations* ⊆ *natural-isomorphism A B G F* τ'
⟨*proof*⟩

lemma *inverse-transformations-sym*:
assumes *inverse-transformations A B F G* σ σ'
shows *inverse-transformations A B G F* σ' σ
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *naturally-isomorphic-reflexive*:
assumes *functor A B F*
shows *naturally-isomorphic A B F F*
⟨*proof*⟩

lemma *naturally-isomorphic-symmetric*:
assumes *naturally-isomorphic A B F G*
shows *naturally-isomorphic A B G F*
⟨*proof*⟩

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*⟩

12.7 Horizontal Composition

Horizontal composition is a way of composing parallel natural transformations σ from F to G and τ from H to K, where functors F and G map A to B and H and K map B to C, to obtain a natural transformation from $H \circ F$ to $K \circ G$.

Since horizontal composition turns out to coincide with ordinary composition of natural transformations as functions, there is little point in defining a cumbersome locale for horizontal composite.

lemma *horizontal-composite*:
assumes *natural-transformation A B F G* σ
and *natural-transformation B C H K* τ
shows *natural-transformation A C* (*H o F*) (*K o G*) (τ *o* σ)
⟨*proof*⟩

lemma *hcomp-ide-dom* [*simp*]:
assumes *natural-transformation A B F G* τ
shows τ *o* (*identity-functor.map A*) = τ
⟨*proof*⟩

lemma *hcomp-ide-cod* [*simp*]:
assumes *natural-transformation A B F G* τ
shows (*identity-functor.map B*) *o* τ = τ

⟨*proof*⟩

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*⟩

Horizontal composition of a vertical composite with a functor.

lemma *whisker-left*:
assumes *functor B C K*
and *natural-transformation A B F G* τ **and** *natural-transformation A B G H* τ'
shows *K o* (*vertical-composite.map A B* τ τ') = *vertical-composite.map A C* (*K o* τ) (*K o* τ')
⟨*proof*⟩

The interchange law for horizontal and vertical composition.

lemma *interchange*:
assumes *natural-transformation B C F G* τ **and** *natural-transformation B C G H* ν
and *natural-transformation C D K L* σ **and** *natural-transformation C D L M* μ
shows *vertical-composite.map C D* σ μ $\circ$ *vertical-composite.map B C* τ ν =
vertical-composite.map B D ($\sigma \circ \tau$) ($\mu \circ \nu$)
⟨*proof*⟩

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 $\tau \circ \sigma$ = *vertical-composite.map B D* (*H o* σ) (τ *o G*)
and $\tau \circ \sigma$ = *vertical-composite.map B D* (τ *o F*) (*K o* σ)
⟨*proof*⟩

end

Chapter 13

BinaryFunctor

theory *BinaryFunctor*
imports *ProductCategory NaturalTransformation*
begin

This theory develops various properties of binary functors, which are functors defined on product categories.

locale *binary-functor* =
  A1: *category A1* +
  A2: *category A2* +
  B: *category B* +
  A1xA2: *product-category A1 A2* +
  functor A1xA2.comp B F
for *A1* :: *'a1 comp* (**infixr** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp* (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_B$ *55*)
and *F* :: *'a1* ∗ *'a2* ⇒ *'b*
begin

  notation *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** $\cdot_{B1}$ *55*)
and *B2* :: *'b2 comp* (**infixr** $\cdot_{B2}$ *55*)
and *F1* :: *'a1* ⇒ *'b1*
and *F2* :: *'a2* ⇒ *'b2*
begin

notation *A1xA2.comp* (**infixr** $\cdot_{A1xA2}$ *55*)
notation *B1xB2.comp* (**infixr** $\cdot_{B1xB2}$ *55*)
notation *A1.in-hom* («- : - $\rightarrow_{A1}$ -»)
notation *A2.in-hom* («- : - $\rightarrow_{A2}$ -»)
notation *B1.in-hom* («- : - $\rightarrow_{B1}$ -»)
notation *B2.in-hom* («- : - $\rightarrow_{B2}$ -»)
notation *A1xA2.in-hom* («- : - $\rightarrow_{A1xA2}$ -»)
notation *B1xB2.in-hom* («- : - $\rightarrow_{B1xB2}$ -»)

definition *map*
where *map f* = (*if A1.arr* (*fst f*) ∧ *A2.arr* (*snd f*)
then (*F1* (*fst f*), *F2* (*snd f*)) *else* (*F1 A1.null*, *F2 A2.null*))

lemma *map-simp* [*simp*]:
assumes *A1xA2.arr f*
shows *map f* = (*F1* (*fst f*), *F2* (*snd f*))
⟨*proof*⟩

lemma *is-functor*:
shows *functor A1xA2.comp B1xB2.comp map*
⟨*proof*⟩

end

sublocale *product-functor* ⊆ *functor A1xA2.comp B1xB2.comp map*
⟨*proof*⟩
sublocale *product-functor* ⊆ *binary-functor A1 A2 B1xB2.comp map* ⟨*proof*⟩

The following locale is concerned with a binary functor from a category to itself. It defines related functors that are useful when considering monoidal structure on a category.

locale *binary-endofunctor* =
C: *category C* +
CC: *product-category C C* +
CCC: *product-category C CC.comp* +
binary-functor C C C T
for *C* :: *'a comp* (**infixr** · *55*)
and *T* :: *'a* ∗ *'a* ⇒ *'a*
begin

definition *ToTC*
where *ToTC f* ≡ *if CCC.arr f then T* (*T* (*fst f*, *fst* (*snd f*)), *snd* (*snd f*)) *else C.null*

lemma *functor-ToTC*:
shows *functor CCC.comp C ToTC*
⟨*proof*⟩

lemma *ToTC-simp* [*simp*]:
assumes *C.arr f* **and** *C.arr g* **and** *C.arr h*
shows *ToTC* (*f*, *g*, *h*) = *T* (*T* (*f*, *g*), *h*)
⟨*proof*⟩

definition *ToCT*
where *ToCT f* ≡ *if CCC.arr f then T* (*fst f*, *T* (*fst* (*snd f*), *snd* (*snd f*))) *else C.null*

lemma *functor-ToCT*:
shows *functor CCC.comp C ToCT*
⟨*proof*⟩

lemma *ToCT-simp* [*simp*]:
assumes *C.arr f* **and** *C.arr g* **and** *C.arr h*
shows *ToCT* (*f*, *g*, *h*) = *T* (*f*, *T* (*g*, *h*))
⟨*proof*⟩

end

A symmetry functor is a binary functor that exchanges its two arguments.

locale *symmetry-functor* =
A1: *category A1* +
A2: *category A2* +
A1xA2: *product-category A1 A2* +
A2xA1: *product-category A2 A1*
for *A1* :: *'a1 comp* (**infixr** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp* (**infixr** $\cdot_{A2}$ *55*)
begin

notation *A1xA2.comp* (**infixr** $\cdot_{A1xA2}$ *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*)
⟨*proof*⟩

lemma *is-functor*:
shows *functor A1xA2.comp A2xA1.comp map*
⟨*proof*⟩

end

sublocale *symmetry-functor* ⊆ *functor A1xA2.comp A2xA1.comp map*
⟨*proof*⟩
sublocale *symmetry-functor* ⊆ *binary-functor A1 A2 A2xA1.comp map* ⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *fixing-ide-gives-functor-2*:
assumes *A2.ide a2*
shows *functor A1 B* (λ*f1*. *F* (*f1*, *a2*))
⟨*proof*⟩

Fixing one or the other argument of a binary functor to be an arrow yields a natural transformation.

lemma *fixing-arr-gives-natural-transformation-1*:
assumes *A1.arr f1*
shows *natural-transformation A2 B* (λ*f2*. *F* (*A1.dom f1*, *f2*)) (λ*f2*. *F* (*A1.cod f1*, *f2*))
(λ*f2*. *F* (*f1*, *f2*))
⟨*proof*⟩

lemma *fixing-arr-gives-natural-transformation-2*:
assumes *A2.arr f2*
shows *natural-transformation A1 B* (λ*f1*. *F* (*f1*, *A2.dom f2*)) (λ*f1*. *F* (*f1*, *A2.cod f2*))
(λ*f1*. *F* (*f1*, *f2*))
⟨*proof*⟩

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⟩

lemma *preserves-comp-2*:
assumes *A2.seq f2′ f2*
shows (*λf1. F* (*f1*, *f2′* $\cdot_{A2}$ *f2*)) =
vertical-composite.map A1 B (*λf1. F* (*f1*, *f2*)) (*λf1. F* (*f1*, *f2′*))
⟨proof⟩

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** $\cdot_{A1}$ *55*)
and *A2* :: *′a2 comp* (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *′b comp* (**infixr** $\cdot_B$ *55*)
and *F* :: *′a1* ∗ *′a2* ⇒ *′b*
and *G* :: *′a1* ∗ *′a2* ⇒ *′b*
and *τ* :: *′a1* ∗ *′a2* ⇒ *′b*
begin

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

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⟩

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⟩

end

end

Chapter 14

FunctorCategory

theory *FunctorCategory*
imports *ConcreteCategory BinaryFunctor*
begin

The functor category $[A, B]$ is the category whose objects are functors from A to B and whose arrows correspond to natural transformations between these functors.

14.1 Construction

Since the arrows of a functor category cannot (in the context of the present development) be directly identified with natural transformations, but rather only with natural transformations that have been equipped with their domain and codomain functors, and since there is no natural value to serve as *null*, we use the general-purpose construction given by *concrete-category* to define this category.

locale *functor-category* =
  A: *category A* +
  B: *category B*
for *A* :: *'a comp*   (**infixr** $\cdot_A$ *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*›
    ⟨*proof*⟩

  lemma *is-concrete-category*:
  shows *concrete-category* (*Collect* (*functor A B*))

($\lambda F\ G.\ Collect\ (natural\text{-}transformation\ A\ B\ F\ G)$) ($\lambda F.\ F$)
($\lambda F\ G\ H\ \tau\ \sigma.\ vertical\text{-}composite.map\ A\ B\ \sigma\ \tau$)
⟨*proof*⟩

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

lemma *is-category*:
shows *category comp*
⟨*proof*⟩

lemma *arrI* [*intro*]:
assumes $f \neq null$ **and** *natural-transformation A B (Dom f) (Cod f) (Map f)*
shows *arr f*
⟨*proof*⟩

lemma *arrE* [*elim*]:
assumes *arr f*
and $f \neq null \Longrightarrow natural\text{-}transformation\ A\ B\ (Dom\ f)\ (Cod\ f)\ (Map\ f) \Longrightarrow T$
shows *T*
⟨*proof*⟩

lemma *arr-MkArr* [*iff*]:
shows $arr\ (MkArr\ F\ G\ \tau) \longleftrightarrow natural\text{-}transformation\ A\ B\ F\ G\ \tau$
⟨*proof*⟩

lemma *ide-char* [*iff*]:
shows $ide\ t \longleftrightarrow t \neq null \wedge functor\ A\ B\ (Map\ t) \wedge Dom\ t = Map\ t \wedge Cod\ t = Map\ t$
⟨*proof*⟩

end

14.2 Additional Properties

In this section some additional facts are proved, which make it easier to work with the *functor-category* locale.

context *functor-category*
begin

lemma *Map-comp* [*simp*]:
assumes *seq t′ t* **and** *A.seq a′ a*
shows $Map\ (t' \cdot t)\ (a' \cdot_A a) = Map\ t'\ a' \cdot_B Map\ t\ a$
⟨*proof*⟩

lemma *Map-comp′*:
assumes *seq t′ t*
shows $Map\ (t' \cdot t) = vertical\text{-}composite.map\ A\ B\ (Map\ t)\ (Map\ t')$

⟨*proof*⟩

lemma *MkArr-eqI*:
assumes $F = F'$ **and** $G = G'$ **and** $\tau = \tau'$
shows *MkArr* F G τ = *MkArr* F' G' τ'
⟨*proof*⟩

lemma *iso-char* [*iff*]:
shows *iso* $t \longleftrightarrow t \neq$ *null* ∧ *natural-isomorphism* A B (*Dom* t) (*Cod* t) (*Map* t)
⟨*proof*⟩

end

14.3 Evaluation Functor

This section defines the evaluation map that applies an arrow of the functor category [A, B] to an arrow of A to obtain an arrow of B and shows that it is functorial.

locale *evaluation-functor* =
A: *category A* +
B: *category B* +
A-B: *functor-category A B* +
A-BxA: *product-category A-B.comp A*
for A :: *'a comp* (**infixr** $\cdot_A$ *55*)
and B :: *'b comp* (**infixr** $\cdot_B$ *55*)
begin

notation *A-B.comp* (**infixr** $\cdot_{[A,B]}$ *55*)
notation *A-BxA.comp* (**infixr** $\cdot_{[A,B]xA}$ *55*)
notation *A-B.in-hom* («- : - $\rightarrow_{[A,B]}$ -»)
notation *A-BxA.in-hom* («- : - $\rightarrow_{[A,B]xA}$ -»)

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*)
⟨*proof*⟩

lemma *is-functor*:
shows *functor A-BxA.comp B map*
⟨*proof*⟩

end

sublocale *evaluation-functor* ⊆ *functor A-BxA.comp B map*
⟨*proof*⟩
sublocale *evaluation-functor* ⊆ *binary-functor A-B.comp A B map* ⟨*proof*⟩

14.4 Currying

This section defines the notion of currying of a natural transformation between binary functors, to obtain a natural transformation between functors into a functor category, along with the inverse operation of uncurrying. We have only proved here what is needed to establish the results in theory *Limit* about limits in functor categories and have not attempted to fully develop the functoriality and naturality properties of these notions.

locale *currying* =
A1: *category A1* +
A2: *category A2* +
B: *category B*
for *A1* :: *'a1 comp* (**infixr** $\cdot_{A1}$ *55*)
and *A2* :: *'a2 comp* (**infixr** $\cdot_{A2}$ *55*)
and *B* :: *'b comp* (**infixr** $\cdot_{B}$ *55*)
begin

interpretation *A1xA2*: *product-category A1 A2* ⟨*proof*⟩
interpretation *A2-B*: *functor-category A2 B* ⟨*proof*⟩
interpretation *A2-BxA2*: *product-category A2-B.comp A2* ⟨*proof*⟩
interpretation *E*: *evaluation-functor A2 B* ⟨*proof*⟩

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* («- : - $\rightarrow_{A1xA2}$ -»)
notation *A2-B.in-hom* («- : - $\rightarrow_{[A2,B]}$ -»)
notation *A2-BxA2.in-hom* («- : - $\rightarrow_{[A2,B]xA2}$ -»)

A proper definition for *curry* requires that it be parametrized by binary functors *F* and *G* that are the domain and codomain of the natural transformations to which it is being applied. Similar parameters are not needed in the case of *uncurry*.

definition *curry* :: (*'a1* × *'a2* ⇒ *'b*) ⇒ (*'a1* × *'a2* ⇒ *'b*) ⇒ (*'a1* × *'a2* ⇒ *'b*)
⇒ *'a1* ⇒ (*'a2*, *'b*) *A2-B.arr*
where *curry F G τ f1* = (*if A1.arr f1 then*
A2-B.MkArr (*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*))
⟨*proof*⟩

lemma *uncurry-simp*:

assumes *A1xA2.arr f*
shows *uncurry* τ *f* = *E.map* (τ (*fst f*), *snd f*)
⟨*proof*⟩

lemma *curry-in-hom*:
assumes *f1*: *A1.arr f1*
and *natural-transformation A1xA2.comp B F G* τ
shows «*curry F G* τ *f1* : *curry F F F* (*A1.dom f1*) $\rightarrow_{[A2,B]}$ *curry G G G* (*A1.cod f1*)»
⟨*proof*⟩

lemma *curry-preserves-functors*:
assumes *functor A1xA2.comp B F*
shows *functor A1 A2-B.comp* (*curry F F F*)
⟨*proof*⟩

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*⟩

lemma *uncurry-preserves-functors*:
assumes *functor A1 A2-B.comp F*
shows *functor A1xA2.comp B* (*uncurry F*)
⟨*proof*⟩

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*⟩

lemma *uncurry-curry*:
assumes *natural-transformation A1xA2.comp B F G* τ
shows *uncurry* (*curry F G* τ) = τ
⟨*proof*⟩

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*⟩

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** $\cdot_{A1}$ *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 *A2-B.comp* (**infixr** $\cdot_{[A2,B]}$ *55*)
notation *A1xA2.in-hom* («- : - $\rightarrow_{A1xA2}$ -»)
notation *A2-B.in-hom* («- : - $\rightarrow_{[A2,B]}$ -»)

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*))
⟨*proof*⟩

lemma *is-functor*:
shows *functor A1 A2-B.comp map*
⟨*proof*⟩

end

sublocale *curried-functor* ⊆ *functor A1 A2-B.comp map*
⟨*proof*⟩

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* («- : - $\rightarrow_{A1xA2}$ -»)
notation *A1-B.in-hom* («- : - $\rightarrow_{[A1,B]}$ -»)

definition *map*
where *map* ≡ *curry F.sym F.sym F.sym*

lemma *map-simp* [*simp*]:
assumes *A2.arr f2*
shows *map f2* =
A1-B.MkArr (λ*f1*. *F* (*f1*, *A2.dom f2*)) (λ*f1*. *F* (*f1*, *A2.cod f2*)) (λ*f1*. *F* (*f1*, *f2*))
⟨*proof*⟩

lemma *is-functor*:
shows *functor A2 A1-B.comp map*
⟨*proof*⟩

end

sublocale *curried-functor′* ⊆ *functor A2 A1-B.comp map*
⟨*proof*⟩

end

Chapter 15

Yoneda

theory *Yoneda*
imports *DualCategory SetCat FunctorCategory*
begin

This theory defines the notion of a "hom-functor" and gives a proof of the Yoneda Lemma. In traditional developments of category theory based on set theories such as ZFC, hom-functors are normally defined to be functors into the large category **Set** whose objects are of *all* sets and whose arrows are functions between sets. However, in HOL there does not exist a single "type of all sets", so the notion of the category of *all* sets and functions does not make sense. To work around this, we consider a more general setting consisting of a category C together with a set category S and a function φ such that whenever b and a are objects of C then 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*.

15.1 Hom-Functors

A hom-functor for a category C allows us to regard the hom-sets of C as objects of a category S of sets and functions. Any description of a hom-functor for C must therefore specify the category S and provide some sort of correspondence between arrows of C and elements of objects of S. If we are to think of each hom-set $C.hom\ b\ a$ of C as corresponding to an object *Hom* (b, a) of S then at a minimum it ought to be the case that the correspondence between arrows and elements is bijective between $C.hom\ b\ a$ and *Hom* (b, a). The *hom-functor* locale defined below captures this idea by assuming a set category S and a function φ taking arrows of C to elements of $S.Univ$, such that φ is injective on each set $C.hom\ b\ a$. We show that these data induce a functor *Hom* from $Cop\times C$ to S in such a way that φ becomes a natural bijection between $C.hom\ b\ a$ and *Hom* (b, a).

locale *hom-functor* =
  C: *category C* +

S: *set-category S setp*
for *C* :: *'c comp* (**infixr** · *55*)
and *S* :: *'s comp* (**infixr** $\cdot_S$ *55*)
and *setp* :: *'s set* ⇒ *bool*
and *φ* :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s* +
assumes *maps-arr-to-Univ*: *C.arr f* ⟹ *φ* (*C.dom f*, *C.cod f*) *f* ∈ *S.Univ*
and *local-inj*: ⟦ *C.ide b*; *C.ide a* ⟧ ⟹ *inj-on* (*φ* (*b*, *a*)) (*C.hom b a*)
and *small-homs*: ⟦ *C.ide b*; *C.ide a* ⟧ ⟹ *setp* (*φ* (*b*, *a*) ‘ *C.hom b a*)
begin

sublocale *Cop*: *dual-category C* ⟨*proof*⟩
sublocale *CopxC*: *product-category Cop.comp C* ⟨*proof*⟩

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

definition *set*
where *set ba* ≡ *φ* (*fst ba*, *snd ba*) ‘ *C.hom* (*fst ba*) (*snd ba*)

lemma *set-subset-Univ*:
assumes *C.ide b* **and** *C.ide a*
shows *set* (*b*, *a*) ⊆ *S.Univ*
⟨*proof*⟩

definition *ψ* :: *'c* ∗ *'c* ⇒ *'s* ⇒ *'c*
where *ψ ba* = *inv-into* (*C.hom* (*fst ba*) (*snd ba*)) (*φ ba*)

lemma *φ-mapsto*:
assumes *C.ide b* **and** *C.ide a*
shows *φ* (*b*, *a*) ∈ *C.hom b a* → *set* (*b*, *a*)
⟨*proof*⟩

lemma *ψ-mapsto*:
assumes *C.ide b* **and** *C.ide a*
shows *ψ* (*b*, *a*) ∈ *set* (*b*, *a*) → *C.hom b a*
⟨*proof*⟩

lemma *ψ-φ* [*simp*]:
assumes «*f* : *b* → *a*»
shows *ψ* (*b*, *a*) (*φ* (*b*, *a*) *f*) = *f*
⟨*proof*⟩

lemma *φ-ψ* [*simp*]:
assumes *C.ide b* **and** *C.ide a*
and *x* ∈ *set* (*b*, *a*)
shows *φ* (*b*, *a*) (*ψ* (*b*, *a*) *x*) = *x*
⟨*proof*⟩

lemma *ψ-img-set*:
assumes *C.ide b* **and** *C.ide a*
shows *ψ (b, a) ‘ set (b, a) = C.hom b a*
⟨*proof*⟩

A hom-functor maps each arrow (g, f) of *CopxC* to the arrow of the set category S corresponding to the function that takes an arrow h of $(\cdot)$ to the arrow $f \cdot h \cdot g$ of $(\cdot)$ obtained by precomposing with g and postcomposing with f.

definition *map*
where *map gf* =
(*if CopxC.arr gf then*
S.mkArr (*set* (*CopxC.dom gf*)) (*set* (*CopxC.cod gf*))
(*φ* (*CopxC.cod gf*) *o* (*λh. snd gf · h · fst gf*) *o* *ψ* (*CopxC.dom gf*))
else S.null)

lemma *arr-map*:
assumes *CopxC.arr gf*
shows *S.arr* (*map gf*)
⟨*proof*⟩

lemma *map-ide* [*simp*]:
assumes *C.ide b* **and** *C.ide a*
shows *map* (*b, a*) = *S.mkIde* (*set* (*b, a*))
⟨*proof*⟩

lemma *set-map*:
assumes *C.ide a* **and** *C.ide b*
shows *S.set* (*map* (*b, a*)) = *set* (*b, a*)
⟨*proof*⟩

The definition does in fact yield a functor.

sublocale *functor CopxC.comp S map*
⟨*proof*⟩

lemma *is-functor*:
shows *functor CopxC.comp S map* ⟨*proof*⟩

sublocale *binary-functor Cop.comp C S map* ⟨*proof*⟩

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

The map φ determines a bijection between *C.hom b a* and *set* (b, a) which is natural in (b, a).

lemma *φ-local-bij*:
assumes *C.ide b* **and** *C.ide a*
shows *bij-betw* (*φ* (*b, a*)) (*C.hom b a*) (*set* (*b, a*))
⟨*proof*⟩

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*⟩

lemma *Dom-map*:
assumes $C.arr\ g$ **and** $C.arr\ f$
shows $S.Dom\ (map\ (g,\ f)) = set\ (C.cod\ g,\ C.dom\ f)$
⟨*proof*⟩

lemma *Cod-map*:
assumes $C.arr\ g$ **and** $C.arr\ f$
shows $S.Cod\ (map\ (g,\ f)) = set\ (C.dom\ g,\ C.cod\ f)$
⟨*proof*⟩

lemma *Fun-map*:
assumes $C.arr\ g$ **and** $C.arr\ f$
shows $S.Fun\ (map\ (g,\ f)) =$
$restrict\ (\varphi\ (C.dom\ g,\ C.cod\ f)\ o\ (\lambda h.\ f \cdot h \cdot g)\ o\ \psi\ (C.cod\ g,\ C.dom\ f))$
$(set\ (C.cod\ g,\ C.dom\ f))$
⟨*proof*⟩

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*⟩

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*⟩

end

Every category C has a hom-functor: take S to be the replete set category generated by the arrow type $'a$ of C and take $\varphi\ (b,\ a)$ to be the map $S.UP :: 'a \Rightarrow 'a\ SC.arr$.

context *category*
begin

interpretation S: *replete-setcat* ‹*TYPE*(*'a*)› ⟨*proof*⟩

lemma *has-hom-functor*:
shows *hom-functor C S.comp S.setp* (λ-. *S.UP*)
⟨*proof*⟩

end

The locales *set-valued-functor* and *set-valued-transformation* provide some abbreviations that are convenient when working with functors and natural transformations into a set category.

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

15.2 Yoneda Functors

A Yoneda functor is the functor from C to $[Cop, S]$ obtained by "currying" a hom-functor in its first argument.

locale *yoneda-functor* =
  C: *category C* +
  Cop: *dual-category C* +
  CopxC: *product-category Cop.comp C* +
  S: *set-category S setp* +
  Hom: *hom-functor C S setp* φ
for C :: *'c comp*   (**infixr** · *55*)
and S :: *'s comp*   (**infixr** $\cdot_S$ *55*)
and *setp* :: *'s set* ⇒ *bool*
and φ :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
begin

  sublocale *Cop-S*: *functor-category Cop.comp S* ⟨*proof*⟩
  sublocale *curried-functor' Cop.comp C S Hom.map* ⟨*proof*⟩

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

  abbreviation ψ
  where $\psi \equiv$ *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* ≡ *Cop-S.Map* (*map f*)

  lemma *Y-simp* [*simp*]:
  assumes *C.arr f*
  shows *Y f* = (λ*g*. *Hom.map* (*g*, *f*))
    ⟨*proof*⟩

  lemma *Y-ide-is-functor*:
  assumes *C.ide a*
  shows *functor Cop.comp S* (*Y a*)
    ⟨*proof*⟩

  lemma *Y-arr-is-transformation*:
  assumes *C.arr f*

shows *natural-transformation Cop.comp S* (*Y* (*C.dom f*)) (*Y* (*C.cod f*)) (*Y f*)
⟨*proof*⟩

lemma *Y-ide-arr* [*simp*]:
assumes *a*: *C.ide a* **and** «*g* : *b′* → *b*»
shows «*Y a g* : *Hom.map* (*b*, *a*) $\rightarrow_S$ *Hom.map* (*b′*, *a*)»
and *Y a g* = *S.mkArr* (*Hom.set* (*b*, *a*)) (*Hom.set* (*b′*, *a*)) (φ (*b′*, *a*) *o Cop.comp g o* ψ (*b*, *a*))
⟨*proof*⟩

lemma *Y-arr-ide* [*simp*]:
assumes *C.ide b* **and** «*f* : *a* → *a′*»
shows «*Y f b* : *Hom.map* (*b*, *a*) $\rightarrow_S$ *Hom.map* (*b*, *a′*)»
and *Y f b* = *S.mkArr* (*Hom.set* (*b*, *a*)) (*Hom.set* (*b*, *a′*)) (φ (*b*, *a′*) *o C f o* ψ (*b*, *a*))
⟨*proof*⟩

end

locale *yoneda-functor-fixed-object* =
yoneda-functor +
fixes *a*
assumes *ide-a*: *C.ide a*
begin

sublocale *functor Cop.comp S* ‹*Y a*›
⟨*proof*⟩
sublocale *set-valued-functor Cop.comp S setp* ‹*Y a*› ⟨*proof*⟩

end

The Yoneda lemma states that, given a category *C* and a functor *F* from *Cop* to a set category *S*, for each object *a* of *C*, the set of natural transformations from the contravariant functor *Y a* to *F* is in bijective correspondence with the set *F.SET a* of elements of *F a*.

Explicitly, if *e* is an arbitrary element of the set *F.SET a*, then the functions λ*x*. *F.FUN* (ψ (*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 τ *b* of τ at an arbitrary object *b* is completely determined by the single arrow τ*.FUN a* (φ (*a*, *a*) *a*))), which is the the element of *F.SET a* that corresponds to the image of the identity *a* under the function τ*.FUN a*. Then τ *b* is the arrow from *Y a b* to *F b* corresponding to the function λ*x*. (*F.FUN* (ψ (*b*, *a*) *x*) (τ*.FUN a* (φ (*a*, *a*) *a*))) from *S.set* (*Y a b*) to *F.SET b*.

The above expressions look somewhat more complicated than the usual versions due to the need to account for the coercions φ and ψ.

locale *yoneda-lemma* =
yoneda-functor-fixed-object C S setp φ *a* +
F: *set-valued-functor Cop.comp S setp F*
for *C* :: *′c comp* (**infixr** · *55*)

and S :: $'s$ *comp* (**infixr** $\cdot_S$ *55*)
and *setp* :: $'s$ *set* $\Rightarrow$ *bool*
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 $(\lambda x.\ F.FUN\ (\psi\ (b,\ a)\ x)\ e) \in Hom.set\ (b,\ a) \rightarrow F.SET\ b$.

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 $\mathcal{T}_o$*-in-hom*:
assumes *e*: $e \in S.set\ (F\ a)$ **and** *b*: $C.ide\ b$
shows «$\mathcal{T}_o\ e\ b : Y\ a\ b \rightarrow_S F\ b$»
⟨*proof*⟩

For each $e \in F.SET\ a$, the mapping $\mathcal{T}_o\ e$ gives the components of a natural transformation $\mathcal{T}$ from $Y\ a$ to F.

lemma $\mathcal{T}_o$*-induces-transformation*:
assumes *e*: $e \in S.set\ (F\ a)$
shows *transformation-by-components Cop.comp S* $(Y\ a)\ F\ (\mathcal{T}_o\ e)$
⟨*proof*⟩

definition $\mathcal{T}$ :: $'s \Rightarrow 'c \Rightarrow 's$
where $\mathcal{T}\ e \equiv$ *transformation-by-components.map Cop.comp S* $(Y\ a)\ (\mathcal{T}_o\ e)$

end

locale *yoneda-lemma-fixed-e* =
yoneda-lemma +
fixes *e*
assumes *E*: $e \in F.SET\ a$
begin

interpretation $\mathcal{T}e$: *transformation-by-components Cop.comp S* ‹$Y\ a$› F ‹$\mathcal{T}_o\ e$›
⟨*proof*⟩
sublocale $\mathcal{T}e$: *natural-transformation Cop.comp S* ‹$Y\ a$› F ‹$\mathcal{T}\ e$›
⟨*proof*⟩

lemma *natural-transformation-*$\mathcal{T}e$:
shows *natural-transformation Cop.comp S* $(Y\ a)\ F\ (\mathcal{T}\ e)$ ⟨*proof*⟩

lemma $\mathcal{T}$*e-ide*:
assumes *Cop.ide b*
shows *S.arr* ($\mathcal{T}$ *e b*)
and $\mathcal{T}$ *e b* = *S.mkArr* (*Hom.set* (*b*, *a*)) (*F.SET b*) (λx. *F.FUN* (ψ (*b*, *a*) *x*) *e*)
⟨*proof*⟩

end

locale *yoneda-lemma-fixed-*τ =
yoneda-lemma +
τ: *natural-transformation Cop.comp S* ‹*Y a*› *F* τ
for τ
begin

sublocale τ: *set-valued-transformation Cop.comp S setp* ‹*Y a*› *F* τ ⟨*proof*⟩

The key lemma: The component τ *b* of τ at an arbitrary object *b* is completely determined by the single element τ*.FUN a* (φ (*a*, *a*) *a*) ∈ *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*⟩

Consequently, if τ' is any natural transformation from *Y a* to *F* that agrees with τ at *a*, then $\tau' = \tau$.

lemma *eqI*:
assumes *natural-transformation Cop.comp S* (*Y a*) *F* τ' **and** τ' *a* = τ *a*
shows $\tau' = \tau$
⟨*proof*⟩

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* ∈ *F.SET a*
shows *natural-transformation Cop.comp S* (*Y a*) *F* ($\mathcal{T}$ *e*) **and** $\mathcal{E}$ ($\mathcal{T}$ *e*) = *e*
⟨*proof*⟩

lemma $\mathcal{E}\tau$*-mapsto*:
assumes *natural-transformation Cop.comp S* (*Y a*) *F* τ
shows $\mathcal{E}$ τ ∈ *F.SET a*
⟨*proof*⟩

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* τ
shows $\mathcal{T}$ ($\mathcal{E}$ τ) = τ
⟨*proof*⟩

The main result.

theorem *yoneda-lemma*:
shows *bij-betw* $\mathcal{T}$ (*F.SET a*) {τ. *natural-transformation Cop.comp S* (*Y a*) *F* τ}
⟨*proof*⟩

end

We now consider the special case in which F is the contravariant functor $Y\ a'$. Then for any e in *Hom.set* $(a,\ a')$ we have $\mathcal{T}\ e = Y\ (\psi\ (a,\ a')\ e)$, and $\mathcal{T}$ is a bijection from *Hom.set* $(a,\ a')$ to the set of natural transformations from $Y\ a$ to $Y\ a'$. It then follows that that the Yoneda functor Y is a fully faithful functor from C to the functor category $[Cop,\ S]$.

locale *yoneda-lemma-for-hom* =
yoneda-functor-fixed-object C S setp φ *a* +
Ya': *yoneda-functor-fixed-object C S setp* φ *a'* +
yoneda-lemma C S setp φ *Y a' a*
for *C* :: *'c comp* (**infixr** · *55*)
and *S* :: *'s comp* (**infixr** $\cdot_S$ *55*)
and *setp* :: *'s set* ⇒ *bool*
and φ :: *'c* ∗ *'c* ⇒ *'c* ⇒ *'s*
and *a* :: *'c*
and *a'* :: *'c* +
assumes *ide-a'*: *C.ide a'*
begin

In case F is the functor $Y\ a'$, for any $e \in$ *Hom.set* $(a,\ a')$ the induced natural transformation $\mathcal{T}\ e$ from $Y\ a$ to $Y\ a'$ is just $Y\ (\psi\ (a,\ a')\ e)$.

lemma *app-*$\mathcal{T}$*-equals*:
assumes *e*: *e* ∈ *Hom.set* (*a*, *a'*)
shows $\mathcal{T}$ *e* = *Y* (ψ (*a*, *a'*) *e*)
⟨*proof*⟩

lemma *is-injective-on-homs*:
shows *inj-on map* (*C.hom a a'*)
⟨*proof*⟩

end

context *yoneda-functor*
begin

sublocale *faithful-functor C Cop-S.comp map*
⟨*proof*⟩

lemma *is-faithful-functor*:
shows *faithful-functor C Cop-S.comp map*
⟨*proof*⟩

sublocale *full-functor C Cop-S.comp map*
⟨*proof*⟩

lemma *is-full-functor*:
shows *full-functor C Cop-S.comp map*
⟨*proof*⟩

sublocale *fully-faithful-functor C Cop-S.comp map* ⟨*proof*⟩

end

end

Chapter 16

Adjunction

theory *Adjunction*
imports *Yoneda*
begin

This theory defines the notions of adjoint functor and adjunction in various ways and establishes their equivalence. The notions "left adjoint functor" and "right adjoint functor" are defined in terms of universal arrows. "Meta-adjunctions" are defined in terms of natural bijections between hom-sets, where the notion of naturality is axiomatized directly. "Hom-adjunctions" formalize the notion of adjunction in terms of natural isomorphisms of hom-functors. "Unit-counit adjunctions" define adjunctions in terms of functors equipped with unit and counit natural transformations that satisfy the usual "triangle identities." The *adjunction* locale is defined as the grand unification of all the definitions, and includes formulas that connect the data from each of them. It is shown that each of the definitions induces an interpretation of the *adjunction* locale, so that all the definitions are essentially equivalent. Finally, it is shown that right adjoint functors are unique up to natural isomorphism.

The reference [7] was useful in constructing this theory.

16.1 Left Adjoint Functor

"e is an arrow from $F\ x$ to y."

locale *arrow-from-functor* =
  C: *category C* +
  D: *category D* +
  F: *functor D C F*
  for D :: $'d$ *comp* (**infixr** $\cdot_D$ *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* («- : - $\rightarrow_C$ -»)
notation *D.in-hom* («- : - $\rightarrow_D$ -»)

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

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

end

"*e* is a terminal arrow from *F x* to *y*."

locale *terminal-arrow-from-functor* =
arrow-from-functor D C F x y e
for *D* :: *'d comp* (**infixr** $\cdot_D$ *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 «*the-coext* x' *f* : $x' \rightarrow_D x$» **and** $f = e \cdot_C F$ (*the-coext* x' *f*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

16.2 Right Adjoint Functor

"*e* is an arrow from *x* to *G y*."

locale *arrow-to-functor* =
C: *category C* +
D: *category D* +
G: *functor C D G*
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and *G* :: *'c* ⇒ *'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* ⇒ *'d* ⇒ *'c* ⇒ *bool*
where *is-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* ⇒ *'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* ⇒ *'d* ⇒ *'c*
where *the-ext* $y'\ g = (THE\ f.\ is\text{-}ext\ y'\ g\ f)$

lemma *the-ext-prop*:
assumes *arrow-to-functor C D G x y′ g*
shows «*the-ext y′ g* : $y \rightarrow_C y'$» **and** $g = G$ (*the-ext y′ g*) $\cdot_D e$
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⇒ *′d* +
assumes *ex-initial-arrow*: *D.ide x* ⟹ (∃ *y e*. *initial-arrow-to-functor C D G x y e*)
begin

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

end

16.3 Various Definitions of Adjunction

16.3.1 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* ⇒ *′c*

and $G :: 'c \Rightarrow 'd$
and $\varphi :: 'd \Rightarrow 'c \Rightarrow 'd$
and $\psi :: 'c \Rightarrow 'd \Rightarrow 'c$ +
assumes φ*-in-hom*: $[\![\ D.ide\ y;\ C.in\text{-}hom\ f\ (F\ y)\ x\ ]\!] \Longrightarrow D.in\text{-}hom\ (\varphi\ y\ f)\ y\ (G\ x)$
and ψ*-in-hom*: $[\![\ C.ide\ x;\ D.in\text{-}hom\ g\ y\ (G\ x)\ ]\!] \Longrightarrow C.in\text{-}hom\ (\psi\ x\ g)\ (F\ y)\ x$
and ψ-φ: $[\![\ D.ide\ y;\ C.in\text{-}hom\ f\ (F\ y)\ x\ ]\!] \Longrightarrow \psi\ x\ (\varphi\ y\ f) = f$
and φ-ψ: $[\![\ C.ide\ x;\ D.in\text{-}hom\ g\ y\ (G\ x)\ ]\!] \Longrightarrow \varphi\ y\ (\psi\ x\ g) = g$
and φ*-naturality*: $[\![\ C.in\text{-}hom\ f\ x\ x';\ D.in\text{-}hom\ g\ y'\ y;\ C.in\text{-}hom\ h\ (F\ y)\ x\ ]\!] \Longrightarrow$
$\varphi\ y'\ (f \cdot_C h \cdot_C F\ g) = G\ f \cdot_D \varphi\ y\ h \cdot_D g$
begin

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

The naturality of ψ is a consequence of the naturality of φ and the other assumptions.

lemma ψ*-naturality*:
assumes f: «$f : x \rightarrow_C x'$» **and** g: «$g : y' \rightarrow_D y$» **and** h: «$h : y \rightarrow_D G\ x$»
shows $f \cdot_C \psi\ x\ h \cdot_C F\ g = \psi\ x'\ (G\ f \cdot_D h \cdot_D g)$
⟨*proof*⟩

lemma *respects-natural-isomorphism*:
assumes *natural-isomorphism* $D\ C\ F'\ F\ \tau$ **and** *natural-isomorphism* $C\ D\ G\ G'\ \mu$
shows *meta-adjunction* $C\ D\ F'\ G'$
$(\lambda y\ f.\ \mu\ (C.cod\ f) \cdot_D \varphi\ y\ (f \cdot_C$ *inverse-transformation.map* $D\ C\ F\ \tau\ y))$
$(\lambda x\ g.\ \psi\ x\ (($*inverse-transformation.map* $C\ D\ G'\ \mu\ x) \cdot_D g) \cdot_C \tau\ (D.dom\ g))$
⟨*proof*⟩

end

16.3.2 Hom-Adjunction

The bijection between hom-sets that defines an adjunction can be represented formally as a natural isomorphism of hom-functors. However, stating the definition this way is more complex than was the case for *meta-adjunction*. One reason is that we need to have a "set category" that is suitable as a target category for the hom-functors, and since the arrows of the categories C and D will in general have distinct types, we need a set category that simultaneously embeds both. Another reason is that we simply have to formally construct the various categories and functors required to express the definition.

This is a good place to point out that I have often included more sublocales in a locale than are strictly required. The main reason for this is the fact that the locale system in Isabelle only gives one name to each entity introduced by a locale: the name that it has in the first locale in which it occurs. This means that entities that make their first appearance deeply nested in sublocales will have to be referred to by long qualified names that can be difficult to understand, or even to discover. To counteract this, I have typically introduced sublocales before the superlocales that contain them to ensure that the entities in the sublocales can be referred to by short meaningful (and predictable) names. In my opinion, though, it would be better if the locale system would make entities that occur in multiple locales accessible by *all* possible qualified names, so that the most perspicuous name could be used in any particular context.

```
locale hom-adjunction =
  C: category C +
  D: category D +
  S: set-category S setp +
  Cop: dual-category C +
  Dop: dual-category D +
  CopxC: product-category Cop.comp C +
  DopxD: product-category Dop.comp D +
  DopxC: product-category Dop.comp C +
  F: functor D C F +
  G: functor C D G +
  HomC: hom-functor C S setp φC +
  HomD: hom-functor D S setp φD +
  Fop: dual-functor Dop.comp Cop.comp F +
  FopxC: product-functor Dop.comp C Cop.comp C Fop.map C.map +
  DopxG: product-functor Dop.comp C Dop.comp D Dop.map G +
  Hom-FopxC: composite-functor DopxC.comp CopxC.comp S FopxC.map HomC.map +
  Hom-DopxG: composite-functor DopxC.comp DopxD.comp S DopxG.map HomD.map +
  Hom-FopxC: set-valued-functor DopxC.comp S setp Hom-FopxC.map +
  Hom-DopxG: set-valued-functor DopxC.comp S setp Hom-DopxG.map +
  Φ: set-valued-transformation DopxC.comp S setp Hom-FopxC.map Hom-DopxG.map Φ +
  Ψ: set-valued-transformation DopxC.comp S setp Hom-DopxG.map Hom-FopxC.map Ψ +
  ΦΨ: inverse-transformations DopxC.comp S Hom-FopxC.map Hom-DopxG.map Φ Ψ
  for C :: 'c comp     (infixr ·C 55)
  and D :: 'd comp     (infixr ·D 55)
  and S :: 's comp     (infixr ·S 55)
  and setp :: 's set ⇒ bool
  and φC :: 'c * 'c ⇒ 'c ⇒ 's
  and φD :: 'd * 'd ⇒ 'd ⇒ 's
  and F :: 'd ⇒ 'c
  and G :: 'c ⇒ 'd
  and Φ :: 'd * 'c ⇒ 's
  and Ψ :: 'd * 'c ⇒ 's
begin

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

  abbreviation ψC :: 'c * 'c ⇒ 's ⇒ 'c
  where ψC ≡ HomC.ψ

  abbreviation ψD :: 'd * 'd ⇒ 's ⇒ 'd
  where ψD ≡ HomD.ψ

end
```

16.3.3 Unit/Counit Adjunction

Expressed in unit/counit terms, an adjunction consists of functors F: $D \to C$ and G: $C \to D$, equipped with natural transformations η: $1 \to GF$ and ε: $FG \to 1$ satisfying certain "triangle identities".

locale *unit-counit-adjunction* =
  C: *category C* +
  D: *category D* +
  F: *functor D C F* +
  G: *functor C D G* +
  GF: *composite-functor D C D F G* +
  FG: *composite-functor C D C G F* +
  FGF: *composite-functor D C C F ‹F o G›* +
  GFG: *composite-functor C D D G ‹G o F›* +
  η: *natural-transformation D D D.map ‹G o F› η* +
  ε: *natural-transformation C C ‹F o G› C.map ε* +
  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 ‹F o G o F› F ‹F o η› ‹ε o F›* +
  GεoηG: *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 ⇒ 'c*
  and *G* :: *'c ⇒ 'd*
  and *η* :: *'d ⇒ 'd*
  and *ε* :: *'c ⇒ 'c* +
  assumes *triangle-F*: *εFoFη.map = F*
  and *triangle-G*: *GεoηG.map = G*
begin

  notation *C.in-hom* («- : - $\to_C$ -»)
  notation *D.in-hom* («- : - $\to_D$ -»)

end

lemma *unit-determines-counit*:
assumes *unit-counit-adjunction C D F G η ε*
and *unit-counit-adjunction C D F G η ε'*
shows *ε = ε'*
⟨*proof*⟩

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*⟩

16.3.4 Adjunction

The grand unification of everything to do with an adjunction.

```
locale adjunction =
  C: category C +
  D: category D +
  S: set-category S setp +
  Cop: dual-category C +
  Dop: dual-category D +
  CopxC: product-category Cop.comp C +
  DopxD: product-category Dop.comp D +
  DopxC: product-category Dop.comp C +
  idDop: identity-functor Dop.comp +
  HomC: hom-functor C S setp φC +
  HomD: hom-functor D S setp φD +
  F: left-adjoint-functor D C F +
  G: right-adjoint-functor C D G +
  GF: composite-functor D C D F G +
  FG: composite-functor C D C G F +
  FGF: composite-functor D C C F FG.map +
  GFG: composite-functor C D D G GF.map +
  Fop: dual-functor Dop.comp Cop.comp F +
  FopxC: product-functor Dop.comp C Cop.comp C Fop.map C.map +
  DopxG: product-functor Dop.comp C Dop.comp D Dop.map G +
  Hom-FopxC: composite-functor DopxC.comp CopxC.comp S FopxC.map HomC.map +
  Hom-DopxG: composite-functor DopxC.comp DopxD.comp S DopxG.map HomD.map +
  Hom-FopxC: set-valued-functor DopxC.comp S setp Hom-FopxC.map +
  Hom-DopxG: set-valued-functor DopxC.comp S setp Hom-DopxG.map +
  η: natural-transformation D D D.map GF.map η +
  ε: natural-transformation C C FG.map C.map ε +
  Fη: natural-transformation D C F ‹F o G o F› ‹F o η› +
  ηG: natural-transformation C D G ‹G o F o G› ‹η o G› +
  εF: natural-transformation D C ‹F o G o F› F ‹ε o F› +
  Gε: natural-transformation C D ‹G o F o G› G ‹G o ε› +
  εFoFη: vertical-composite D C F FGF.map F ‹F o η› ‹ε o F› +
  GεoηG: vertical-composite C D G GFG.map G ‹η o G› ‹G o ε› +
  φψ: meta-adjunction C D F G φ ψ +
  ηε: unit-counit-adjunction C D F G η ε +
  ΦΨ: hom-adjunction C D S setp φC φD F G Φ Ψ
  for C :: 'c comp      (infixr ·C 55)
  and D :: 'd comp      (infixr ·D 55)
  and S :: 's comp      (infixr ·S 55)
  and setp :: 's set ⇒ bool
  and φC :: 'c * 'c ⇒ 'c ⇒ 's
  and φD :: 'd * 'd ⇒ 'd ⇒ 's
  and F :: 'd ⇒ 'c
  and G :: 'c ⇒ 'd
  and φ :: 'd ⇒ 'c ⇒ 'd
  and ψ :: 'c ⇒ 'd ⇒ 'c
```

and 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: $\llbracket$ *D.ide y*; «$f : F\ y \to_C\ x$» $\rrbracket \Longrightarrow \varphi\ y\ f = G\ f \cdot_D \eta\ y$
and ψ*-in-terms-of-*ε: $\llbracket$ *C.ide x*; «$g : y \to_D\ G\ x$» $\rrbracket \Longrightarrow \psi\ x\ g = \varepsilon\ x \cdot_C F\ g$
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: $\llbracket$ *D.ide y*; «$f : F\ y \to_C\ x$» $\rrbracket \Longrightarrow$
$\varphi\ y\ f = (\Phi\Psi.\psi D\ (y,\ G\ x)\ o\ S.Fun\ (\Phi\ (y,\ x))\ o\ \varphi C\ (F\ y,\ x))\ f$
and ψ*-in-terms-of-*Ψ: $\llbracket$ *C.ide x*; «$g : y \to_D\ G\ x$» $\rrbracket \Longrightarrow$
$\psi\ x\ g = (\Phi\Psi.\psi C\ (F\ y,\ x)\ o\ S.Fun\ (\Psi\ (y,\ x))\ o\ \varphi D\ (y,\ G\ x))\ g$
and Φ*-in-terms-of-*φ:
$\llbracket$ *C.ide x*; *D.ide y* $\rrbracket \Longrightarrow$
$\Phi\ (y,\ x) = S.mkArr\ (HomC.set\ (F\ y,\ x))\ (HomD.set\ (y,\ G\ x))$
$(\varphi D\ (y,\ G\ x)\ o\ \varphi\ y\ o\ \Phi\Psi.\psi C\ (F\ y,\ x))$
and Ψ*-in-terms-of-*ψ:
$\llbracket$ *C.ide x*; *D.ide y* $\rrbracket \Longrightarrow$
$\Psi\ (y,\ x) = S.mkArr\ (HomD.set\ (y,\ G\ x))\ (HomC.set\ (F\ y,\ x))$
$(\varphi C\ (F\ y,\ x)\ o\ \psi\ x\ o\ \Phi\Psi.\psi D\ (y,\ G\ x))$

16.4 Meta-Adjunctions Induce Unit/Counit Adjunctions

context *meta-adjunction*
begin

interpretation *GF*: *composite-functor D C D F G* ⟨*proof*⟩
interpretation *FG*: *composite-functor C D C G F* ⟨*proof*⟩
interpretation *FGF*: *composite-functor D C C F FG.map* ⟨*proof*⟩
interpretation *GFG*: *composite-functor C D D G GF.map* ⟨*proof*⟩

definition ηo :: $'d \Rightarrow 'd$
where $\eta o\ y = \varphi\ y\ (F\ y)$

lemma ηo*-in-hom*:
assumes *D.ide y*
shows «$\eta o\ y : y \to_D\ G\ (F\ y)$»
⟨*proof*⟩

lemma φ*-in-terms-of-*ηo:
assumes *D.ide y* **and** «$f : F\ y \to_C\ x$»
shows $\varphi\ y\ f = G\ f \cdot_D \eta o\ y$
⟨*proof*⟩

lemma φ*-F-char*:
assumes «$g : y' \to_D\ y$»
shows $\varphi\ y'\ (F\ g) = \eta o\ y \cdot_D\ g$
⟨*proof*⟩

interpretation η: *transformation-by-components D D D.map GF.map* ηo
⟨*proof*⟩

lemma η*-map-simp*:
assumes *D.ide y*
shows η*.map y* = φ *y* (*F y*)
⟨*proof*⟩

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

lemma εo*-in-hom*:
assumes *C.ide x*
shows «$\varepsilon o\ x : F\ (G\ x) \rightarrow_C x$»
⟨*proof*⟩

lemma ψ*-in-terms-of-*εo:
assumes *C.ide x* **and** «$g : y \rightarrow_D G\ x$»
shows $\psi\ x\ g = \varepsilon o\ x \cdot_C F\ g$
⟨*proof*⟩

lemma ψ*-G-char*:
assumes «f: $x \rightarrow_C x'$»
shows $\psi\ x'\ (G\ f) = f \cdot_C \varepsilon o\ x$
⟨*proof*⟩

interpretation ε: *transformation-by-components C C FG.map C.map* εo
⟨*proof*⟩

lemma ε*-map-simp*:
assumes *C.ide x*
shows ε*.map x* = ψ *x* (*G x*)
⟨*proof*⟩

interpretation *FD*: *composite-functor D D C D.map F* ⟨*proof*⟩
interpretation *CF*: *composite-functor D C C F C.map* ⟨*proof*⟩
interpretation *GC*: *composite-functor C C D C.map G* ⟨*proof*⟩
interpretation *DG*: *composite-functor C D D G D.map* ⟨*proof*⟩

interpretation $F\eta$: *natural-transformation D C F* ‹*F o G o F*› ‹*F o* η*.map*›
⟨*proof*⟩

interpretation εF: *natural-transformation D C* ‹*F o G o F*› *F* ‹ε*.map o F*›
⟨*proof*⟩

interpretation ηG: *natural-transformation C D G* ‹*G o F o G*› ‹η*.map o G*›
⟨*proof*⟩

interpretation $G\varepsilon$: *natural-transformation C D* ‹*G o F o G*› *G* ‹*G o* ε*.map*›

⟨*proof*⟩

interpretation $\varepsilon FoF\eta$: *vertical-composite* $D\ C\ F$ ‹F *o* G *o* F› F ‹F *o* η.*map*› ‹ε.*map* *o* F›
⟨*proof*⟩
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*›
⟨*proof*⟩

lemma *unit-counit-F*:
assumes $D.ide\ y$
shows $F\ y = \varepsilon o\ (F\ y) \cdot_C F\ (\eta o\ y)$
⟨*proof*⟩

lemma *unit-counit-G*:
assumes $C.ide\ x$
shows $G\ x = G\ (\varepsilon o\ x) \cdot_D \eta o\ (G\ x)$
⟨*proof*⟩

lemma *induces-unit-counit-adjunction'*:
shows *unit-counit-adjunction* $C\ D\ F\ G\ \eta$.*map* ε.*map*
⟨*proof*⟩

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

theorem *induces-unit-counit-adjunction*:
shows *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$
⟨*proof*⟩

lemma *η-is-natural-transformation*:
shows *natural-transformation* $D\ D\ D$.*map* GF.*map* η
⟨*proof*⟩

lemma *ε-is-natural-transformation*:
shows *natural-transformation* $C\ C\ FG$.*map* C.*map* ε
⟨*proof*⟩

From the defined η and ε we can recover the original φ and ψ.

lemma *φ-in-terms-of-η*:
assumes $D.ide\ y$ **and** «$f : F\ y \rightarrow_C x$»
shows $\varphi\ y\ f = G\ f \cdot_D \eta\ y$
⟨*proof*⟩

lemma *ψ-in-terms-of-ε*:
assumes $C.ide\ x$ **and** «$g : y \rightarrow_D G\ x$»
shows $\psi\ x\ g = \varepsilon\ x \cdot_C F\ g$
⟨*proof*⟩

end

16.5 Meta-Adjunctions Induce Left and Right Adjoint Functors

context *meta-adjunction*
begin

interpretation *unit-counit-adjunction* $C\ D\ F\ G\ \eta\ \varepsilon$
⟨*proof*⟩

lemma *has-terminal-arrows-from-functor*:
assumes x: $C.ide\ x$
shows *terminal-arrow-from-functor* $D\ C\ F\ (G\ x)\ x\ (\varepsilon\ x)$
and $\bigwedge y'\ f.$ *arrow-from-functor* $D\ C\ F\ y'\ x\ f$
$\Longrightarrow$ *terminal-arrow-from-functor.the-coext* $D\ C\ F\ (G\ x)\ (\varepsilon\ x)\ y'\ f = \varphi\ y'\ f$
⟨*proof*⟩

lemma *has-left-adjoint-functor*:
shows *left-adjoint-functor* $D\ C\ F$
⟨*proof*⟩

lemma *has-initial-arrows-to-functor*:
assumes y: $D.ide\ y$
shows *initial-arrow-to-functor* $C\ D\ G\ y\ (F\ y)\ (\eta\ y)$
and $\bigwedge x'\ g.$ *arrow-to-functor* $C\ D\ G\ y\ x'\ g \Longrightarrow$
initial-arrow-to-functor.the-ext $C\ D\ G\ (F\ y)\ (\eta\ y)\ x'\ g = \psi\ x'\ g$
⟨*proof*⟩

lemma *has-right-adjoint-functor*:
shows *right-adjoint-functor* $C\ D\ G$
⟨*proof*⟩

end

16.6 Unit/Counit Adjunctions Induce Meta-Adjunctions

context *unit-counit-adjunction*
begin

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

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

interpretation *meta-adjunction* $C\ D\ F\ G\ \varphi\ \psi$
⟨*proof*⟩

theorem *induces-meta-adjunction*:

shows *meta-adjunction C D F G* φ ψ ⟨*proof*⟩

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 $\eta\ y = \varphi\ y\ (F\ y)$
⟨*proof*⟩

lemma ε*-in-terms-of-*ψ:
assumes *C.ide x*
shows $\varepsilon\ x = \psi\ x\ (G\ x)$
⟨*proof*⟩

end

16.7 Left and Right Adjoint Functors Induce Meta-Adjunctions

A left adjoint functor induces a meta-adjunction, modulo the choice of a right adjoint and counit.

context *left-adjoint-functor*
begin

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

definition εo :: *'c* ⇒ *'c*
where εo *a* = (*SOME e*. *terminal-arrow-from-functor D C F* (*Go a*) *a e*)

lemma *Go-*εo*-terminal*:
assumes ∃ *b e*. *terminal-arrow-from-functor D C F b a e*
shows *terminal-arrow-from-functor D C F* (*Go a*) *a* (εo *a*)
⟨*proof*⟩

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* ⇒ *'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*)) ($f \cdot_C \varepsilon o$ (*C.dom f*))
else D.null)

lemma *G-ide*:
assumes *C.ide x*
shows *G x* = *Go x*
⟨*proof*⟩

lemma *G-is-functor*:
shows *functor C D G*

⟨*proof*⟩

interpretation *G*: *functor C D G* ⟨*proof*⟩

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*))
⟨*proof*⟩

interpretation *idC*: *identity-functor C* ⟨*proof*⟩
interpretation *GF*: *composite-functor C D C G F* ⟨*proof*⟩

interpretation *ε*: *transformation-by-components C C GF.map C.map εo*
⟨*proof*⟩

definition *ψ*
where *ψ x h* = *C* (*ε.map x*) (*F h*)

lemma *ψ-in-hom*:
assumes *C.ide x* **and** «$g : y \rightarrow_D G\ x$»
shows «$\psi\ x\ g : F\ y \rightarrow_C x$»
⟨*proof*⟩

lemma *ψ-natural*:
assumes *f*: «$f : x \rightarrow_C x'$» **and** *g*: «$g : y' \rightarrow_D y$» **and** *h*: «$h : y \rightarrow_D G\ x$»
shows $f \cdot_C \psi\ x\ h \cdot_C F\ g = \psi\ x'\ ((G\ f \cdot_D h) \cdot_D g)$
⟨*proof*⟩

lemma *ψ-inverts-coext*:
assumes *x*: *C.ide x* **and** *g*: «$g : y \rightarrow_D G\ x$»
shows *arrow-from-functor.is-coext D C F* (*G x*) (*ε.map x*) *y* (*ψ x g*) *g*
⟨*proof*⟩

lemma *ψ-invertible*:
assumes *y*: *D.ide y* **and** *f*: «$f : F\ y \rightarrow_C x$»
shows $\exists!g.$ «$g : y \rightarrow_D G\ x$» $\wedge\ \psi\ x\ g = f$
⟨*proof*⟩

definition *φ*
where *φ y f* = (*THE g*. «$g : y \rightarrow_D G$ (*C.cod f*)» $\wedge$ *ψ* (*C.cod f*) *g* = *f*)

lemma *φ-in-hom*:
assumes *D.ide y* **and** «$f : F\ y \rightarrow_C x$»
shows «$\varphi\ y\ f : y \rightarrow_D G\ x$»
⟨*proof*⟩

lemma *φ-ψ*:
assumes *C.ide x* **and** «$g : y \rightarrow_D G\ x$»

shows $\varphi\ y\ (\psi\ x\ g) = g$
⟨*proof*⟩

lemma ψ-φ:
assumes *D.ide* y **and** «$f : F\ y \rightarrow_C x$»
shows $\psi\ x\ (\varphi\ y\ f) = f$
⟨*proof*⟩

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*⟩

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

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* :: $'d \Rightarrow 'c$
where *Fo* y = (*SOME* x. $\exists\, u$. *initial-arrow-to-functor* $C\ D\ G\ y\ x\ u$)

definition ηo :: $'d \Rightarrow 'd$
where $\eta o\ y$ = (*SOME* u. *initial-arrow-to-functor* $C\ D\ G\ y$ (*Fo* y) u)

lemma *Fo*-ηo-*initial*:
assumes $\exists\, x\ u$. *initial-arrow-to-functor* $C\ D\ G\ y\ x\ u$
shows *initial-arrow-to-functor* $C\ D\ G\ y$ (*Fo* y) ($\eta o\ y$)
⟨*proof*⟩

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)) (ηo (*D.dom* g))
(*Fo* (*D.cod* g)) (ηo (*D.cod* g) $\cdot_D$ g)
else C.null)

lemma *F-ide*:
assumes *D.ide* y
shows $F\ y$ = *Fo* y
⟨*proof*⟩

lemma *F-is-functor*:

shows *functor D C F*
⟨*proof*⟩

interpretation *F*: *functor D C F* ⟨*proof*⟩

lemma *F-simp*:
assumes *D.arr g*
shows *F g = initial-arrow-to-functor.the-ext C D G (Fo (D.dom g)) (ηo (D.dom g))*
(Fo (D.cod g)) (ηo (D.cod g) $\cdot_D$ g)
⟨*proof*⟩

interpretation *FG*: *composite-functor D C D F G* ⟨*proof*⟩

interpretation *η*: *transformation-by-components D D D.map FG.map ηo*
⟨*proof*⟩

definition *φ*
where *φ y h = D (G h) (η.map y)*

lemma *φ-in-hom*:
assumes *y*: *D.ide y* **and** *f*: «*f* : *F y* $\rightarrow_C$ *x*»
shows «*φ y f* : *y* $\rightarrow_D$ *G x*»
⟨*proof*⟩

lemma *φ-natural*:
assumes *f*: «*f* : *x* $\rightarrow_C$ *x'*» **and** *g*: «*g* : *y'* $\rightarrow_D$ *y*» **and** *h*: «*h* : *F y* $\rightarrow_C$ *x*»
shows *φ y' (f* $\cdot_C$ *h* $\cdot_C$ *F g) = (G f* $\cdot_D$ *φ y h)* $\cdot_D$ *g*
⟨*proof*⟩

lemma *φ-inverts-ext*:
assumes *y*: *D.ide y* **and** *f*: «*f* : *F y* $\rightarrow_C$ *x*»
shows *arrow-to-functor.is-ext C D G (F y) (η.map y) x (φ y f) f*
⟨*proof*⟩

lemma *φ-invertible*:
assumes *x*: *C.ide x* **and** *g*: «*g* : *y* $\rightarrow_D$ *G x*»
shows ∃!*f*. «*f* : *F y* $\rightarrow_C$ *x*» ∧ *φ y f = g*
⟨*proof*⟩

definition *ψ*
where *ψ x g = (THE f.* «*f* : *F (D.dom g)* $\rightarrow_C$ *x*» ∧ *φ (D.dom g) f = g)*

lemma *ψ-in-hom*:
assumes *C.ide x* **and** «*g* : *y* $\rightarrow_D$ *G x*»
shows *C.in-hom (ψ x g) (F y) x*
⟨*proof*⟩

lemma *ψ-φ*:
assumes *D.ide y* **and** «*f* : *F y* $\rightarrow_C$ *x*»

shows $\psi\ x\ (\varphi\ y\ f) = f$
⟨*proof*⟩

lemma φ-ψ:
assumes *C.ide x* **and** «$g : y \rightarrow_D G\ x$»
shows $\varphi\ y\ (\psi\ x\ g) = g$
⟨*proof*⟩

theorem *induces-meta-adjunction*:
shows *meta-adjunction C D F G* φ ψ
⟨*proof*⟩

end

16.8 Meta-Adjunctions Induce Hom-Adjunctions

To obtain a hom-adjunction from a meta-adjunction, we need to exhibit hom-functors from C and D to a common set category S, so it is necessary to apply an actual concrete construction of such a category. We use the replete set category generated by the disjoint sum $'c + 'd$ of the arrow types of C and D.

context *meta-adjunction*
begin

interpretation *S*: *replete-setcat* ‹*TYPE*('*c*+'*d*)› ⟨*proof*⟩

definition *inC* :: '*c* ⇒ ('*c*+'*d*) *setcat.arr*
where *inC* ≡ *S.UP o Inl*

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

interpretation *S*: *replete-setcat* ‹*TYPE*('*c*+'*d*)› ⟨*proof*⟩
interpretation *Cop*: *dual-category C* ⟨*proof*⟩
interpretation *Dop*: *dual-category D* ⟨*proof*⟩
interpretation *CopxC*: *product-category Cop.comp C* ⟨*proof*⟩
interpretation *DopxD*: *product-category Dop.comp D* ⟨*proof*⟩
interpretation *DopxC*: *product-category Dop.comp C* ⟨*proof*⟩
interpretation *HomC*: *hom-functor C S.comp S.setp* ‹λ-. *inC*›
⟨*proof*⟩
interpretation *HomD*: *hom-functor D S.comp S.setp* ‹λ-. *inD*›
⟨*proof*⟩
interpretation *Fop*: *dual-functor D C F* ⟨*proof*⟩
interpretation *FopxC*: *product-functor Dop.comp C Cop.comp C Fop.map C.map* ⟨*proof*⟩
interpretation *DopxG*: *product-functor Dop.comp C Dop.comp D Dop.map G* ⟨*proof*⟩
interpretation *Hom-FopxC*: *composite-functor DopxC.comp CopxC.comp S.comp FopxC.map HomC.map* ⟨*proof*⟩
interpretation *Hom-DopxG*: *composite-functor DopxC.comp DopxD.comp S.comp DopxG.map HomD.map* ⟨*proof*⟩

lemma *inC-ψ* [*simp*]:
assumes *C.ide b* **and** *C.ide a* **and** $x \in inC$ ‘ *C.hom b a*
shows *inC* (*HomC.ψ* (*b*, *a*) *x*) = *x*
⟨*proof*⟩

lemma *ψ-inC* [*simp*]:
assumes *C.arr f*
shows *HomC.ψ* (*C.dom f*, *C.cod f*) (*inC f*) = *f*
⟨*proof*⟩

lemma *inD-ψ* [*simp*]:
assumes *D.ide b* **and** *D.ide a* **and** $x \in inD$ ‘ *D.hom b a*
shows *inD* (*HomD.ψ* (*b*, *a*) *x*) = *x*
⟨*proof*⟩

lemma *ψ-inD* [*simp*]:
assumes *D.arr f*
shows *HomD.ψ* (*D.dom f*, *D.cod f*) (*inD f*) = *f*
⟨*proof*⟩

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)))
⟨*proof*⟩

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))))
⟨*proof*⟩

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*⟩

interpretation Φ: *transformation-by-components*
DopxC.comp S.comp Hom-FopxC.map Hom-DopxG.map Φ*o*
⟨*proof*⟩

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*))
⟨*proof*⟩

abbreviation Ψ*o*
where Ψ*o 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 Ψ*o-in-hom*:
assumes *yx*: *DopxC.ide yx*
shows «Ψ*o yx* : *Hom-DopxG.map yx* $\rightarrow_S$ *Hom-FopxC.map yx*»
⟨*proof*⟩

lemma Φ-*inv*:
assumes *yx*: *DopxC.ide yx*
shows *S.inverse-arrows* (Φ.*map yx*) (Ψ*o yx*)
⟨*proof*⟩

interpretation Φ: *natural-isomorphism DopxC.comp S.comp*
Hom-FopxC.map Hom-DopxG.map Φ.*map*
⟨*proof*⟩

interpretation Ψ: *inverse-transformation DopxC.comp S.comp*
Hom-FopxC.map Hom-DopxG.map Φ.*map* ⟨*proof*⟩

interpretation ΦΨ: *inverse-transformations DopxC.comp S.comp*
Hom-FopxC.map Hom-DopxG.map Φ.*map* Ψ.*map*
⟨*proof*⟩

abbreviation Φ **where** Φ ≡ Φ.*map*
abbreviation Ψ **where** Ψ ≡ Ψ.*map*

abbreviation *HomC* **where** *HomC* ≡ *HomC.map*
abbreviation φ*C* **where** φ*C* ≡ λ-. *inC*
abbreviation *HomD* **where** *HomD* ≡ *HomD.map*
abbreviation φ*D* **where** φ*D* ≡ λ-. *inD*

theorem *induces-hom-adjunction*: *hom-adjunction C D S.comp S.setp* φ*C* φ*D F G* Φ Ψ
⟨*proof*⟩

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*)))
⟨*proof*⟩

The original φ and ψ can be recovered from Φ and Ψ.

interpretation Φ: *set-valued-transformation DopxC.comp S.comp S.setp*
Hom-FopxC.map Hom-DopxG.map Φ*.map* ⟨*proof*⟩

interpretation Ψ: *set-valued-transformation DopxC.comp S.comp S.setp*
Hom-DopxG.map Hom-FopxC.map Ψ*.map* ⟨*proof*⟩

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*⟩

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*⟩

end

16.9 Hom-Adjunctions Induce Meta-Adjunctions

context *hom-adjunction*
begin

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

definition ψ :: *'c* ⇒ *'d* ⇒ *'c*
where
ψ *x h* = (*HomC.*ψ (*F* (*D.dom h*), *x*) *o* Ψ*.FUN* (*D.dom h*, *x*) *o* 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*)))
(φ*C* (*F* (*D.dom* (*fst gf*)), *C.cod* (*snd gf*))
o (λ*h*. *snd gf* $\cdot_C$ *h* $\cdot_C$ *F* (*fst gf*))
o HomC.ψ (*F* (*D.cod* (*fst gf*)), *C.dom* (*snd gf*)))
⟨*proof*⟩

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 (λh. *G* (*snd gf*) $\cdot_D$ *h* $\cdot_D$ *fst gf*)
o *HomD*.ψ (*D.cod* (*fst gf*), *G* (*C.dom* (*snd gf*))))
⟨*proof*⟩

lemma Φ-*Fun-mapsto*:
assumes *D.ide y* **and** «*f* : *F y* $\rightarrow_C$ *x*»
shows Φ.*FUN* (*y*, *x*) $\in$ *HomC.set* (*F y*, *x*) $\rightarrow$ *HomD.set* (*y*, *G x*)
⟨*proof*⟩

lemma φ-*mapsto*:
assumes *y*: *D.ide y*
shows φ *y* $\in$ *C.hom* (*F y*) *x* $\rightarrow$ *D.hom y* (*G x*)
⟨*proof*⟩

lemma Φ-*simp*:
assumes *D.ide y* **and** *C.ide x*
shows *S.arr* (Φ (*y*, *x*))
and Φ (*y*, *x*) = *S.mkArr* (*HomC.set* (*F y*, *x*)) (*HomD.set* (*y*, *G x*))
(φD (*y*, *G x*) *o* φ *y* *o* ψC (*F y*, *x*))
⟨*proof*⟩

lemma Ψ-*Fun-mapsto*:
assumes *C.ide x* **and** «*g* : *y* $\rightarrow_D$ *G x*»
shows Ψ.*FUN* (*y*, *x*) $\in$ *HomD.set* (*y*, *G x*) $\rightarrow$ *HomC.set* (*F y*, *x*)
⟨*proof*⟩

lemma ψ-*mapsto*:
assumes *x*: *C.ide x*
shows ψ *x* $\in$ *D.hom y* (*G x*) $\rightarrow$ *C.hom* (*F y*) *x*
⟨*proof*⟩

lemma Ψ-*simp*:
assumes *D.ide y* **and** *C.ide x*
shows *S.arr* (Ψ (*y*, *x*))
and Ψ (*y*, *x*) = *S.mkArr* (*HomD.set* (*y*, *G x*)) (*HomC.set* (*F y*, *x*))
(φC (*F y*, *x*) *o* ψ *x* *o* ψD (*y*, *G x*))
⟨*proof*⟩

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* φ ψ
⟨*proof*⟩

theorem *induces-meta-adjunction*:
shows *meta-adjunction C D F G* φ ψ ⟨*proof*⟩

end

16.10 Putting it All Together

Combining the above results, an interpretation of any one of the locales: *left-adjoint-functor*, *right-adjoint-functor*, *meta-adjunction*, *hom-adjunction*, and *unit-counit-adjunction* extends to an interpretation of *adjunction*.

context *meta-adjunction*
begin

interpretation S: *replete-setcat* ⟨*proof*⟩
interpretation F: *left-adjoint-functor* D C F ⟨*proof*⟩
interpretation G: *right-adjoint-functor* C D G ⟨*proof*⟩

interpretation $\eta\varepsilon$: *unit-counit-adjunction* C D F G η ε
⟨*proof*⟩
interpretation $\Phi\Psi$: *hom-adjunction* C D $S.comp$ $S.setp$ φC φD F G Φ Ψ
⟨*proof*⟩

theorem *induces-adjunction*:
shows *adjunction* C D $S.comp$ $S.setp$ φC φD F G φ ψ η ε Φ Ψ
⟨*proof*⟩

end

context *unit-counit-adjunction*
begin

interpretation $\varphi\psi$: *meta-adjunction* C D F G φ ψ ⟨*proof*⟩

interpretation S: *replete-setcat* ⟨*proof*⟩
interpretation F: *left-adjoint-functor* D C F ⟨*proof*⟩
interpretation G: *right-adjoint-functor* C D G ⟨*proof*⟩

interpretation $\Phi\Psi$: *hom-adjunction* C D $S.comp$ $S.setp$
$\varphi\psi.\varphi C$ $\varphi\psi.\varphi D$ F G $\varphi\psi.\Phi$ $\varphi\psi.\Psi$
⟨*proof*⟩

theorem *induces-adjunction*:
shows *adjunction* C D $S.comp$ $S.setp$ $\varphi\psi.\varphi C$ $\varphi\psi.\varphi D$ F G φ ψ η ε $\varphi\psi.\Phi$ $\varphi\psi.\Psi$
⟨*proof*⟩

end

context *hom-adjunction*
begin

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

⟨*proof*⟩
interpretation F: *left-adjoint-functor D C F* ⟨*proof*⟩
interpretation G: *right-adjoint-functor C D G* ⟨*proof*⟩
interpretation $\eta\varepsilon$: *unit-counit-adjunction C D F G* $\varphi\psi.\eta$ $\varphi\psi.\varepsilon$
⟨*proof*⟩

theorem *induces-adjunction*:
shows *adjunction C D S setp* φC φD *F G* φ ψ $\varphi\psi.\eta$ $\varphi\psi.\varepsilon$ Φ Ψ
⟨*proof*⟩

end

context *left-adjoint-functor*
begin

interpretation $\varphi\psi$: *meta-adjunction C D F G* φ ψ
⟨*proof*⟩
interpretation S: *replete-setcat* ⟨*proof*⟩

theorem *induces-adjunction*:
shows *adjunction C D S.comp S.setp* $\varphi\psi.\varphi C$ $\varphi\psi.\varphi D$ *F G* φ ψ $\varphi\psi.\eta$ $\varphi\psi.\varepsilon$ $\varphi\psi.\Phi$ $\varphi\psi.\Psi$
⟨*proof*⟩

end

context *right-adjoint-functor*
begin

interpretation $\varphi\psi$: *meta-adjunction C D F G* φ ψ
⟨*proof*⟩
interpretation S: *replete-setcat* ⟨*proof*⟩

theorem *induces-adjunction*:
shows *adjunction C D S.comp S.setp* $\varphi\psi.\varphi C$ $\varphi\psi.\varphi D$ *F G* φ ψ $\varphi\psi.\eta$ $\varphi\psi.\varepsilon$ $\varphi\psi.\Phi$ $\varphi\psi.\Psi$
⟨*proof*⟩

end

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

lemma *adjoint-functors-respects-naturally-isomorphic*:
assumes *adjoint-functors C D F G*
and *naturally-isomorphic D C* F' F **and** *naturally-isomorphic C D G* G'
shows *adjoint-functors C D* F' G'
⟨*proof*⟩

lemma *left-adjoint-functor-respects-naturally-isomorphic*:
assumes *left-adjoint-functor D C F*

and *naturally-isomorphic D C F F'*
shows *left-adjoint-functor D C F'*
⟨*proof*⟩

lemma *right-adjoint-functor-respects-naturally-isomorphic*:
assumes *right-adjoint-functor C D G*
and *naturally-isomorphic C D G G'*
shows *right-adjoint-functor C D G'*
⟨*proof*⟩

16.11 Inverse Functors are Adjoints

lemma *inverse-functors-induce-meta-adjunction*:
assumes *inverse-functors C D F G*
shows *meta-adjunction C D F G* $(\lambda x.\ G)$ $(\lambda y.\ F)$
⟨*proof*⟩

lemma *inverse-functors-are-adjoints*:
assumes *inverse-functors A B F G*
shows *adjoint-functors A B F G*
⟨*proof*⟩

context *inverse-functors*
begin

lemma *η-char*:
shows *meta-adjunction.η B F* $(\lambda x.\ G)$ = *identity-functor.map B*
⟨*proof*⟩

lemma *ε-char*:
shows *meta-adjunction.ε A F G* $(\lambda y.\ F)$ = *identity-functor.map A*
⟨*proof*⟩

end

16.12 Composition of Adjunctions

locale *composite-adjunction* =
A: *category A* +
B: *category B* +
C: *category C* +
F: *functor B A F* +
G: *functor A B G* +
F': *functor C B F'* +
G': *functor B C G'* +
FG: *meta-adjunction A B F G* φ ψ +
F'G': *meta-adjunction B C F' G'* φ' ψ'
for *A* :: *'a comp* (**infixr** $\cdot_A$ *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 φ :: $'b \Rightarrow 'a \Rightarrow 'b$
and ψ :: $'a \Rightarrow 'b \Rightarrow 'a$
and φ' :: $'c \Rightarrow 'b \Rightarrow 'c$
and ψ' :: $'b \Rightarrow 'c \Rightarrow 'b$
begin

interpretation S: *replete-setcat* ⟨*proof*⟩
interpretation FG: *adjunction A B S.comp S.setp*
$FG.\varphi C$ $FG.\varphi D$ F G φ ψ $FG.\eta$ $FG.\varepsilon$ $FG.\Phi$ $FG.\Psi$
⟨*proof*⟩
interpretation $F'G'$: *adjunction B C S.comp S.setp* $F'G'.\varphi C$ $F'G'.\varphi D$ F' G' φ' ψ'
$F'G'.\eta$ $F'G'.\varepsilon$ $F'G'.\Phi$ $F'G'.\Psi$
⟨*proof*⟩

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*⟩

interpretation $K\eta H$: *natural-transformation C C* ‹$G'\ o\ F'$› ‹$G'\ o\ G\ o\ F\ o\ F'$›
‹$G'\ o\ FG.\eta\ o\ F'$›
⟨*proof*⟩
interpretation $G'\eta F'o\eta'$: *vertical-composite C C C.map* ‹$G'\ o\ F'$› ‹$G'\ o\ G\ o\ F\ o\ F'$›
$F'G'.\eta$ ‹$G'\ o\ FG.\eta\ o\ F'$› ⟨*proof*⟩

interpretation $F\varepsilon G$: *natural-transformation A A* ‹$F\ o\ F'\ o\ G'\ o\ G$› ‹$F\ o\ G$›
‹$F\ o\ F'G'.\varepsilon\ o\ G$›
⟨*proof*⟩
interpretation $\varepsilon oF\varepsilon'G$: *vertical-composite A A* ‹$F \circ F' \circ G' \circ G$› ‹$F\ o\ G$› *A.map*
‹$F\ o\ F'G'.\varepsilon\ o\ G$› $FG.\varepsilon$ ⟨*proof*⟩

interpretation *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*⟩
interpretation S: *replete-setcat* ⟨*proof*⟩
interpretation *adjunction A C S.comp S.setp* φC φD ‹$F \circ F'$› ‹$G' \circ G$›
‹$\lambda z.\ \varphi'\ z \circ \varphi\ (F'\ z)$› ‹$\lambda x.\ \psi\ x \circ \psi'\ (G\ x)$› η ε Φ Ψ
⟨*proof*⟩

lemma *η-char*:
shows $\eta = G'\eta F'o\eta'.map$
⟨*proof*⟩

lemma *ε-char*:
shows $\varepsilon = \varepsilon o F \varepsilon' G.map$
⟨*proof*⟩

end

16.13 Right Adjoints are Unique up to Natural Isomorphism

As an example of the use of the of the foregoing development, we show that two right adjoints to the same functor are naturally isomorphic.

theorem *two-right-adjoints-naturally-isomorphic*:
assumes *adjoint-functors* $C\ D\ F\ G$ **and** *adjoint-functors* $C\ D\ F\ G'$
shows *naturally-isomorphic* $C\ D\ G\ G'$
⟨*proof*⟩

end

Chapter 17

Equivalence of Categories

In this chapter we define the notions of equivalence and adjoint equivalence of categories and establish some properties of functors that are part of an equivalence.

theory *EquivalenceOfCategories*
imports *Adjunction*
begin

locale *equivalence-of-categories* =
C: *category C* +
D: *category D* +
F: *functor D C F* +
G: *functor C D G* +
η: *natural-isomorphism D D D.map G o F* η +
ε: *natural-isomorphism C C F o G C.map* ε
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and *F* :: *'d* ⇒ *'c*
and *G* :: *'c* ⇒ *'d*
and η :: *'d* ⇒ *'d*
and ε :: *'c* ⇒ *'c*
begin

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

lemma *C-arr-expansion*:
assumes *C.arr f*
shows ε (*C.cod f*) $\cdot_C$ *F* (*G f*) $\cdot_C$ *C.inv* (ε (*C.dom f*)) = *f*
and *C.inv* (ε (*C.cod f*)) $\cdot_C$ *f* $\cdot_C$ ε (*C.dom f*) = *F* (*G f*)
⟨*proof*⟩

lemma *G-is-faithful*:
shows *faithful-functor C D G*
⟨*proof*⟩

lemma *G-is-essentially-surjective*:
shows *essentially-surjective-functor C D G*
⟨*proof*⟩

interpretation *ε-inv*: *inverse-transformation C C ‹F o G› C.map ε* ⟨*proof*⟩
interpretation *η-inv*: *inverse-transformation D D D.map ‹G o F› η* ⟨*proof*⟩
interpretation *GF*: *equivalence-of-categories D C G F ε-inv.map η-inv.map* ⟨*proof*⟩

lemma *F-is-faithful*:
shows *faithful-functor D C F*
⟨*proof*⟩

lemma *F-is-essentially-surjective*:
shows *essentially-surjective-functor D C F*
⟨*proof*⟩

lemma *G-is-full*:
shows *full-functor C D G*
⟨*proof*⟩

end

context *equivalence-of-categories*
begin

interpretation *ε-inv*: *inverse-transformation C C ‹F o G› C.map ε* ⟨*proof*⟩
interpretation *η-inv*: *inverse-transformation D D D.map ‹G o F› η* ⟨*proof*⟩
interpretation *GF*: *equivalence-of-categories D C G F ε-inv.map η-inv.map* ⟨*proof*⟩

lemma *F-is-full*:
shows *full-functor D C F*
⟨*proof*⟩

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 ⇒ 'd* +
assumes *induces-equivalence*: ∃ *F η ε. equivalence-of-categories C D F G η ε*

begin

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

end

sublocale *equivalence-of-categories* ⊆ *equivalence-functor C D G*
⟨*proof*⟩

An equivalence functor is fully faithful and essentially surjective.

sublocale *equivalence-functor* ⊆ *fully-faithful-functor C D G*
⟨*proof*⟩

sublocale *equivalence-functor* ⊆ *essentially-surjective-functor C D G*
⟨*proof*⟩

lemma (**in** *inverse-functors*) *induce-equivalence*:
shows *equivalence-of-categories A B F G B.map A.map*
⟨*proof*⟩

lemma (**in** *invertible-functor*) *is-equivalence*:
shows *equivalence-functor A B G*
⟨*proof*⟩

lemma (**in** *identity-functor*) *is-equivalence*:
shows *equivalence-functor C C map*
⟨*proof*⟩

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*⟩

end

An adjoint equivalence is an equivalence of categories that is also an adjunction.

locale *adjoint-equivalence* =
unit-counit-adjunction C D F G η ε +
η: *natural-isomorphism D D D.map G o F η* +
ε: *natural-isomorphism C C F o G C.map ε*
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and *F* :: *'d* ⇒ *'c*

and $G :: 'c \Rightarrow 'd$
and $\eta :: 'd \Rightarrow 'd$
and $\varepsilon :: 'c \Rightarrow 'c$

An adjoint equivalence is clearly an equivalence of categories.

sublocale *adjoint-equivalence* ⊆ *equivalence-of-categories* ⟨*proof*⟩

context *adjoint-equivalence*
begin

The triangle identities for an adjunction reduce to inverse relations when η and ε are natural isomorphisms.

lemma *triangle-G'*:
assumes *C.ide a*
shows *D.inverse-arrows* (η (*G a*)) (*G* (ε *a*))
⟨*proof*⟩

lemma *triangle-F'*:
assumes *D.ide b*
shows *C.inverse-arrows* (*F* (η *b*)) (ε (*F b*))
⟨*proof*⟩

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* η ε
shows *adjoint-equivalence D C G F* (*inverse-transformation.map C C* (*C.map*) ε)
(*inverse-transformation.map D D* (*G o F*) η)
⟨*proof*⟩

end

Every fully faithful and essentially surjective functor underlies an adjoint equivalence. To prove this without repeating things that were already proved in *Category3.Adjunction*, we first show that a fully faithful and essentially surjective functor is a left adjoint functor, and then we show that if the left adjoint in a unit-counit adjunction is fully faithful and essentially surjective, then the unit and counit are natural isomorphisms; hence the adjunction is in fact an adjoint equivalence.

locale *fully-faithful-and-essentially-surjective-functor* =
C: *category C* +
D: *category D* +
fully-faithful-functor C D F +
essentially-surjective-functor C D F
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and $F :: 'c \Rightarrow 'd$
begin

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

lemma *is-left-adjoint-functor*:
shows *left-adjoint-functor C D F*
⟨*proof*⟩

lemma *extends-to-adjoint-equivalence*:
shows $\exists$ *G* η ε. *adjoint-equivalence C D G F* η ε
⟨*proof*⟩

lemma *is-right-adjoint-functor*:
shows *right-adjoint-functor C D F*
⟨*proof*⟩

lemma *is-equivalence-functor*:
shows *equivalence-functor C D F*
⟨*proof*⟩

sublocale *equivalence-functor C D F*
⟨*proof*⟩

end

context *equivalence-of-categories*
begin

The following development shows that an equivalence of categories can be refined to an adjoint equivalence by replacing just the counit.

abbreviation ε'
where $\varepsilon'\ a \equiv \varepsilon\ a \cdot_C F\ (D.inv\ (\eta\ (G\ a))) \cdot_C C.inv\ (\varepsilon\ (F\ (G\ a)))$

interpretation ε': *transformation-by-components C C* ‹*F* ∘ *G*› *C.map* ε'
⟨*proof*⟩

interpretation ε': *natural-isomorphism C C* ‹*F* ∘ *G*› *C.map* ε'*.map*
⟨*proof*⟩

lemma *Fη-inverse*:
assumes *D.ide b*
shows $F\ (\eta\ (G\ (F\ b))) = F\ (G\ (F\ (\eta\ b)))$
and $F\ (\eta\ b) \cdot_C \varepsilon\ (F\ b) = \varepsilon\ (F\ (G\ (F\ b))) \cdot_C F\ (\eta\ (G\ (F\ b)))$
and *C.inverse-arrows* $(F\ (\eta\ b))\ (\varepsilon'\ (F\ b))$
and $F\ (\eta\ b) = C.inv\ (\varepsilon'\ (F\ b))$
and $C.inv\ (F\ (\eta\ b)) = \varepsilon'\ (F\ b)$
⟨*proof*⟩

interpretation *FoGoF*: *composite-functor D C C F* ‹*F o G*› ⟨*proof*⟩

interpretation *GoFoG*: *composite-functor C D D G* ‹*G o F*› ⟨*proof*⟩

interpretation *natural-transformation D C F FoGoF.map* ‹$F \circ \eta$›
⟨*proof*⟩

interpretation *natural-transformation C D G GoFoG.map* ‹$\eta \circ G$›
⟨*proof*⟩

interpretation *natural-transformation D C FoGoF.map F* ‹ε'.*map* $\circ$ *F*›
⟨*proof*⟩

interpretation *natural-transformation C D GoFoG.map G* ‹*G* $\circ$ ε'.*map*›
⟨*proof*⟩

interpretation $\varepsilon'F$-$F\eta$: *vertical-composite D C F FoGoF.map F* ‹$F \circ \eta$› ‹ε'.*map* $\circ$ *F*› ⟨*proof*⟩
interpretation $G\varepsilon'$-ηG: *vertical-composite C D G GoFoG.map G* ‹η *o G*› ‹*G o* ε'.*map*›
⟨*proof*⟩

interpretation $\eta\varepsilon'$: *unit-counit-adjunction C D F G* η ε'.*map*
⟨*proof*⟩

interpretation $\eta\varepsilon'$: *adjoint-equivalence C D F G* η ε'.*map* ⟨*proof*⟩

lemma *refines-to-adjoint-equivalence*:
shows *adjoint-equivalence C D F G* η ε'.*map*
⟨*proof*⟩

end

end

Chapter 18

FreeCategory

theory *FreeCategory*
imports *Category ConcreteCategory*
begin

This theory defines locales for constructing the free category generated by a graph, as well as some special cases, including the discrete category generated by a set of objects, the "quiver" generated by a set of arrows, and a "parallel pair" of arrows, which is the diagram shape required for equalizers. Other diagram shapes can be constructed in a similar fashion.

18.1 Graphs

The following locale gives a definition of graphs in a traditional style.

locale *graph* =
fixes *Obj* :: *'obj set*
and *Arr* :: *'arr set*
and *Dom* :: *'arr* ⇒ *'obj*
and *Cod* :: *'arr* ⇒ *'obj*
assumes *dom-is-obj*: $x \in Arr \Longrightarrow Dom\ x \in Obj$
and *cod-is-obj*: $x \in Arr \Longrightarrow Cod\ x \in Obj$
begin

The list of arrows p forms a path from object x to object y if the domains and codomains of the arrows match up in the expected way.

definition *path*
where $path\ x\ y\ p \equiv (p = [] \wedge x = y \wedge x \in Obj) \vee$
$(p \neq [] \wedge x = Dom\ (hd\ p) \wedge y = Cod\ (last\ p) \wedge$
$(\forall n.\ n \geq 0 \wedge n < length\ p \longrightarrow nth\ p\ n \in Arr) \wedge$
$(\forall n.\ n \geq 0 \wedge n < (length\ p){-}1 \longrightarrow Cod\ (nth\ p\ n) = Dom\ (nth\ p\ (n{+}1))))$

lemma *path-Obj*:
assumes $x \in Obj$
shows $path\ x\ x\ []$

⟨*proof*⟩

lemma *path-single-Arr*:
assumes $x \in Arr$
shows *path* (*Dom x*) (*Cod x*) [*x*]
⟨*proof*⟩

lemma *path-concat*:
assumes *path x y p* **and** *path y z q*
shows *path x z* (*p* @ *q*)
⟨*proof*⟩

end

18.2 Free Categories

The free category generated by a graph has as its arrows all triples *MkArr x y p*, where *x* and *y* are objects and *p* is a path from *x* to *y*. We construct it here an instance of the general construction given by the *concrete-category* locale.

locale *free-category* =
G: *graph Obj Arr D C*
for *Obj* :: *'obj set*
and *Arr* :: *'arr set*
and *D* :: *'arr* ⇒ *'obj*
and *C* :: *'arr* ⇒ *'obj*
begin

type-synonym (*'o*, *'a*) *arr* = (*'o*, *'a list*) *concrete-category.arr*

sublocale *concrete-category* ‹*Obj* :: *'obj set*› ‹λ*x y*. *Collect* (*G.path x y*)›
‹λ-. []› ‹λ- - - *g f*. *f* @ *g*›
⟨*proof*⟩

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*])
⟨*proof*⟩

end

18.3 Discrete Categories

A discrete category is a category in which every arrow is an identity. We could construct it as the free category generated by a graph with no arrows, but it is simpler just to apply the *concrete-category* construction directly.

locale *discrete-category* =
fixes *Obj* :: *'obj set*
begin

type-synonym *'o arr* = (*'o*, *unit*) *concrete-category.arr*

sublocale *concrete-category* ‹*Obj* :: *'obj set*› ‹λ*x y*. *if x* = *y then* {*x*} *else* {}›
‹λ*x*. *x*› ‹λ- - *x* - -. *x*›
⟨*proof*⟩

abbreviation *comp* (**infixr** · *55*)
where *comp* ≡ *COMP*
notation *in-hom* («- : - → -»)

lemma *is-discrete*:
shows *arr f* ⟷ *ide f*
⟨*proof*⟩

lemma *arr-char*:
shows *arr f* ⟷ *Dom f* ∈ *Obj* ∧ *f* = *MkIde* (*Dom f*)
⟨*proof*⟩

lemma *arr-char'*:
shows *arr f* ⟷ *f* ∈ *MkIde* ‘ *Obj*
⟨*proof*⟩

lemma *dom-char*:
shows *dom f* = (*if arr f then f else null*)
⟨*proof*⟩

lemma *cod-char*:
shows *cod f* = (*if arr f then f else null*)
⟨*proof*⟩

lemma *in-hom-char*:
shows «*f* : *a* → *b*» ⟷ *arr f* ∧ *f* = *a* ∧ *f* = *b*
⟨*proof*⟩

lemma *seq-char*:
shows *seq g f* ⟷ *arr f* ∧ *f* = *g*
⟨*proof*⟩

lemma *comp-char*:
shows *g* · *f* = (*if seq g f then f else null*)

⟨*proof*⟩

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*
⟨*proof*⟩

end

18.4 Quivers

A quiver is a two-object category whose non-identity arrows all point in the same direction. A quiver is specified by giving the set of these non-identity arrows.

locale *quiver* =
fixes *Arr* :: *'arr set*
begin

type-synonym *'a arr* = (*unit*, *'a*) *concrete-category.arr*

sublocale *free-category* {*False*, *True*} *Arr* λ-. *False* λ-. *True*
⟨*proof*⟩

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*⟩

lemma *arr-char'*:

shows *arr f* $\longleftrightarrow$ *f* =
MkIde False $\vee$ *f* = *MkIde True* $\vee$ *f* $\in$ (λ*x*. *MkArr False True* [*x*]) ' *Arr*
⟨*proof*⟩

lemma *arr-char*:
shows *arr f* $\longleftrightarrow$ *f* = *Zero* $\vee$ *f* = *One* $\vee$ *f* $\in$ *fromArr* ' *Arr*
⟨*proof*⟩

lemma *dom-char*:
shows *dom f* = (*if arr f then*
if f = *One then One else Zero*
else null)
⟨*proof*⟩

lemma *cod-char*:
shows *cod f* = (*if arr f then*
if f = *Zero then Zero else One*
else null)
⟨*proof*⟩

lemma *seq-char*:
shows *seq g f* $\longleftrightarrow$ *arr g* $\wedge$ *arr f* $\wedge$ ((*f* = *Zero* $\wedge$ *g* $\neq$ *One*) $\vee$ (*f* $\neq$ *Zero* $\wedge$ *g* = *One*))
⟨*proof*⟩

lemma *not-ide-fromArr*:
shows $\neg$ *ide* (*fromArr x*)
⟨*proof*⟩

lemma *in-hom-char*:
shows «*f* : *a* $\rightarrow$ *b*» $\longleftrightarrow$ (*a* = *Zero* $\wedge$ *b* = *Zero* $\wedge$ *f* = *Zero*) $\vee$
(*a* = *One* $\wedge$ *b* = *One* $\wedge$ *f* = *One*) $\vee$
(*a* = *Zero* $\wedge$ *b* = *One* $\wedge$ *f* $\in$ *fromArr* ' *Arr*)
⟨*proof*⟩

lemma *Zero-not-eq-One* [*simp*]:
shows *Zero* $\neq$ *One*
⟨*proof*⟩

lemma *Zero-not-eq-fromArr* [*simp*]:
shows *Zero* $\notin$ *fromArr* ' *Arr*
⟨*proof*⟩

lemma *One-not-eq-fromArr* [*simp*]:
shows *One* $\notin$ *fromArr* ' *Arr*
⟨*proof*⟩

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*⟩

lemma *comp-simp* [*simp*]:
assumes *seq g f*
shows $f = Zero \Longrightarrow g \cdot f = g$
and $g = One \Longrightarrow g \cdot f = f$
⟨*proof*⟩

lemma *arr-fromArr*:
assumes $x \in Arr$
shows *arr* (*fromArr x*)
⟨*proof*⟩

lemma *toArr-in-Arr*:
assumes *arr f* **and** ¬*ide f*
shows *toArr f* ∈ *Arr*
⟨*proof*⟩

lemma *toArr-fromArr* [*simp*]:
assumes $x \in Arr$
shows *toArr* (*fromArr x*) = *x*
⟨*proof*⟩

lemma *fromArr-toArr* [*simp*]:
assumes *arr f* **and** ¬*ide f*
shows *fromArr* (*toArr f*) = *f*
⟨*proof*⟩

end

18.5 Parallel Pairs

A parallel pair is a quiver with two non-identity arrows. It is important in the definition of equalizers.

locale *parallel-pair* =
quiver {*False*, *True*} :: *bool set*
begin

typedef *arr* = *UNIV* :: *bool quiver.arr set* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dom-char*:
shows *dom f* = (*if f* = *j0* ∨ *f* = *j1 then Zero else if arr f then f else null*)
⟨*proof*⟩

lemma *cod-char*:
shows *cod f* = (*if f* = *j0* ∨ *f* = *j1 then One else if arr f then f else null*)
⟨*proof*⟩

lemma *j0-not-eq-j1* [*simp*]:
shows *j0* ≠ *j1*
⟨*proof*⟩

lemma *Zero-not-eq-j0* [*simp*]:
shows *Zero* ≠ *j0*
⟨*proof*⟩

lemma *Zero-not-eq-j1* [*simp*]:
shows *Zero* ≠ *j1*
⟨*proof*⟩

lemma *One-not-eq-j0* [*simp*]:
shows *One* ≠ *j0*
⟨*proof*⟩

lemma *One-not-eq-j1* [*simp*]:
shows *One* ≠ *j1*
⟨*proof*⟩

lemma *dom-simp* [*simp*]:
shows *dom Zero* = *Zero*
and *dom One* = *One*
and *dom j0* = *Zero*
and *dom j1* = *Zero*
⟨*proof*⟩

lemma *cod-simp* [*simp*]:
shows *cod Zero* = *Zero*
and *cod One* = *One*
and *cod j0* = *One*
and *cod j1* = *One*
⟨*proof*⟩

end

end

Chapter 19

DiscreteCategory

theory *DiscreteCategory*
imports *Category*
begin

The locale defined here permits us to construct a discrete category having a specified set of objects, assuming that the set does not exhaust the elements of its type. In that case, we have the convenient situation that the arrows of the category can be directly identified with the elements of the given set, rather than having to pass between the two via tedious coercion maps. If it cannot be guaranteed that the given set is not the universal set at its type, then the more general discrete category construction defined (using coercions) in *FreeCategory* can be used.

locale *discrete-category* =
  fixes *Obj* :: *'a set*
  and *Null* :: *'a*
  assumes *Null-not-in-Obj*: $Null \notin Obj$
begin

  definition *comp* :: *'a comp* (**infixr** · *55*)
  where $y \cdot x \equiv (\mathit{if}\ x \in Obj \wedge x = y\ \mathit{then}\ x\ \mathit{else}\ Null)$

  interpretation *partial-composition comp*
    ⟨*proof*⟩

  lemma *null-char*:
  shows $null = Null$
    ⟨*proof*⟩

  lemma *ide-char* [*iff*]:
  shows $ide\ f \longleftrightarrow f \in Obj$
    ⟨*proof*⟩

  lemma *domains-char*:
  shows $domains\ f = \{x.\ x \in Obj \wedge x = f\}$
    ⟨*proof*⟩

theorem *is-category*:
shows *category comp*
⟨*proof*⟩

end

sublocale *discrete-category* $\subseteq$ *category comp*
⟨*proof*⟩

context *discrete-category*
begin

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

lemma *dom-char* [*simp*]:
shows $dom\ f = (if\ f \in Obj\ then\ f\ else\ null)$
⟨*proof*⟩

lemma *cod-char* [*simp*]:
shows $cod\ f = (if\ f \in Obj\ then\ f\ else\ null)$
⟨*proof*⟩

lemma *comp-char* [*simp*]:
shows $comp\ g\ f = (if\ f \in Obj \land f = g\ then\ f\ else\ null)$
⟨*proof*⟩

lemma *is-discrete*:
shows $ide = arr$
⟨*proof*⟩

lemma *seq-char* [*iff*]:
shows $seq\ f\ g \longleftrightarrow ide\ f \land f = g$
⟨*proof*⟩

end

end

Chapter 20

Limit

theory *Limit*
imports *FreeCategory DiscreteCategory Adjunction*
begin

This theory defines the notion of limit in terms of diagrams and cones and relates it to the concept of a representation of a functor. The diagonal functor associated with a diagram shape J is defined and it is shown that a right adjoint to the diagonal functor gives limits of shape J and that a category has limits of shape J if and only if the diagonal functor is a left adjoint functor. Products and equalizers are defined as special cases of limits, and it is shown that a category with equalizers has limits of shape J if it has products indexed by the sets of objects and arrows of J. The existence of limits in a set category is investigated, and it is shown that every set category has equalizers and that a set category S has I-indexed products if and only if the universe of S "admits I-indexed tupling." The existence of limits in functor categories is also developed, showing that limits in functor categories are "determined pointwise" and that a functor category $[A, B]$ has limits of shape J if B does. Finally, it is shown that the Yoneda functor preserves limits.

This theory concerns itself only with limits; I have made no attempt to consider colimits. Although it would be possible to rework the entire development in dual form, it is possible that there is a more efficient way to dualize at least parts of it without repeating all the work. This is something that deserves further thought.

20.1 Representations of Functors

A representation of a contravariant functor F: $Cop \rightarrow S$, 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 $\Phi \in Y\ a \rightarrow F$, where Y: $C \rightarrow [Cop, S]$ is the Yoneda functor.

locale *representation-of-functor* =
  C: *category C* +
  Cop: *dual-category C* +
  S: *set-category S setp* +

F: *functor Cop.comp S F* +
Hom: *hom-functor C S setp* φ +
Ya: *yoneda-functor-fixed-object C S setp* φ *a* +
natural-isomorphism Cop.comp S ‹*Ya.Y a*› *F* Φ
for *C* :: ′*c comp* (**infixr** · *55*)
and *S* :: ′*s comp* (**infixr** $\cdot_S$ *55*)
and *setp* :: ′*s set* ⇒ *bool*
and φ :: ′*c* ∗ ′*c* ⇒ ′*c* ⇒ ′*s*
and *F* :: ′*c* ⇒ ′*s*
and *a* :: ′*c*
and Φ :: ′*c* ⇒ ′*s*
begin

abbreviation *Y* **where** *Y* ≡ *Ya.Y*
abbreviation ψ **where** ψ ≡ *Hom*.ψ

end

Two representations of the same functor are uniquely isomorphic.

locale *two-representations-one-functor* =
C: *category C* +
Cop: *dual-category C* +
S: *set-category S setp* +
F: *set-valued-functor Cop.comp S setp F* +
yoneda-functor C S setp φ +
Ya: *yoneda-functor-fixed-object C S setp* φ *a* +
Ya′: *yoneda-functor-fixed-object C S setp* φ *a*′ +
Φ: *representation-of-functor C S setp* φ *F a* Φ +
Φ′: *representation-of-functor C S setp* φ *F a*′ Φ′
for *C* :: ′*c comp* (**infixr** · *55*)
and *S* :: ′*s comp* (**infixr** $\cdot_S$ *55*)
and *setp* :: ′*s set* ⇒ *bool*
and *F* :: ′*c* ⇒ ′*s*
and φ :: ′*c* ∗ ′*c* ⇒ ′*c* ⇒ ′*s*
and *a* :: ′*c*
and Φ :: ′*c* ⇒ ′*s*
and *a*′ :: ′*c*
and Φ′ :: ′*c* ⇒ ′*s*
begin

interpretation Ψ: *inverse-transformation Cop.comp S* ‹*Y a*› *F* Φ ⟨*proof*⟩
interpretation Ψ′: *inverse-transformation Cop.comp S* ‹*Y a*′› *F* Φ′ ⟨*proof*⟩
interpretation ΦΨ′: *vertical-composite Cop.comp S* ‹*Y a*› *F* ‹*Y a*′› Φ Ψ′*.map* ⟨*proof*⟩
interpretation Φ′Ψ: *vertical-composite Cop.comp S* ‹*Y a*′› *F* ‹*Y a*› Φ′ Ψ*.map* ⟨*proof*⟩

lemma *are-uniquely-isomorphic*:
shows ∃!φ. «φ : *a* → *a*′» ∧ *C.iso* φ ∧ *map* φ = *Cop-S.MkArr* (*Y a*) (*Y a*′) ΦΨ′*.map*
⟨*proof*⟩

end

20.2 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* ⇒ *'c*
begin

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

end

lemma *comp-diagram-functor*:
assumes *diagram J C D* **and** *functor J' J F*
shows *diagram J' C* (*D o F*)
  ⟨*proof*⟩

A *cone* over a diagram $D: J \to C$ is a natural transformation from a constant functor to D. The value of the constant functor is the *apex* of the cone.

locale *cone* =
  C: *category C* +
  J: *category J* +
  D: *diagram J C D* +
  A: *constant-functor J C a* +
  natural-transformation J C A.map D χ
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *C* :: *'c comp* (**infixr** $\cdot$ *55*)
and *D* :: *'j* ⇒ *'c*
and *a* :: *'c*
and χ :: *'j* ⇒ *'c*
begin

  lemma *ide-apex*:

shows *C.ide a*
⟨*proof*⟩

lemma *component-in-hom*:
assumes *J.arr j*
shows «χ *j* : *a* → *D* (*J.cod j*)»
⟨*proof*⟩

lemma *cod-determines-component*:
assumes *J.arr j*
shows χ *j* = χ (*J.cod j*)
⟨*proof*⟩

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*⟩

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*⟩

context *functor*
begin

lemma *preserves-diagrams*:
fixes *J* :: *'j comp*
assumes *diagram J A D*
shows *diagram J B* (*F o D*)
⟨*proof*⟩

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*⟩

end

context *diagram*

begin

abbreviation *cone*
where $cone\ a\ \chi \equiv Limit.cone\ J\ C\ D\ a\ \chi$

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

lemma *cones-map-mapsto*:
assumes $C.arr\ f$
shows $cones\text{-}map\ f \in$
$extensional\ (cones\ (C.cod\ f)) \cap (cones\ (C.cod\ f) \rightarrow cones\ (C.dom\ f))$
⟨*proof*⟩

lemma *cones-map-ide*:
assumes $\chi \in cones\ a$
shows $cones\text{-}map\ a\ \chi = \chi$
⟨*proof*⟩

lemma *cones-map-comp*:
assumes $C.seq\ f\ g$
shows $cones\text{-}map\ (f \cdot g) = restrict\ (cones\text{-}map\ g\ o\ cones\text{-}map\ f)\ (cones\ (C.cod\ f))$
⟨*proof*⟩

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 $natural\text{-}transformation\ J\ C\ D\ D'\ \tau$
and $cone\ J\ C\ D\ a\ \chi$
and f: $partial\text{-}composition.in\text{-}hom\ C\ f\ a'\ a$
shows $diagram.cones\text{-}map\ J\ C\ D'\ f\ (vertical\text{-}composite.map\ J\ C\ \chi\ \tau)$
$= vertical\text{-}composite.map\ J\ C\ (diagram.cones\text{-}map\ J\ C\ D\ f\ \chi)\ \tau$
⟨*proof*⟩

Given a diagram D, we can construct a contravariant set-valued functor, which takes each object a of C to the set of cones over D with apex a, and takes each arrow f of C to the function on cones over D induced by pre-composition with *f*. For this, we need to introduce a set category S whose universe is large enough to contain all the cones over D, and we need to have an explicit correspondence between cones and elements of the universe of S. A replete set category S equipped with an injective mapping $\iota::('j \Rightarrow 'c) \Rightarrow 's$ serves this purpose.

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

  notation *S.in-hom* («- : - $\rightarrow_S$ -»)

  abbreviation o **where** o ≡ *S.DN*

  definition *map* :: *'c* ⇒ *'s*
  where *map* = (λ*f*. *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)
    ⟨*proof*⟩

  lemma *arr-map*:
  assumes *C.arr f*
  shows *S.arr* (*map f*)
  ⟨*proof*⟩

  lemma *map-ide*:
  assumes *C.ide a*
  shows *map a* = *S.mkIde* (*ι* ‘ *D.cones a*)
  ⟨*proof*⟩

  lemma *map-preserves-dom*:
  assumes *Cop.arr f*
  shows *map* (*Cop.dom f*) = *S.dom* (*map f*)
    ⟨*proof*⟩

  lemma *map-preserves-cod*:
  assumes *Cop.arr f*
  shows *map* (*Cop.cod f*) = *S.cod* (*map f*)
    ⟨*proof*⟩

lemma *map-preserves-comp*:
assumes *Cop.seq g f*
shows *map* $(g \cdot^{op} f) = map\ g \cdot_S map\ f$
⟨*proof*⟩

lemma *is-functor*:
shows *functor Cop.comp S map*
⟨*proof*⟩

end

sublocale *cones-functor* ⊆ *functor Cop.comp S map* ⟨*proof*⟩
sublocale *cones-functor* ⊆ *set-valued-functor Cop.comp S* ‹λ*A*. *A* ⊆ *S.Univ*› *map* ⟨*proof*⟩

20.3 Limits

20.3.1 Limit Cones

A *limit cone* for a diagram D is a cone χ over D with the universal property that any other cone χ' over the diagram D factors uniquely through χ.

locale *limit-cone* =
C: *category C* +
J: *category J* +
D: *diagram J C D* +
cone J C D a χ
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *C* :: *'c comp* (**infixr** · *55*)
and *D* :: *'j* ⇒ *'c*
and *a* :: *'c*
and χ :: *'j* ⇒ *'c* +
assumes *is-universal*: *cone J C D a'* χ' ⟹ ∃!*f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ'
begin

definition *induced-arrow* :: *'c* ⇒ (*'j* ⇒ *'c*) ⇒ *'c*
where *induced-arrow a'* χ' = (*THE f*. «*f* : *a'* → *a*» ∧ *D.cones-map f* χ = χ')

lemma *induced-arrowI*:
assumes χ': χ' ∈ *D.cones a'*
shows «*induced-arrow a'* χ' : *a'* → *a*»
and *D.cones-map* (*induced-arrow a'* χ') χ = χ'
⟨*proof*⟩

lemma *cones-map-induced-arrow*:
shows *induced-arrow a'* ∈ *D.cones a'* → *C.hom a' a*
and ⋀χ'. χ' ∈ *D.cones a'* ⟹ *D.cones-map* (*induced-arrow a'* χ') χ = χ'
⟨*proof*⟩

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*⟩

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*⟩

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'* χ' = *f*
⟨*proof*⟩

lemma *induced-arrow-self*:
shows *induced-arrow a* χ = *a*
⟨*proof*⟩

end

context *diagram*
begin

abbreviation *limit-cone*
where *limit-cone a* χ $\equiv$ *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* χ)

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

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*⟩

lemma *ex-limitE*:
assumes $\exists\, a$. *has-as-limit a*

obtains *a* χ **where** *limit-cone a* χ
⟨*proof*⟩

end

20.3.2 Limits by Representation

A limit for a diagram D can also be given by a representation (a, Φ) of the cones functor.

locale *representation-of-cones-functor* =
C: *category C* +
Cop: *dual-category C* +
J: *category J* +
D: *diagram J C D* +
S: *replete-concrete-set-category S UNIV* ι +
Cones: *cones-functor J C D S* ι +
Hom: *hom-functor C S* ‹$\lambda A.\ A \subseteq S.Univ$› φ +
representation-of-functor C S S.setp 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 :: $'c \Rightarrow {'s}$

20.3.3 Putting it all Together

A "limit situation" combines and connects the ways of presenting a limit.

locale *limit-situation* =
C: *category C* +
Cop: *dual-category C* +
J: *category J* +
D: *diagram J C D* +
S: *replete-concrete-set-category S UNIV* ι +
Cones: *cones-functor J C D S* ι +
Hom: *hom-functor C S S.setp* φ +
Φ: *representation-of-functor C S S.setp* φ *Cones.map a* Φ +
χ: *limit-cone J C D a* χ
for J :: $'j$ *comp* (**infixr** $\cdot_J$ *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: $\chi = S.DN\ (S.Fun\ (\Phi\ a)\ (\varphi\ (a,\ a)\ a))$

and Φ-*in-terms-of-*χ:
  Cop.ide $a' \Longrightarrow \Phi\ a' = S.mkArr\ (Hom.set\ (a', a))\ (\iota\ ‘\ D.cones\ a')$
  $(\lambda x.\ \iota\ (D.cones\text{-}map\ (Hom.\psi\ (a', a)\ x)\ \chi))$

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*.

20.3.4 Limit Cones Induce Limit Situations

To obtain a limit situation from a limit cone, we need to introduce a set category that is large enough to contain the hom-sets of C as well as the cones over D. We use the category of all $'c + ('j \Rightarrow 'c)$-sets for this.

context *limit-cone*
begin

  interpretation *Cop*: *dual-category C* ⟨*proof*⟩
  interpretation *CopxC*: *product-category Cop.comp C* ⟨*proof*⟩
  interpretation *S*: *replete-setcat* ‹$TYPE('c + ('j \Rightarrow 'c))$› ⟨*proof*⟩

  notation *S.comp*   (**infixr** $\cdot_S$ *55*)

  interpretation *Sr*: *replete-concrete-set-category S.comp UNIV* ‹*S.UP o Inr*›
    ⟨*proof*⟩

  interpretation *Cones*: *cones-functor J C D S.comp* ‹*S.UP o Inr*› ⟨*proof*⟩

  interpretation *Hom*: *hom-functor C S.comp S.setp* ‹λ-. *S.UP o Inl*›
    ⟨*proof*⟩

  interpretation *Y*: *yoneda-functor C S.comp S.setp* ‹λ-. *S.UP o Inl*› ⟨*proof*⟩
  interpretation *Ya*: *yoneda-functor-fixed-object C S.comp S.setp* ‹λ-. *S.UP o Inl*› *a*
    ⟨*proof*⟩

  abbreviation *inl* :: $'c \Rightarrow 'c + ('j \Rightarrow 'c)$ **where** *inl* ≡ *Inl*
  abbreviation *inr* :: $('j \Rightarrow 'c) \Rightarrow 'c + ('j \Rightarrow 'c)$ **where** *inr* ≡ *Inr*
  abbreviation ι **where** $\iota \equiv$ *S.UP o inr*
  abbreviation o **where** o ≡ *Cones*.o
  abbreviation φ **where** $\varphi \equiv$ λ-. *S.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 $Y\ a\ a' = S.mkIde\ (Hom.set\ (a', a))$

⟨*proof*⟩

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*))
(*φ* (*C.dom g*, *a*) *o Cop.comp g o ψ* (*C.cod g*, *a*))
⟨*proof*⟩

lemma *is-cone* [*simp*]:
shows $\chi \in$ *D.cones a*
⟨*proof*⟩

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 Φ*o* :: ′*c* ⇒ (′*c* + (′*j* ⇒ ′*c*)) *setcat.arr*
where
Φ*o a*′ = *S.mkArr* (*Hom.set* (*a*′, *a*)) (*ι* ‘ *D.cones a*′) (*λx*. *ι* (*D.cones-map* (*ψ* (*a*′, *a*) *x*) *χ*))

lemma Φ*o-in-hom*:
assumes *a*′: *C.ide a*′
shows «Φ*o a*′ : *S.mkIde* (*Hom.set* (*a*′, *a*)) $\rightarrow_S$ *S.mkIde* (*ι* ‘ *D.cones a*′)»
⟨*proof*⟩

interpretation Φ: *transformation-by-components Cop.comp S.comp* ‹*Y a*› *Cones.map* Φ*o*
⟨*proof*⟩

interpretation Φ: *set-valued-transformation Cop.comp S.comp S.setp*
‹*Y a*› *Cones.map* Φ.*map* ⟨*proof*⟩

interpretation Φ: *natural-isomorphism Cop.comp S.comp* ‹*Y a*› *Cones.map* Φ.*map*
⟨*proof*⟩

interpretation *R*: *representation-of-functor C S.comp S.setp φ Cones.map a* Φ.*map* ⟨*proof*⟩

lemma *χ-in-terms-of-*Φ:
shows *χ* = o (Φ.*FUN a* (*φ* (*a*, *a*) *a*))
⟨*proof*⟩

abbreviation *Hom*
where *Hom* ≡ *Hom.map*

abbreviation Φ
where Φ ≡ Φ.*map*

lemma *induces-limit-situation*:
shows *limit-situation J C D S.comp φ ι a* Φ *χ*

⟨*proof*⟩

no-notation *S.comp* (**infixr** $\cdot_S$ *55*)

end

sublocale *limit-cone* ⊆ *limit-situation J C D replete-setcat.comp* φ ι *a* Φ χ
⟨*proof*⟩

20.3.5 Representations of the Cones Functor Induce Limit Situations

context *representation-of-cones-functor*
begin

interpretation Φ: *set-valued-transformation Cop.comp S S.setp* ‹*Y a*› *Cones.map* Φ ⟨*proof*⟩
interpretation Ψ: *inverse-transformation Cop.comp S* ‹*Y a*› *Cones.map* Φ ⟨*proof*⟩
interpretation Ψ: *set-valued-transformation Cop.comp S S.setp*
Cones.map ‹*Y a*› Ψ*.map* ⟨*proof*⟩

abbreviation o
where o ≡ *Cones.*o

abbreviation χ
where χ ≡ o (*S.Fun* (Φ *a*) (φ (*a*, *a*) *a*))

lemma *Cones-SET-eq-ι-img-cones*:
assumes *C.ide* a'
shows *Cones.SET* a' = ι ‘ *D.cones* a'
⟨*proof*⟩

lemma $\iota\chi$:
shows ι χ = *S.Fun* (Φ *a*) (φ (*a*, *a*) *a*)
⟨*proof*⟩

interpretation χ: *cone J C D a* χ
⟨*proof*⟩

lemma *cone-χ*:
shows *D.cone a* χ ⟨*proof*⟩

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*) (ι χ)
⟨*proof*⟩

lemma *χ-is-universal*:
assumes *D.cone* a' χ'
shows «ψ (a', *a*) (Ψ*.FUN* a' (ι χ')) : $a' \rightarrow a$»
and *D.cones-map* (ψ (a', *a*) (Ψ*.FUN* a' (ι χ'))) χ = χ'

and ⟦ «$f' : a' \to a$»; *D.cones-map* f' $\chi = \chi'$ ⟧ $\Longrightarrow f' = \psi\ (a', a)\ (\Psi.FUN\ a'\ (\iota\ \chi'))$
⟨*proof*⟩

interpretation χ: *limit-cone J C D a* χ
⟨*proof*⟩

lemma χ*-is-limit-cone*:
shows *D.limit-cone a* χ ⟨*proof*⟩

lemma *induces-limit-situation*:
shows *limit-situation J C D S* $\varphi\ \iota\ a\ \Phi\ \chi$
⟨*proof*⟩

end

sublocale *representation-of-cones-functor* ⊆ *limit-situation J C D S* $\varphi\ \iota\ a\ \Phi\ \chi$
⟨*proof*⟩

20.4 Categories with Limits

context *category*
begin

A category C has limits of shape J if every diagram of shape J admits a limit cone.

definition *has-limits-of-shape*
where *has-limits-of-shape J* ≡ ∀ *D. diagram J C D* ⟶ (∃ *a* χ. *limit-cone J C D a* χ)

A category has limits at a type $'j$ if it has limits of shape J for every category J whose arrows are of type $'j$.

definition *has-limits*
where *has-limits* (- :: $'j$) ≡ ∀ *J* :: $'j$ *comp. category J* ⟶ *has-limits-of-shape J*

Whether a category has limits of shape J truly depends only on the "shape" (*i.e.* isomorphism class) of J and not on details of its construction.

lemma *has-limits-preserved-by-isomorphism*:
assumes *has-limits-of-shape J* **and** *isomorphic-categories J J'*
shows *has-limits-of-shape J'*
⟨*proof*⟩

end

20.4.1 Diagonal Functors

The existence of limits can also be expressed in terms of adjunctions: a category C has limits of shape J if the diagonal functor taking each object a in C to the constant-a diagram and each arrow $f \in C.hom\ a\ a'$ to the constant-f natural transformation between diagrams is a left adjoint functor.

locale *diagonal-functor* =

C: *category C* +
J: *category J* +
J-C: *functor-category J C*
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *C* :: *'c comp* (**infixr** $\cdot$ *55*)
begin

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

definition *map* :: *'c* ⇒ (*'j*, *'c*) *J-C.arr*
where *map f* = (*if C.arr f then J-C.MkArr* (*constant-functor.map J C* (*C.dom f*))
(*constant-functor.map J C* (*C.cod f*))
(*constant-transformation.map J C f*)
else J-C.null)

lemma *is-functor*:
shows *functor C J-C.comp map*
⟨*proof*⟩

sublocale *functor C J-C.comp map*
⟨*proof*⟩

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*⟩

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*⟩

lemma *bij-betw-ide-diagram*:
shows *bij-betw J-C.Map* (*Collect J-C.ide*) (*Collect* (*diagram J C*))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

Coextension along an arrow from a functor is equivalent to a transformation of cones.

lemma *coextension-iff-cones-map*:
assumes x: *arrow-from-functor* C J-C.*comp map* a d x
and g: «$g : a' \rightarrow a$»
and x': «$x' : map\ a' \rightarrow_{[J,C]} d$»
shows *arrow-from-functor.is-coext* C J-C.*comp map* a x a' x' g
$\longleftrightarrow$ J-C.*Map* x' = *diagram.cones-map* J C (J-C.*Map* d) g (J-C.*Map* x)
⟨*proof*⟩

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** $\cdot$ *55*)
and G :: (*'j*, *'c*) *functor-category.arr* ⇒ *'c*
and φ :: *'c* ⇒ (*'j*, *'c*) *functor-category.arr* ⇒ *'c*
and ψ :: (*'j*, *'c*) *functor-category.arr* ⇒ *'c* ⇒ (*'j*, *'c*) *functor-category.arr* +
assumes *adjoint*: *adjoint-functors* J-C.*comp* C Δ.*map* G
begin

interpretation S: *replete-setcat* ⟨*proof*⟩
interpretation *Adj*: *adjunction* J-C.*comp* C S.*comp* S.*setp* *Adj*.φC *Adj*.φD Δ.*map* G
φ ψ *Adj*.η *Adj*.ε *Adj*.Φ *Adj*.Ψ
⟨*proof*⟩

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*⟩

corollary *gives-limits*:
assumes *diagram J C D*
shows *diagram.has-as-limit J C D* (*G* (*J-C.MkIde D*))
⟨*proof*⟩

end

lemma (**in** *category*) *limits-are-isomorphic*:
fixes *J* :: *'j comp*
assumes *limit-cone J C D a χ* **and** *limit-cone J C D a' χ'*
shows *isomorphic a a'* **and** *iso* (*limit-cone.induced-arrow J C D a χ a' χ'*)
⟨*proof*⟩

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*⟩

20.5 Right Adjoint Functors Preserve Limits

context *right-adjoint-functor*
begin

lemma *preserves-limits*:
fixes *J* :: *'j comp*
assumes *diagram J C E* **and** *diagram.has-as-limit J C E a*
shows *diagram.has-as-limit J D* (*G o E*) (*G a*)
⟨*proof*⟩

end

20.6 Special Kinds of Limits

20.6.1 Terminal Objects

An object of a category *C* is a terminal object if and only if it is a limit of the empty diagram in *C*.

locale *empty-diagram* =
diagram J C D
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *C* :: *'c comp* (**infixr** $\cdot$ *55*)
and *D* :: *'j* ⇒ *'c* +
assumes *is-empty*: ¬*J.arr j*
begin

lemma *has-as-limit-iff-terminal*:
shows *has-as-limit a* $\longleftrightarrow$ *C.terminal a*
⟨*proof*⟩

end

20.6.2 Products

A *product* in a category C is a limit of a discrete diagram in C.

locale *discrete-diagram* =
J: *category J* +
diagram J C D
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *C* :: *'c comp* (**infixr** $\cdot$ *55*)
and *D* :: *'j* ⇒ *'c* +
assumes *is-discrete*: *J.arr* = *J.ide*
begin

abbreviation *mkCone*
where *mkCone F* ≡ (λ*j*. *if J.arr j then F j else C.null*)

lemma *cone-mkCone*:
assumes *C.ide a* **and** $\bigwedge$*j*. *J.arr j* ⟹ «*F j* : *a* → *D j*»
shows *cone a* (*mkCone F*)
⟨*proof*⟩

lemma *mkCone-cone*:
assumes *cone a* π
shows *mkCone* π = π
⟨*proof*⟩

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** $\cdot$ *55*)
and *D* :: *'i* ⇒ *'c*
and *null* :: *'i* +
assumes *maps-to-ide*: *i* ∈ *I* ⟹ *C.ide* (*D i*)
begin

definition *map*
where *map j* ≡ *if J.arr j then D j else C.null*

end

sublocale *discrete-diagram-from-map* ⊆ *discrete-diagram J.comp C map*
⟨*proof*⟩

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** · *55*)
and *D* :: *'j* ⇒ *'c*
and *a* :: *'c*
and *π* :: *'j* ⇒ *'c*
begin

lemma *is-cone*:
shows *D.cone a π* ⟨*proof*⟩

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** ⋀*j. J.arr j* ⟹ «*F j*: *b* → *D j*»
shows ∃!*f*. «*f* : *b* → *a*» ∧ (∀*j. J.arr j* ⟶ *π j* · *f* = *F j*)
⟨*proof*⟩

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*⟩

end

context *discrete-diagram*
begin

lemma *product-coneI*:
assumes *limit-cone a π*
shows *product-cone J C D a π*
⟨*proof*⟩

end

context *category*
begin

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

lemma *product-is-ide*:
assumes *has-as-product J D a*
shows *ide a*
⟨*proof*⟩

A category has I-indexed products for an $'i$-set I if every I-indexed discrete diagram has a product. In order to reap the benefits of being able to directly identify the elements of a set I with the objects of discrete category it generates (thereby avoiding the use of coercion maps), it is necessary to assume that $I \neq UNIV$. If we want to assert that a category has products indexed by the universe of some type $'i$, we have to pass to a larger type, such as $'i$ *option*.

definition *has-products*
where *has-products* (*I* :: *'i set*) ≡
I ≠ *UNIV* ∧
(∀ *J D*. *discrete-diagram J C D* ∧ *Collect* (*partial-composition.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*
⟨*proof*⟩

lemma *has-products-if-has-limits*:
assumes *has-limits* (*undefined* :: *'j*) **and** *I* ≠ (*UNIV* :: *'j set*)
shows *has-products I*
⟨*proof*⟩

lemma *has-finite-products-if-has-finite-limits*:
assumes ⋀*J* :: *'j comp*. (*finite* (*Collect* (*partial-composition.arr J*))) ⟹ *has-limits-of-shape J*
and *finite* (*I* :: *'j set*) **and** *I* ≠ *UNIV*
shows *has-products I*
⟨*proof*⟩

lemma *has-products-preserved-by-bijection*:
assumes *has-products I* **and** *bij-betw φ I I'* **and** *I'* ≠ *UNIV*
shows *has-products I'*
⟨*proof*⟩

lemma *ide-is-unary-product*:
assumes *ide a*

shows $\bigwedge m\ n :: nat.\ m \neq n \Longrightarrow$ *has-as-product* (*discrete-category.comp* {*m* :: *nat*} (*n* :: *nat*))
(λ*i*. *if i* = *m then a else null*) *a*
⟨*proof*⟩

lemma *has-unary-products*:
assumes *card I* = *1* **and** *I* ≠ *UNIV*
shows *has-products I*
⟨*proof*⟩

end

20.6.3 Equalizers

An *equalizer* in a category *C* is a limit of a parallel pair of arrows in *C*.

locale *parallel-pair-diagram* =
J: *parallel-pair* +
C: *category C*
for *C* :: *'c comp* (**infixr** · *55*)
and *f0* :: *'c*
and *f1* :: *'c* +
assumes *is-parallel*: *C.par f0 f1*
begin

no-notation *J.comp* (**infixr** · *55*)
notation *J.comp* (**infixr** $\cdot_J$ *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*⟩

end

sublocale *parallel-pair-diagram* ⊆ *diagram J.comp C map*
⟨*proof*⟩

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*⟩

lemma *is-equalized-by-cone*:
assumes *cone a χ*
shows *is-equalized-by* (*χ* (*J.Zero*))
⟨*proof*⟩

lemma *mkCone-cone*:
assumes *cone a χ*
shows *mkCone* (*χ J.Zero*) = *χ*
⟨*proof*⟩

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*⟩

lemma *is-universal'*:
assumes *D.is-equalized-by e'*
shows ∃!*h*. «*h* : *C.dom e'* → *C.dom e*» ∧ *e · h = e'*
⟨*proof*⟩

lemma *induced-arrowI'*:
assumes *D.is-equalized-by e'*
shows «*induced-arrow* (*C.dom e'*) (*D.mkCone e'*) : *C.dom e'* → *C.dom e*»
and *e · induced-arrow* (*C.dom e'*) (*D.mkCone e'*) = *e'*

⟨*proof*⟩

end

context *category*
begin

definition *has-as-equalizer*
where *has-as-equalizer f0 f1 e* ≡ *par f0 f1* ∧ *parallel-pair-diagram.has-as-equalizer C f0 f1 e*

definition *has-equalizers*
where *has-equalizers* = (∀ *f0 f1*. *par f0 f1* ⟶ (∃ *e*. *has-as-equalizer f0 f1 e*))

lemma *has-as-equalizerI* [*intro*]:
assumes *par f g* **and** *seq f e* **and** *f* · *e* = *g* · *e*
and ⋀*e'*. ⟦*seq f e'*; *f* · *e'* = *g* · *e'*⟧ ⟹ ∃!*h*. *e* · *h* = *e'*
shows *has-as-equalizer f g e*
⟨*proof*⟩

end

20.7 Limits by Products and Equalizers

A category with equalizers has limits of shape J if it has products indexed by the set of arrows of J and the set of objects of J. The proof is patterned after [4], Theorem 2, page 109:

> "The limit of F: $J \to C$ is the equalizer e of f, g: $\Pi_i\ F_i \to \Pi_u\ F_{cod\ u}$ $(u \in arr\ J,\ i \in J)$ where $p_u\ f = p_{cod\ u}$, $p_u\ g = F_u\ o\ p_{dom\ u}$; the limiting cone μ is $\mu_j = p_j\ e$, for $j \in J$."

locale *category-with-equalizers* =
  category C
for *C* :: *'c comp*   (**infixr** · *55*) +
assumes *has-equalizers*: *has-equalizers*
begin

lemma *has-limits-if-has-products*:
fixes *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
assumes *category J* **and** *has-products* (*Collect* (*partial-composition.ide J*))
and *has-products* (*Collect* (*partial-composition.arr J*))
shows *has-limits-of-shape J*
⟨*proof*⟩

end

20.8 Limits in a Set Category

In this section, we consider the special case of limits in a set category.

locale *diagram-in-set-category* =
  J: *category J* +
  S: *set-category S is-set* +
  diagram J S D
for *J* :: *'j comp* (**infixr** $\cdot_J$ *55*)
and *S* :: *'s comp* (**infixr** $\cdot$ *55*)
and *is-set* :: *'s set* ⇒ *bool*
and *D* :: *'j* ⇒ *'s*
begin

notation *S.in-hom* («- : - → -»)

An object *a* of a set category *S* is a limit of a diagram in *S* if and only if there is a bijection between the set *S.hom S.unity a* of points of *a* and the set of cones over the diagram that have apex *S.unity*.

lemma *limits-are-sets-of-cones*:
shows *has-as-limit a* ⟷ *S.ide a* ∧ (∃ φ. *bij-betw* φ (*S.hom S.unity a*) (*cones S.unity*))
⟨*proof*⟩

end

locale *diagram-in-replete-set-category* =
  J: *category J* +
  S: *replete-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* ⇒ *'s*
begin

sublocale *diagram-in-set-category J S S.setp D*
⟨*proof*⟩

end

context *set-category*
begin

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

lemma *has-equalizers*$_{SC}$:
shows *has-equalizers*
⟨*proof*⟩

end

sublocale *set-category* ⊆ *category-with-equalizers S*

⟨*proof*⟩

context *set-category*
begin

The aim of the next results is to characterize the conditions under which a set category has products. In a traditional development of category theory, one shows that the category **Set** of *all* sets has all small (*i.e.* set-indexed) products. In the present context we do not have a category of *all* sets, but rather only a category of all sets with elements at a particular type. Clearly, we cannot expect such a category to have products indexed by arbitrarily large sets. The existence of I-indexed products in a set category S implies that the universe $S.Univ$ of S must be large enough to admit the formation of I-tuples of its elements. Conversely, for a set category S the ability to form I-tuples in *Univ* implies that S has I-indexed products. Below we make this precise by defining the notion of when a set category S "admits I-indexed tupling" and we show that S has I-indexed products if and only if it admits I-indexed tupling.

The definition of "S admits I-indexed tupling" says that there is an injective map, from the space of extensional functions from I to *Univ*, to *Univ*. However for a convenient statement and proof of the desired result, the definition of extensional function from theory *HOL−Library.FuncSet* needs to be modified. The theory *HOL−Library.FuncSet* uses the definite, but arbitrarily chosen value *undefined* as the value to be assumed by an extensional function outside of its domain. In the context of the *set-category*, though, it is more natural to use $S.unity$, which is guaranteed to be an element of the universe of S, for this purpose. Doing things that way makes it simpler to establish a bijective correspondence between cones over D with apex *unity* and the set of extensional functions d that map each arrow j of J to an element $d\ j$ of *set* $(D\ j)$. Possibly it makes sense to go back and make this change in *set-category*, but that would mean completely abandoning *HOL−Library.FuncSet* and essentially introducing a duplicate version for use with *set-category*. As a compromise, what I have done here is to locally redefine the few notions from *HOL−Library.FuncSet* that I need in order to prove the next set of results. The redefined notions are primed to avoid confusion with the original versions.

definition *extensional′*
where $extensional'\ A \equiv \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = unity\}$

abbreviation *PiE′*
where $PiE'\ A\ B \equiv Pi\ A\ B \cap extensional'\ A$

abbreviation *restrict′*
where $restrict'\ f\ A \equiv \lambda x.\ if\ x \in A\ then\ f\ x\ else\ unity$

lemma *extensional′I* [*intro*]:
assumes $\bigwedge x.\ x \notin A \Longrightarrow f\ x = unity$
shows $f \in extensional'\ A$
⟨*proof*⟩

lemma *extensional′-arb*:

assumes $f \in$ *extensional′ A* **and** $x \notin A$
shows $f\ x =$ *unity*
⟨*proof*⟩

lemma *extensional′-monotone*:
assumes $A \subseteq B$
shows *extensional′ A* $\subseteq$ *extensional′ B*
⟨*proof*⟩

lemma *PiE′-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow PiE'\ A\ B \subseteq PiE'\ A\ C$
⟨*proof*⟩

end

locale *discrete-diagram-in-set-category* =
S: *set-category S* $\mathfrak{S}$ +
discrete-diagram J S D +
diagram-in-set-category J S $\mathfrak{S}$ *D*
for *J* :: *′j comp* (**infixr** $\cdot_J$ *55*)
and *S* :: *′s comp* (**infixr** $\cdot$ *55*)
and $\mathfrak{S}$ :: *′s set* $\Rightarrow$ *bool*
and *D* :: *′j* $\Rightarrow$ *′s*
begin

For D a discrete diagram in a set category, there is a bijective correspondence between cones over D with apex unity and the set of extensional functions d that map each arrow j of J to an element of $S.set\ (D\ j)$.

abbreviation *I*
where $I \equiv$ *Collect J.arr*

definition *funToCone*
where *funToCone F* $\equiv \lambda j.$ *if J.arr j then S.mkPoint* $(D\ j)\ (F\ j)$ *else S.null*

definition *coneToFun*
where *coneToFun* $\chi \equiv \lambda j.$ *if J.arr j then S.img* $(\chi\ j)$ *else S.unity*

lemma *funToCone-mapsto*:
shows *funToCone* $\in$ *S.PiE′ I* $(S.set\ o\ D) \rightarrow$ *cones S.unity*
⟨*proof*⟩

lemma *coneToFun-mapsto*:
shows *coneToFun* $\in$ *cones S.unity* $\rightarrow$ *S.PiE′ I* $(S.set\ o\ D)$
⟨*proof*⟩

lemma *funToCone-coneToFun*:
assumes $\chi \in$ *cones S.unity*
shows *funToCone* (*coneToFun* χ) $= \chi$
⟨*proof*⟩

lemma *coneToFun-funToCone*:
assumes $F \in S.PiE'\ I\ (S.set\ o\ D)$
shows *coneToFun* (*funToCone* F) $= F$
⟨*proof*⟩

lemma *bij-coneToFun*:
shows *bij-betw coneToFun* (*cones S.unity*) ($S.PiE'\ I\ (S.set\ o\ D)$)
⟨*proof*⟩

lemma *bij-funToCone*:
shows *bij-betw funToCone* ($S.PiE'\ I\ (S.set\ o\ D)$) (*cones S.unity*)
⟨*proof*⟩

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* $I \equiv \exists\pi.\ \pi \in PiE'\ I\ (\lambda\text{-}.\ Univ) \rightarrow Univ \wedge$ *inj-on* π ($PiE'\ I\ (\lambda\text{-}.\ Univ)$)

lemma *admits-tupling-monotone*:
assumes *admits-tupling* I **and** $I' \subseteq I$
shows *admits-tupling* I'
⟨*proof*⟩

lemma *admits-tupling-respects-bij*:
assumes *admits-tupling* I **and** *bij-betw* $\varphi\ I\ I'$
shows *admits-tupling* I'
⟨*proof*⟩

end

context *replete-set-category*
begin

lemma *has-products-iff-admits-tupling*:
fixes I :: *'i set*
shows *has-products* $I \longleftrightarrow I \neq UNIV \wedge$ *admits-tupling* I
⟨*proof*⟩

end

context *replete-set-category*
begin

Characterization of the completeness properties enjoyed by a set category: A set category S has all limits at a type $'j$, if and only if S admits I-indexed tupling for all

$'j$-sets I such that $I \neq UNIV$.

theorem *has-limits-iff-admits-tupling*:
shows *has-limits* (*undefined* :: $'j$) $\longleftrightarrow$ ($\forall I$ :: $'j$ *set*. $I \neq UNIV \longrightarrow$ *admits-tupling* I)
⟨*proof*⟩

end

20.9 Limits in Functor Categories

In this section, we consider the special case of limits in functor categories, with the objective of showing that limits in a functor category $[A, B]$ are given pointwise, and that $[A, B]$ has all limits that B has.

locale *parametrized-diagram* =
J: *category J* +
A: *category A* +
B: *category B* +
JxA: *product-category J A* +
binary-functor J A B D
for J :: $'j$ *comp* (**infixr** $\cdot_J$ *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 $L \equiv \lambda l\ \chi.\ \lambda a.$ *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 $P \equiv \lambda l\ \chi.\ \lambda a\ f.$ «f : l (*A.dom a*) $\rightarrow_B$ l (*A.cod a*)» $\wedge$
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 $\forall a$. *A.ide a* $\longrightarrow$ *limit-cone J B* (λj. D (j, a)) (l a) (χ a)
shows $\bigwedge a$. *A.arr a* $\Longrightarrow$ ($\exists! f$. $P\ l\ \chi\ a\ f$) $\wedge$ $P\ l\ \chi\ a$ ($L\ l\ \chi\ a$)

⟨*proof*⟩

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*⟩

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*⟩

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 $D :: 'j \Rightarrow ('a,\ 'b)$ *functor-category.arr*
begin

interpretation *JxA*: *product-category J A* ⟨*proof*⟩
interpretation *A-BxA*: *product-category A-B.comp A* ⟨*proof*⟩
interpretation *E*: *evaluation-functor A B* ⟨*proof*⟩
interpretation *Curry*: *currying J A B* ⟨*proof*⟩

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 $at\ a\ \tau \equiv \lambda j.\ Curry.uncurry\ \tau\ (j,\ a)$

lemma *at-simp*:
assumes *A.arr a* **and** *J.arr j* **and** *A-B.arr* $(\tau\ j)$
shows $at\ a\ \tau\ j = A\text{-}B.Map\ (\tau\ j)\ a$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *at-ide-is-diagram*:
assumes *a*: *A.ide a*
shows *diagram J B* (*at a D*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *cones-map-pointwise*:
assumes *cone x* χ **and** *cone x'* χ'
and *f*: *f* ∈ *A-B.hom x' x*
shows *cones-map f* $\chi = \chi'$ ⟷
(∀ *a*. *A.ide a* ⟶ *diagram.cones-map J B* (*at a D*) (*A-B.Map f a*) (*at a* χ) = *at a* χ')
⟨*proof*⟩

If χ is a cone with apex a over D, then χ is a limit cone if, for each object x of X, the cone obtained by evaluating χ at x is a limit cone with apex *A-B.Map a x* for the diagram in C obtained by evaluating D at x.

lemma *cone-is-limit-if-pointwise-limit*:
assumes *cone-*χ: *cone x* χ
and ∀ *a*. *A.ide a* ⟶ *diagram.limit-cone J B* (*at a D*) (*A-B.Map x a*) (*at a* χ)
shows *limit-cone x* χ
⟨*proof*⟩

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*⟩

lemma *has-limits-if-target-does*:
assumes *B.has-limits* (*undefined* :: *'j*)
shows *has-limits* (*undefined* :: *'j*)
⟨*proof*⟩

end

20.10 The Yoneda Functor Preserves Limits

In this section, we show that the Yoneda functor from C to $[Cop, S]$ preserves limits.

context *yoneda-functor*
begin

lemma *preserves-limits*:
fixes *J* :: *'j comp*
assumes *diagram J C D* **and** *diagram.has-as-limit J C D a*
shows *diagram.has-as-limit J Cop-S.comp* (*map o D*) (*map a*)
⟨*proof*⟩

end

end

Chapter 21

Category with Pullbacks

theory *CategoryWithPullbacks*
imports *Limit*
begin

In this chapter, we give a traditional definition of pullbacks in a category as limits of cospan diagrams and we define a locale *category-with-pullbacks* that is satisfied by categories in which every cospan diagram has a limit. These definitions build on the general definition of limit that we gave in *Category3.Limit*. We then define a locale *elementary-category-with-pullbacks* that axiomatizes categories equipped with chosen functions that assign to each cospan a corresponding span of "projections", which enjoy the familiar universal property of a pullback. After developing consequences of the axioms, we prove that the two locales are in agreement, in the sense that every interpretation of *category-with-pullbacks* extends to an interpretation of *elementary-category-with-pullbacks*, and conversely, the underlying category of an interpretation of *elementary-category-with-pullbacks* always yields an interpretation of *category-with-pullbacks*.

21.1 Commutative Squares

context *category*
begin

The following provides some useful technology for working with commutative squares.

definition *commutative-square*
where *commutative-square f g h k* ≡ *cospan f g* ∧ *span h k* ∧ *dom f* = *cod h* ∧ *f* · *h* = *g* · *k*

lemma *commutative-squareI* [*intro*, *simp*]:
assumes *cospan f g* **and** *span h k* **and** *dom f* = *cod h* **and** *f* · *h* = *g* · *k*
shows *commutative-square f g h k*
⟨*proof*⟩

lemma *commutative-squareE* [*elim*]:
assumes *commutative-square f g h k*

and ⟦ *arr f*; *arr g*; *arr h*; *arr k*; *cod f* = *cod g*; *dom h* = *dom k*; *dom f* = *cod h*;
dom g = *cod k*; *f* · *h* = *g* · *k* ⟧ ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *commutative-square-comp-arr*:
assumes *commutative-square f g h k* **and** *seq h l*
shows *commutative-square f g* (*h* · *l*) (*k* · *l*)
⟨*proof*⟩

lemma *arr-comp-commutative-square*:
assumes *commutative-square f g h k* **and** *seq l f*
shows *commutative-square* (*l* · *f*) (*l* · *g*) *h k*
⟨*proof*⟩

end

21.2 Cospan Diagrams

The "shape" of a cospan diagram is a category having two non-identity arrows with distinct domains and a common codomain.

locale *cospan-shape*
begin

datatype *Arr* = *Null* | *AA* | *BB* | *TT* | *AT* | *BT*

fun *comp*
where *comp AA AA* = *AA*
| *comp AT AA* = *AT*
| *comp TT AT* = *AT*
| *comp BB BB* = *BB*
| *comp BT BB* = *BT*
| *comp TT BT* = *BT*
| *comp TT TT* = *TT*
| *comp* - - = *Null*

interpretation *partial-composition comp*
⟨*proof*⟩

lemma *null-char*:
shows *null* = *Null*
⟨*proof*⟩

lemma *ide-char*:
shows *ide f* ⟷ *f* = *AA* ∨ *f* = *BB* ∨ *f* = *TT*
⟨*proof*⟩

fun *Dom*

where *Dom AA* = *AA*
  | *Dom BB* = *BB*
  | *Dom TT* = *TT*
  | *Dom AT* = *AA*
  | *Dom BT* = *BB*
  | *Dom* - = *Null*

fun *Cod*
where *Cod AA* = *AA*
  | *Cod BB* = *BB*
  | *Cod TT* = *TT*
  | *Cod AT* = *TT*
  | *Cod BT* = *TT*
  | *Cod* - = *Null*

lemma *domains-char′*:
shows *domains f* = (*if f* = *Null then* {} *else* {*Dom f*})
⟨*proof*⟩

lemma *codomains-char′*:
shows *codomains f* = (*if f* = *Null then* {} *else* {*Cod f*})
⟨*proof*⟩

lemma *arr-char*:
shows $arr\ f \longleftrightarrow f \neq Null$
⟨*proof*⟩

lemma *seq-char*:
shows $seq\ g\ f \longleftrightarrow (f = AA \wedge (g = AA \vee g = AT)) \vee$
$(f = BB \wedge (g = BB \vee g = BT)) \vee$
$(f = AT \wedge g = TT) \vee$
$(f = BT \wedge g = TT) \vee$
$(f = TT \wedge g = TT)$
⟨*proof*⟩

interpretation *category comp*
⟨*proof*⟩

lemma *is-category*:
shows *category comp*
⟨*proof*⟩

lemma *dom-char*:
shows *dom* = *Dom*
⟨*proof*⟩

lemma *cod-char*:
shows *cod* = *Cod*
⟨*proof*⟩

end

sublocale *cospan-shape* ⊆ *category comp*
⟨*proof*⟩

locale *cospan-diagram* =
J: *cospan-shape* +
C: *category C*
for *C* :: *'c comp* (**infixr** · *55*)
and *f0* :: *'c*
and *f1* :: *'c* +
assumes *is-cospan*: *C.cospan f0 f1*
begin

no-notation *J.comp* (**infixr** · *55*)
notation *J.comp* (**infixr** $\cdot_J$ *55*)

fun *map*
where *map J.AA* = *C.dom f0*
| *map J.BB* = *C.dom f1*
| *map J.TT* = *C.cod f0*
| *map J.AT* = *f0*
| *map J.BT* = *f1*
| *map* - = *C.null*

end

sublocale *cospan-diagram* ⊆ *diagram J.comp C map*
⟨*proof*⟩

21.3 Category with Pullbacks

A *pullback* in a category C is a limit of a cospan diagram in C.

context *cospan-diagram*
begin

definition *mkCone*
where *mkCone p0 p1* ≡ λ*j*. *if j* = *J.AA then p0*
else if j = *J.BB then p1*
else if j = *J.AT then f0* · *p0*
else if j = *J.BT then f1* · *p1*
else if j = *J.TT then f0* · *p0*
else C.null

abbreviation *is-rendered-commutative-by*
where *is-rendered-commutative-by p0 p1* ≡ *C.seq f0 p0* ∧ *f0* · *p0* = *f1* · *p1*

abbreviation *has-as-pullback*
where *has-as-pullback p0 p1* ≡ *limit-cone* (*C.dom p0*) (*mkCone p0 p1*)

lemma *cone-mkCone*:
assumes *is-rendered-commutative-by p0 p1*
shows *cone* (*C.dom p0*) (*mkCone p0 p1*)
⟨*proof*⟩

lemma *is-rendered-commutative-by-cone*:
assumes *cone a χ*
shows *is-rendered-commutative-by* (*χ J.AA*) (*χ J.BB*)
⟨*proof*⟩

lemma *mkCone-cone*:
assumes *cone a χ*
shows *mkCone* (*χ J.AA*) (*χ J.BB*) = *χ*
⟨*proof*⟩

lemma *cone-iff-commutative-square*:
shows *cone* (*C.dom h*) (*mkCone h k*) ⟷ *C.commutative-square f0 f1 h k*
⟨*proof*⟩

lemma *cones-map-mkCone-eq-iff*:
assumes *is-rendered-commutative-by p0 p1* **and** *is-rendered-commutative-by p0′ p1′*
and «*h* : *C.dom p0′* → *C.dom p0*»
shows *cones-map h* (*mkCone p0 p1*) = *mkCone p0′ p1′* ⟷ *p0* · *h* = *p0′* ∧ *p1* · *h* = *p1′*
⟨*proof*⟩

end

locale *pullback-cone* =
J: *cospan-shape* +
C: *category C* +
D: *cospan-diagram C f0 f1* +
limit-cone J.comp C D.map ‹*C.dom p0*› ‹*D.mkCone p0 p1*›
for *C* :: ′*c comp* (**infixr** · *55*)
and *f0* :: ′*c*
and *f1* :: ′*c*
and *p0* :: ′*c*
and *p1* :: ′*c*
begin

lemma *renders-commutative*:
shows *D.is-rendered-commutative-by p0 p1*
⟨*proof*⟩

lemma *is-universal′*:
assumes *D.is-rendered-commutative-by p0′ p1′*

shows $\exists!h.$ «h : $C.dom\ p0' \rightarrow C.dom\ p0$» $\wedge$ $p0 \cdot h = p0' \wedge p1 \cdot h = p1'$
⟨*proof*⟩

lemma *induced-arrowI'*:
assumes *D.is-rendered-commutative-by p0' p1'*
shows «*induced-arrow* (*C.dom p0'*) (*D.mkCone p0' p1'*) : *C.dom p0'* → *C.dom p0*»
and *p0* · *induced-arrow* (*C.dom p0'*) (*D.mkCone p0' p1'*) = *p0'*
and *p1* · *induced-arrow* (*C.dom p1'*) (*D.mkCone p0' p1'*) = *p1'*
⟨*proof*⟩

end

context *category*
begin

definition *has-as-pullback*
where *has-as-pullback f0 f1 p0 p1* ≡
cospan f0 f1 ∧ *cospan-diagram.has-as-pullback C f0 f1 p0 p1*

definition *has-pullbacks*
where *has-pullbacks* = (∀ *f0 f1*. *cospan f0 f1* ⟶ (∃ *p0 p1*. *has-as-pullback f0 f1 p0 p1*))

lemma *has-as-pullbackI* [*intro*]:
assumes *cospan f g* **and** *commutative-square f g p q*
and ⋀*h k*. *commutative-square f g h k* ⟹ ∃!*l*. *p* · *l* = *h* ∧ *q* · *l* = *k*
shows *has-as-pullback f g p q*
⟨*proof*⟩

lemma *has-as-pullbackE* [*elim*]:
assumes *has-as-pullback f g p q*
and ⟦*cospan f g*; *commutative-square f g p q*;
⋀*h k*. *commutative-square f g h k* ⟹ ∃!*l*. *p* · *l* = *h* ∧ *q* · *l* = *k*⟧ ⟹ *T*
shows *T*
⟨*proof*⟩

end

locale *category-with-pullbacks* =
category +
assumes *has-pullbacks*: *has-pullbacks*

21.4 Elementary Category with Pullbacks

An *elementary category with pullbacks* is a category equipped with a specific way of mapping each cospan to a span such that the resulting square commutes and such that the span is universal for that property. It is useful to assume that the functions, mapping a cospan to the two projections of the pullback, are extensional; that is, they yield *null* when applied to arguments that do not form a cospan.

locale *elementary-category-with-pullbacks* =
category C
for C :: $'a$ *comp* (**infixr** $\cdot$ *55*)
and *prj0* :: $'a \Rightarrow 'a \Rightarrow 'a$ ($\mathrm{p}_0[\text{-}, \text{-}]$)
and *prj1* :: $'a \Rightarrow 'a \Rightarrow 'a$ ($\mathrm{p}_1[\text{-}, \text{-}]$) +
assumes *prj0-ext*: $\neg$ *cospan f g* $\Longrightarrow \mathrm{p}_0[f, g] = null$
and *prj1-ext*: $\neg$ *cospan f g* $\Longrightarrow \mathrm{p}_1[f, g] = null$
and *pullback-commutes* [*intro*]: *cospan f g* $\Longrightarrow$ *commutative-square f g* $\mathrm{p}_1[f, g]\ \mathrm{p}_0[f, g]$
and *universal*: *commutative-square f g h k* $\Longrightarrow \exists!l.\ \mathrm{p}_1[f, g] \cdot l = h \wedge \mathrm{p}_0[f, g] \cdot l = k$
begin

lemma *pullback-commutes'*:
assumes *cospan f g*
shows $f \cdot \mathrm{p}_1[f, g] = g \cdot \mathrm{p}_0[f, g]$
⟨*proof*⟩

lemma *prj0-in-hom'*:
assumes *cospan f g*
shows «$\mathrm{p}_0[f, g]$: *dom* $\mathrm{p}_0[f, g] \rightarrow$ *dom g*»
⟨*proof*⟩

lemma *prj1-in-hom'*:
assumes *cospan f g*
shows «$\mathrm{p}_1[f, g]$: *dom* $\mathrm{p}_0[f, g] \rightarrow$ *dom f*»
⟨*proof*⟩

The following gives us a notation for the common domain of the two projections of a pullback.

definition *pbdom* (**infix** $\downarrow\downarrow$ *51*)
where $f \downarrow\downarrow g \equiv$ *dom* $\mathrm{p}_0[f, g]$

lemma *pbdom-in-hom* [*intro*]:
assumes *cospan f g*
shows «$f \downarrow\downarrow g : f \downarrow\downarrow g \rightarrow f \downarrow\downarrow g$»
⟨*proof*⟩

lemma *ide-pbdom* [*simp*]:
assumes *cospan f g*
shows *ide* $(f \downarrow\downarrow g)$
⟨*proof*⟩

lemma *prj0-in-hom* [*intro*, *simp*]:
assumes *cospan f g* **and** $a = f \downarrow\downarrow g$ **and** $b =$ *dom g*
shows «$\mathrm{p}_0[f, g] : a \rightarrow b$»
⟨*proof*⟩

lemma *prj1-in-hom* [*intro*, *simp*]:
assumes *cospan f g* **and** $a = f \downarrow\downarrow g$ **and** $b =$ *dom f*
shows «$\mathrm{p}_1[f, g] : a \rightarrow b$»

⟨*proof*⟩

lemma *prj0-simps* [*simp*]:
assumes *cospan f g*
shows *arr* $\mathrm{p}_0[f,\ g]$ **and** *dom* $\mathrm{p}_0[f,\ g] = f \downarrow\downarrow g$ **and** *cod* $\mathrm{p}_0[f,\ g] = dom\ g$
⟨*proof*⟩

lemma *prj0-simps-arr* [*iff*]:
shows *arr* $\mathrm{p}_0[f,\ g] \longleftrightarrow$ *cospan f g*
⟨*proof*⟩

lemma *prj1-simps* [*simp*]:
assumes *cospan f g*
shows *arr* $\mathrm{p}_1[f,\ g]$ **and** *dom* $\mathrm{p}_1[f,\ g] = f \downarrow\downarrow g$ **and** *cod* $\mathrm{p}_1[f,\ g] = dom\ f$
⟨*proof*⟩

lemma *prj1-simps-arr* [*iff*]:
shows *arr* $\mathrm{p}_1[f,\ g] \longleftrightarrow$ *cospan f g*
⟨*proof*⟩

lemma *span-prj*:
assumes *cospan f g*
shows *span* $\mathrm{p}_0[f,\ g]\ \mathrm{p}_1[f,\ g]$
⟨*proof*⟩

We introduce a notation for tupling, which produces the induced arrow into a pullback. In our notation, the "0-side", which we regard as the input, occurs on the right, and the "1-side", which we regard as the output, occurs on the left.

definition *tuple* (⟨- ⟦-, -⟧ -⟩)
where $\langle h\ [\![f,\ g]\!]\ k\rangle \equiv$ *if commutative-square f g h k then*
THE l. $\mathrm{p}_0[f,\ g] \cdot l = k \wedge \mathrm{p}_1[f,\ g] \cdot l = h$
else null

lemma *tuple-in-hom* [*intro*]:
assumes *commutative-square f g h k*
shows «$\langle h\ [\![f,\ g]\!]\ k\rangle : dom\ h \rightarrow f \downarrow\downarrow g$»
⟨*proof*⟩

lemma *tuple-is-extensional*:
assumes ¬ *commutative-square f g h k*
shows $\langle h\ [\![f,\ g]\!]\ k\rangle = null$
⟨*proof*⟩

lemma *tuple-simps* [*simp*]:
assumes *commutative-square f g h k*
shows *arr* $\langle h\ [\![f,\ g]\!]\ k\rangle$ **and** *dom* $\langle h\ [\![f,\ g]\!]\ k\rangle = dom\ h$ **and** *cod* $\langle h\ [\![f,\ g]\!]\ k\rangle = f \downarrow\downarrow g$
⟨*proof*⟩

lemma *prj-tuple* [*simp*]:

assumes *commutative-square f g h k*
shows $\mathrm{p}_0[f,\ g] \cdot \langle h\ [\![f,\ g]\!]\ k\rangle = k$ **and** $\mathrm{p}_1[f,\ g] \cdot \langle h\ [\![f,\ g]\!]\ k\rangle = h$
⟨*proof*⟩

lemma *tuple-prj*:
assumes *cospan f g* **and** *seq* $\mathrm{p}_1[f,\ g]\ h$
shows $\langle \mathrm{p}_1[f,\ g] \cdot h\ [\![f,\ g]\!]\ \mathrm{p}_0[f,\ g] \cdot h\rangle = h$
⟨*proof*⟩

lemma *tuple-prj-spc* [*simp*]:
assumes *cospan f g*
shows $\langle \mathrm{p}_1[f,\ g]\ [\![f,\ g]\!]\ \mathrm{p}_0[f,\ g]\rangle = f \downarrow\downarrow g$
⟨*proof*⟩

lemma *prj-joint-monic*:
assumes *cospan f g* **and** *seq* $\mathrm{p}_1[f,\ g]\ h$ **and** *seq* $\mathrm{p}_1[f,\ g]\ h'$
and $\mathrm{p}_0[f,\ g] \cdot h = \mathrm{p}_0[f,\ g] \cdot h'$ **and** $\mathrm{p}_1[f,\ g] \cdot h = \mathrm{p}_1[f,\ g] \cdot h'$
shows $h = h'$
⟨*proof*⟩

The pullback of an identity along an arbitrary arrow is an isomorphism.

lemma *iso-pullback-ide*:
assumes *cospan* $\mu\ \nu$ **and** *ide* μ
shows *iso* $\mathrm{p}_0[\mu,\ \nu]$
⟨*proof*⟩

lemma *comp-tuple-arr*:
assumes *commutative-square f g h k* **and** *seq h l*
shows $\langle h\ [\![f,\ g]\!]\ k\rangle \cdot l = \langle h \cdot l\ [\![f,\ g]\!]\ k \cdot l\rangle$
⟨*proof*⟩

lemma *pullback-arr-cod*:
assumes *arr f*
shows *inverse-arrows* $\mathrm{p}_1[f,\ \mathit{cod}\ f]\ \langle \mathit{dom}\ f\ [\![f,\ \mathit{cod}\ f]\!]\ f\rangle$
and *inverse-arrows* $\mathrm{p}_0[\mathit{cod}\ f,\ f]\ \langle f\ [\![\mathit{cod}\ f,\ f]\!]\ \mathit{dom}\ f\rangle$
⟨*proof*⟩

The pullback of a monomorphism along itself is automatically symmetric: the left and right projections are equal.

lemma *pullback-mono-self*:
assumes *mono f*
shows $\mathrm{p}_0[f,\ f] = \mathrm{p}_1[f,\ f]$
⟨*proof*⟩

lemma *pullback-iso-self*:
assumes *iso f*
shows $\mathrm{p}_0[f,\ f] = \mathrm{p}_1[f,\ f]$
⟨*proof*⟩

lemma *pullback-ide-self* [*simp*]:
assumes *ide a*
shows $\mathrm{p}_0[a,\ a] = \mathrm{p}_1[a,\ a]$
⟨*proof*⟩

end

21.5 Agreement between the Definitions

It is very easy to write locale assumptions that have unintended consequences or that are even inconsistent. So, to keep ourselves honest, we don't just accept the definition of "elementary category with pullbacks", but in fact we formally establish the sense in which it agrees with our standard definition of "category with pullbacks", which is given in terms of limit cones. This is extra work, but it ensures that we didn't make a mistake.

context *category-with-pullbacks*
begin

definition *some-prj1* ($\mathrm{p}_1{}^?$[-, -])
where $\mathrm{p}_1{}^?[f,\ g]$ ≡ *if cospan f g then*
fst (*SOME x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*))
else null

definition *some-prj0* ($\mathrm{p}_0{}^?$[-, -])
where $\mathrm{p}_0{}^?[f,\ g]$ ≡ *if cospan f g then*
snd (*SOME x. cospan-diagram.has-as-pullback C f g* (*fst x*) (*snd x*))
else null

lemma *prj-yields-pullback*:
assumes *cospan f g*
shows *cospan-diagram.has-as-pullback C f g* $\mathrm{p}_1{}^?[f,\ g]$ $\mathrm{p}_0{}^?[f,\ g]$
⟨*proof*⟩

interpretation *elementary-category-with-pullbacks C some-prj0 some-prj1*
⟨*proof*⟩

proposition *extends-to-elementary-category-with-pullbacks*:
shows *elementary-category-with-pullbacks C some-prj0 some-prj1*
⟨*proof*⟩

end

context *elementary-category-with-pullbacks*
begin

interpretation *category-with-pullbacks C*
⟨*proof*⟩

proposition *is-category-with-pullbacks*:

shows *category-with-pullbacks C*
⟨*proof*⟩

end

sublocale *elementary-category-with-pullbacks* ⊆ *category-with-pullbacks*
⟨*proof*⟩

end

Chapter 22

Cartesian Category

In this chapter, we explore the notion of a "cartesian category", which we define to be a category having binary products and a terminal object. We show that every cartesian category extends to an "elementary cartesian category", whose definition assumes that specific choices have been made for projections and terminal object. Conversely, the underlying category of an elementary cartesian category is a cartesian category. We also show that cartesian categories are the same thing as categories with finite products.

theory *CartesianCategory*
imports *Limit SetCat CategoryWithPullbacks*
begin

22.1 Category with Binary Products

22.1.1 Binary Product Diagrams

The "shape" of a binary product diagram is a category having two distinct identity arrows and no non-identity arrows.

locale *binary-product-shape*
begin

sublocale *concrete-category* ‹*UNIV* :: *bool set*› ‹$\lambda a\ b.$ *if* $a = b$ *then* $\{()\}$ *else* $\{\}$›
‹λ-. ()› ‹λ- - - - -. ()›
⟨*proof*⟩

abbreviation *comp*
where *comp* ≡ *COMP*

abbreviation *FF*
where *FF* ≡ *MkIde False*

abbreviation *TT*
where *TT* ≡ *MkIde True*

lemma *arr-char*:
shows $arr\ f \longleftrightarrow f = FF \vee f = TT$
⟨*proof*⟩

lemma *ide-char*:
shows $ide\ f \longleftrightarrow f = FF \vee f = TT$
⟨*proof*⟩

lemma *is-discrete*:
shows $ide\ f \longleftrightarrow arr\ f$
⟨*proof*⟩

lemma *dom-simp* [*simp*]:
assumes *arr f*
shows $dom\ f = f$
⟨*proof*⟩

lemma *cod-simp* [*simp*]:
assumes *arr f*
shows $cod\ f = f$
⟨*proof*⟩

lemma *seq-char*:
shows $seq\ f\ g \longleftrightarrow arr\ f \wedge f = g$
⟨*proof*⟩

lemma *comp-simp* [*simp*]:
assumes *seq f g*
shows $comp\ f\ g = f$
⟨*proof*⟩

end

locale *binary-product-diagram* =
J: *binary-product-shape* +
C: *category C*
for $C :: 'c\ comp$ (**infixr** $\cdot$ *55*)
and $a0 :: 'c$
and $a1 :: 'c$ +
assumes *is-discrete*: $C.ide\ a0 \wedge C.ide\ a1$
begin

notation *J.comp* (**infixr** $\cdot_J$ *55*)

fun *map*
where $map\ J.FF = a0$
| $map\ J.TT = a1$
| map - $= C.null$

sublocale *diagram J.comp C map*
⟨*proof*⟩

end

22.1.2 Category with Binary Products

A *binary product* in a category C is a limit of a binary product diagram in C.

context *binary-product-diagram*
begin

definition *mkCone*
where *mkCone p0 p1* ≡ *λj. if j = J.FF then p0 else if j = J.TT then p1 else C.null*

abbreviation *is-rendered-commutative-by*
where *is-rendered-commutative-by p0 p1* ≡
C.seq a0 p0 ∧ *C.seq a1 p1* ∧ *C.dom p0* = *C.dom p1*

abbreviation *has-as-binary-product*
where *has-as-binary-product p0 p1* ≡ *limit-cone* (*C.dom p0*) (*mkCone p0 p1*)

lemma *cone-mkCone*:
assumes *is-rendered-commutative-by p0 p1*
shows *cone* (*C.dom p0*) (*mkCone p0 p1*)
⟨*proof*⟩

lemma *is-rendered-commutative-by-cone*:
assumes *cone a χ*
shows *is-rendered-commutative-by* (*χ J.FF*) (*χ J.TT*)
⟨*proof*⟩

lemma *mkCone-cone*:
assumes *cone a χ*
shows *mkCone* (*χ J.FF*) (*χ J.TT*) = *χ*
⟨*proof*⟩

lemma *cone-iff-span*:
shows *cone* (*C.dom h*) (*mkCone h k*) ⟷ *C.span h k* ∧ *C.cod h* = *a0* ∧ *C.cod k* = *a1*
⟨*proof*⟩

lemma *cones-map-mkCone-eq-iff*:
assumes *is-rendered-commutative-by p0 p1* **and** *is-rendered-commutative-by p0′ p1′*
and «*h* : *C.dom p0′* → *C.dom p0*»
shows *cones-map h* (*mkCone p0 p1*) = *mkCone p0′ p1′* ⟷ *p0* · *h* = *p0′* ∧ *p1* · *h* = *p1′*
⟨*proof*⟩

end

locale *binary-product-cone* =

J: *binary-product-shape* +
C: *category C* +
D: *binary-product-diagram C f0 f1* +
limit-cone J.comp C D.map ‹C.dom p0› ‹D.mkCone p0 p1›
for *C* :: *'c comp* (**infixr** · *55*)
and *f0* :: *'c*
and *f1* :: *'c*
and *p0* :: *'c*
and *p1* :: *'c*
begin

lemma *renders-commutative*:
shows *D.is-rendered-commutative-by p0 p1*
⟨*proof*⟩

lemma *is-universal'*:
assumes *D.is-rendered-commutative-by p0' p1'*
shows ∃!*h*. «*h* : *C.dom p0'* → *C.dom p0*» ∧ *p0* · *h* = *p0'* ∧ *p1* · *h* = *p1'*
⟨*proof*⟩

lemma *induced-arrowI'*:
assumes *D.is-rendered-commutative-by p0' p1'*
shows «*induced-arrow* (*C.dom p0'*) (*D.mkCone p0' p1'*) : *C.dom p0'* → *C.dom p0*»
and *p0* · *induced-arrow* (*C.dom p0'*) (*D.mkCone p0' p1'*) = *p0'*
and *p1* · *induced-arrow* (*C.dom p1'*) (*D.mkCone p0' p1'*) = *p1'*
⟨*proof*⟩

end

context *category*
begin

definition *has-as-binary-product*
where *has-as-binary-product a0 a1 p0 p1* ≡
ide a0 ∧ *ide a1* ∧ *binary-product-diagram.has-as-binary-product C a0 a1 p0 p1*

definition *has-binary-products*
where *has-binary-products* =
(∀ *a0 a1*. *ide a0* ∧ *ide a1* ⟶ (∃ *p0 p1*. *has-as-binary-product a0 a1 p0 p1*))

lemma *has-as-binary-productI* [*intro*]:
assumes *ide a* **and** *ide b*
and «*p* : *c* → *a*» **and** «*q* : *c* → *b*»
and ⋀*x f g*. ⟦«*f* : *x* → *a*»; «*g* : *x* → *b*»⟧ ⟹ ∃!*h*. «*h* : *x* → *c*» ∧ *p* · *h* = *f* ∧ *q* · *h* = *g*
shows *has-as-binary-product a b p q*
⟨*proof*⟩

lemma *has-as-binary-productE* [*elim*]:
assumes *has-as-binary-product a b p q*

and $⟦«p : dom\ p \to a»; «q : dom\ p \to b»;$
$\bigwedge x\ f\ g.\ ⟦«f : x \to a»; «g : x \to b»⟧ \Longrightarrow \exists!h.\ p \cdot h = f \wedge q \cdot h = g⟧ \Longrightarrow T$
shows T
⟨*proof*⟩

end

locale *category-with-binary-products* =
category +
assumes *has-binary-products*: *has-binary-products*

22.1.3 Elementary Category with Binary Products

An *elementary category with binary products* is a category equipped with a specific way of mapping each pair of objects a and b to a pair of arrows $\mathfrak{p}_1[a,\ b]$ and $\mathfrak{p}_0[a,\ b]$ that comprise a universal span.

locale *elementary-category-with-binary-products* =
category C
for C :: *'a comp* (**infixr** $\cdot$ *55*)
and *pr0* :: $'a \Rightarrow 'a \Rightarrow 'a$ ($\mathfrak{p}_0$[-, -])
and *pr1* :: $'a \Rightarrow 'a \Rightarrow 'a$ ($\mathfrak{p}_1$[-, -]) +
assumes *span-pr*: $⟦\ ide\ a;\ ide\ b\ ⟧ \Longrightarrow span\ \mathfrak{p}_1[a,\ b]\ \mathfrak{p}_0[a,\ b]$
and *cod-pr0*: $⟦\ ide\ a;\ ide\ b\ ⟧ \Longrightarrow cod\ \mathfrak{p}_0[a,\ b] = b$
and *cod-pr1*: $⟦\ ide\ a;\ ide\ b\ ⟧ \Longrightarrow cod\ \mathfrak{p}_1[a,\ b] = a$
and *universal*: $span\ f\ g \Longrightarrow \exists!l.\ \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$
begin

lemma *pr0-in-hom'*:
assumes *ide a* **and** *ide b*
shows $«\mathfrak{p}_0[a,\ b] : dom\ \mathfrak{p}_0[a,\ b] \to b»$
⟨*proof*⟩

lemma *pr1-in-hom'*:
assumes *ide a* **and** *ide b*
shows $«\mathfrak{p}_1[a,\ b] : dom\ \mathfrak{p}_0[a,\ b] \to a»$
⟨*proof*⟩

We introduce a notation for tupling, which denotes the arrow into a product that is induced by a span.

definition *tuple* (⟨-, -⟩)
where $\langle f,\ g\rangle \equiv$ *if span f g then*
$THE\ l.\ \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$
else null

The following defines product of arrows (not just of objects). It will take a little while before we can prove that it is functorial, but for right now it is nice to have it as a notation for the apex of a product cone. We have to go through some slightly unnatural contortions in the development here, though, to avoid having to introduce a separate preliminary notation just for the product of objects.

definition *prod* (**infixr** $\otimes$ *51*)
where $f \otimes g \equiv \langle f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g],\ g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g]\rangle$

lemma *seq-pr-tuple*:
assumes *span f g*
shows $seq\ \mathfrak{p}_0[cod\ f,\ cod\ g]\ \langle f,\ g\rangle$
⟨*proof*⟩

lemma *tuple-pr-arr*:
assumes *ide a* **and** *ide b* **and** $seq\ \mathfrak{p}_0[a,\ b]\ h$
shows $\langle \mathfrak{p}_1[a,\ b] \cdot h,\ \mathfrak{p}_0[a,\ b] \cdot h\rangle = h$
⟨*proof*⟩

lemma *pr-tuple* [*simp*]:
assumes *span f g* **and** $cod\ f = a$ **and** $cod\ g = b$
shows $\mathfrak{p}_1[a,\ b] \cdot \langle f,\ g\rangle = f$ **and** $\mathfrak{p}_0[a,\ b] \cdot \langle f,\ g\rangle = g$
⟨*proof*⟩

lemma *cod-tuple*:
assumes *span f g*
shows $cod\ \langle f,\ g\rangle = cod\ f \otimes cod\ g$
⟨*proof*⟩

lemma *tuple-in-hom* [*intro*]:
assumes «$f : a \rightarrow b$» **and** «$g : a \rightarrow c$»
shows «$\langle f,\ g\rangle : a \rightarrow b \otimes c$»
⟨*proof*⟩

lemma *tuple-in-hom′* [*simp*]:
assumes *arr f* **and** $dom\ f = a$ **and** $cod\ f = b$
and *arr g* **and** $dom\ g = a$ **and** $cod\ g = c$
shows «$\langle f,\ g\rangle : a \rightarrow b \otimes c$»
⟨*proof*⟩

lemma *tuple-ext*:
assumes $\neg$ *span f g*
shows $\langle f,\ g\rangle = null$
⟨*proof*⟩

lemma *tuple-simps* [*simp*]:
assumes *span f g*
shows $arr\ \langle f,\ g\rangle$ **and** $dom\ \langle f,\ g\rangle = dom\ f$ **and** $cod\ \langle f,\ g\rangle = cod\ f \otimes cod\ g$
⟨*proof*⟩

lemma *tuple-pr* [*simp*]:
assumes *ide a* **and** *ide b*
shows $\langle \mathfrak{p}_1[a,\ b],\ \mathfrak{p}_0[a,\ b]\rangle = a \otimes b$
⟨*proof*⟩

lemma *pr-in-hom* [*intro*, *simp*]:
assumes *ide a* **and** *ide b* **and** $x = a \otimes b$
shows «$\mathfrak{p}_0[a, b] : x \to b$» **and** «$\mathfrak{p}_1[a, b] : x \to a$»
⟨*proof*⟩

lemma *pr-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* $\mathfrak{p}_0[a, b]$ **and** *dom* $\mathfrak{p}_0[a, b] = a \otimes b$ **and** *cod* $\mathfrak{p}_0[a, b] = b$
and *arr* $\mathfrak{p}_1[a, b]$ **and** *dom* $\mathfrak{p}_1[a, b] = a \otimes b$ **and** *cod* $\mathfrak{p}_1[a, b] = a$
⟨*proof*⟩

lemma *pr-joint-monic*:
assumes *ide a* **and** *ide b* **and** *seq* $\mathfrak{p}_0[a, b]$ h
and $\mathfrak{p}_0[a, b] \cdot h = \mathfrak{p}_0[a, b] \cdot h'$ **and** $\mathfrak{p}_1[a, b] \cdot h = \mathfrak{p}_1[a, b] \cdot h'$
shows $h = h'$
⟨*proof*⟩

lemma *comp-tuple-arr* [*simp*]:
assumes *span f g* **and** *arr h* **and** *dom f* = *cod h*
shows $\langle f, g\rangle \cdot h = \langle f \cdot h, g \cdot h\rangle$
⟨*proof*⟩

lemma *ide-prod* [*intro*, *simp*]:
assumes *ide a* **and** *ide b*
shows *ide* $(a \otimes b)$
⟨*proof*⟩

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

lemma *prod-in-hom'* [*simp*]:
assumes *arr f* **and** *dom f* = *a* **and** *cod f* = *c*
and *arr g* **and** *dom g* = *b* **and** *cod g* = *d*
shows «$f \otimes g : a \otimes b \to c \otimes d$»
⟨*proof*⟩

lemma *prod-simps* [*simp*]:
assumes *arr f0* **and** *arr f1*
shows *arr* (*f0* ⊗ *f1*)
and *dom* (*f0* ⊗ *f1*) = *dom f0* ⊗ *dom f1*
and *cod* (*f0* ⊗ *f1*) = *cod f0* ⊗ *cod f1*
⟨*proof*⟩

lemma *has-as-binary-product*:
assumes *ide a* **and** *ide b*
shows *has-as-binary-product a b* $\mathfrak{p}_1[a, b]$ $\mathfrak{p}_0[a, b]$
⟨*proof*⟩

end

22.1.4 Agreement between the Definitions

We now show that a category with binary products extends (by making a choice) to an elementary category with binary products, and that the underlying category of an elementary category with binary products is a category with binary products.

context *category-with-binary-products*
begin

definition *some-pr1* ($\mathfrak{p}_1{}^?[\text{-}, \text{-}]$)
where *some-pr1 a b* ≡ *if ide a* ∧ *ide b then*
fst (*SOME x. has-as-binary-product a b* (*fst x*) (*snd x*))
else null

definition *some-pr0* ($\mathfrak{p}_0{}^?[\text{-}, \text{-}]$)
where *some-pr0 a b* ≡ *if ide a* ∧ *ide b then*
snd (*SOME x. has-as-binary-product a b* (*fst x*) (*snd x*))
else null

lemma *pr-yields-binary-product*:
assumes *ide a* **and** *ide b*
shows *has-as-binary-product a b* $\mathfrak{p}_1{}^?[a, b]$ $\mathfrak{p}_0{}^?[a, b]$
⟨*proof*⟩

interpretation *elementary-category-with-binary-products C some-pr0 some-pr1*
⟨*proof*⟩

proposition *extends-to-elementary-category-with-binary-products*:
shows *elementary-category-with-binary-products C some-pr0 some-pr1*
⟨*proof*⟩

abbreviation *some-prod* (**infixr** $\otimes^?$ *51*)
where *some-prod* ≡ *prod*

end

context *elementary-category-with-binary-products*
begin

sublocale *category-with-binary-products C*
⟨*proof*⟩

proposition *is-category-with-binary-products*:
shows *category-with-binary-products C*
⟨*proof*⟩

end

22.1.5 Further Properties

context *elementary-category-with-binary-products*
begin

lemma *interchange*:
assumes *seq h f* **and** *seq k g*
shows $(h \otimes k) \cdot (f \otimes g) = h \cdot f \otimes k \cdot g$
⟨*proof*⟩

lemma *pr-naturality* [*simp*]:
assumes *arr g* **and** *dom g* = *b* **and** *cod g* = *d*
and *arr f* **and** *dom f* = *a* **and** *cod f* = *c*
shows $\mathfrak{p}_0[c, d] \cdot (f \otimes g) = g \cdot \mathfrak{p}_0[a, b]$
and $\mathfrak{p}_1[c, d] \cdot (f \otimes g) = f \cdot \mathfrak{p}_1[a, b]$
⟨*proof*⟩

abbreviation *dup* (d[-])
where d[*f*] ≡ ⟨*f*, *f*⟩

lemma *dup-in-hom* [*intro*, *simp*]:
assumes «*f* : *a* → *b*»
shows «d[*f*] : *a* → *b* ⊗ *b*»
⟨*proof*⟩

lemma *dup-simps* [*simp*]:
assumes *arr f*
shows *arr* d[*f*] **and** *dom* d[*f*] = *dom f* **and** *cod* d[*f*] = *cod f* ⊗ *cod f*
⟨*proof*⟩

lemma *dup-naturality*:
assumes «*f* : *a* → *b*»
shows $\mathrm{d}[b] \cdot f = (f \otimes f) \cdot \mathrm{d}[a]$
⟨*proof*⟩

lemma *pr-dup* [*simp*]:
assumes *ide a*
shows $\mathfrak{p}_0[a, a] \cdot \mathrm{d}[a] = a$ **and** $\mathfrak{p}_1[a, a] \cdot \mathrm{d}[a] = a$
⟨*proof*⟩

lemma *prod-tuple*:
assumes *span f g* **and** *seq h f* **and** *seq k g*
shows $(h \otimes k) \cdot \langle f, g\rangle = \langle h \cdot f, k \cdot g\rangle$
⟨*proof*⟩

lemma *tuple-eqI*:
assumes *ide b* **and** *ide c* **and** *seq* $\mathfrak{p}_0[b, c]$ *f* **and** *seq* $\mathfrak{p}_1[b, c]$ *f*
and $\mathfrak{p}_0[b, c] \cdot f = f0$ **and** $\mathfrak{p}_1[b, c] \cdot f = f1$
shows $f = \langle f1, f0\rangle$
⟨*proof*⟩

lemma *tuple-expansion*:
assumes *span f g*
shows $(f \otimes g) \cdot \mathrm{d}[dom\ f] = \langle f, g\rangle$
⟨*proof*⟩

definition *assoc* (a[-, -, -])
where $\mathrm{a}[a, b, c] \equiv \langle \mathfrak{p}_1[a, b] \cdot \mathfrak{p}_1[a \otimes b, c], \langle \mathfrak{p}_0[a, b] \cdot \mathfrak{p}_1[a \otimes b, c], \mathfrak{p}_0[a \otimes b, c]\rangle\rangle$

definition *assoc′* (a^{-1}[-, -, -])
where $\mathrm{a}^{-1}[a, b, c] \equiv \langle\langle \mathfrak{p}_1[a, b \otimes c], \mathfrak{p}_1[b, c] \cdot \mathfrak{p}_0[a, b \otimes c]\rangle, \mathfrak{p}_0[b, c] \cdot \mathfrak{p}_0[a, b \otimes c]\rangle$

lemma *assoc-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «$\mathrm{a}[a, b, c] : (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$»
⟨*proof*⟩

lemma *assoc-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* $\mathrm{a}[a, b, c]$
and *dom* $\mathrm{a}[a, b, c] = (a \otimes b) \otimes c$
and *cod* $\mathrm{a}[a, b, c] = a \otimes (b \otimes c)$
⟨*proof*⟩

lemma *assoc′-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «$\mathrm{a}^{-1}[a, b, c] : a \otimes (b \otimes c) \rightarrow (a \otimes b) \otimes c$»
⟨*proof*⟩

lemma *assoc′-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* $\mathrm{a}^{-1}[a, b, c]$
and *dom* $\mathrm{a}^{-1}[a, b, c] = a \otimes (b \otimes c)$
and *cod* $\mathrm{a}^{-1}[a, b, c] = (a \otimes b) \otimes c$
⟨*proof*⟩

lemma *pr-assoc*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathfrak{p}_0[b, c] \cdot \mathfrak{p}_0[a, b \otimes c] \cdot \mathrm{a}[a, b, c] = \mathfrak{p}_0[a \otimes b, c]$
and $\mathfrak{p}_1[b, c] \cdot \mathfrak{p}_0[a, b \otimes c] \cdot \mathrm{a}[a, b, c] = \mathfrak{p}_0[a, b] \cdot \mathfrak{p}_1[a \otimes b, c]$
and $\mathfrak{p}_1[a, b \otimes c] \cdot \mathrm{a}[a, b, c] = \mathfrak{p}_1[a, b] \cdot \mathfrak{p}_1[a \otimes b, c]$
⟨*proof*⟩

lemma *pr-assoc′*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathfrak{p}_1[a, b] \cdot \mathfrak{p}_1[a \otimes b, c] \cdot \mathrm{a}^{-1}[a, b, c] = \mathfrak{p}_1[a, b \otimes c]$
and $\mathfrak{p}_0[a, b] \cdot \mathfrak{p}_1[a \otimes b, c] \cdot \mathrm{a}^{-1}[a, b, c] = \mathfrak{p}_1[b, c] \cdot \mathfrak{p}_0[a, b \otimes c]$
and $\mathfrak{p}_0[a \otimes b, c] \cdot \mathrm{a}^{-1}[a, b, c] = \mathfrak{p}_0[b, c] \cdot \mathfrak{p}_0[a, b \otimes c]$
⟨*proof*⟩

lemma *assoc-naturality*:
assumes «$f0 : a0 \rightarrow b0$» **and** «$f1 : a1 \rightarrow b1$» **and** «$f2 : a2 \rightarrow b2$»
shows $\mathrm{a}[b0, b1, b2] \cdot ((f0 \otimes f1) \otimes f2) = (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[a0, a1, a2]$
⟨*proof*⟩

lemma *pentagon*:
assumes *ide a* **and** *ide b* **and** *ide c* **and** *ide d*
shows $((a \otimes \mathrm{a}[b, c, d]) \cdot \mathrm{a}[a, b \otimes c, d]) \cdot (\mathrm{a}[a, b, c] \otimes d) = \mathrm{a}[a, b, c \otimes d] \cdot \mathrm{a}[a \otimes b, c, d]$
⟨*proof*⟩

lemma *inverse-arrows-assoc*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *inverse-arrows* $\mathrm{a}[a, b, c]$ $\mathrm{a}^{-1}[a, b, c]$
⟨*proof*⟩

lemma *inv-prod*:
assumes *iso f* **and** *iso g*
shows *iso* (*prod f g*)
and *inv* (*prod f g*) = *prod* (*inv f*) (*inv g*)
⟨*proof*⟩

interpretation *CC*: *product-category C C* ⟨*proof*⟩

abbreviation *Prod*
where *Prod fg* ≡ *fst fg* ⊗ *snd fg*
abbreviation *Prod′*
where *Prod′ fg* ≡ *snd fg* ⊗ *fst fg*

interpretation Π: *binary-functor C C C Prod*
⟨*proof*⟩

interpretation *Prod′*: *binary-functor C C C Prod′*
⟨*proof*⟩

lemma *binary-functor-Prod*:
shows *binary-functor C C C Prod* **and** *binary-functor C C C Prod′*
⟨*proof*⟩

interpretation *CCC*: *product-category C CC.comp* ⟨*proof*⟩
interpretation *T*: *binary-endofunctor C Prod* ⟨*proof*⟩
interpretation *ToTC*: *functor CCC.comp C T.ToTC*
⟨*proof*⟩
interpretation *ToCT*: *functor CCC.comp C T.ToCT*
⟨*proof*⟩

abbreviation α
where $\alpha\ f \equiv \mathrm{a}[cod\ (fst\ f), cod\ (fst\ (snd\ f)), cod\ (snd\ (snd\ f))] \cdot$
$((fst\ f \otimes fst\ (snd\ f)) \otimes snd\ (snd\ f))$

lemma *α-simp-ide*:
assumes *CCC.ide a*
shows $\alpha\ a = \mathrm{a}[fst\ a,\ fst\ (snd\ a),\ snd\ (snd\ a)]$
⟨*proof*⟩

interpretation *α*: *natural-isomorphism CCC.comp C T.ToTC T.ToCT α*
⟨*proof*⟩

lemma *α-is-natural-isomorphism*:
shows *natural-isomorphism CCC.comp C T.ToTC T.ToCT α*
⟨*proof*⟩

definition *sym* (s[-, -])
where $\mathrm{s}[a1,\ a0] \equiv$ *if ide a0* $\wedge$ *ide a1 then* $\langle \mathfrak{p}_0[a1,\ a0],\ \mathfrak{p}_1[a1,\ a0]\rangle$ *else null*

lemma *sym-in-hom* [*intro*]:
assumes *ide a* **and** *ide b*
shows «$\mathrm{s}[a,\ b] : a \otimes b \to b \otimes a$»
⟨*proof*⟩

lemma *sym-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* $\mathrm{s}[a,\ b]$ **and** *dom* $\mathrm{s}[a,\ b] = a \otimes b$ **and** *cod* $\mathrm{s}[a,\ b] = b \otimes a$
⟨*proof*⟩

lemma *comp-sym-tuple* [*simp*]:
assumes «$f0 : a \to b0$» **and** «$f1 : a \to b1$»
shows $\mathrm{s}[b0,\ b1] \cdot \langle f0,\ f1\rangle = \langle f1,\ f0\rangle$
⟨*proof*⟩

lemma *prj-sym* [*simp*]:
assumes *ide a0* **and** *ide a1*
shows $\mathfrak{p}_0[a1,\ a0] \cdot \mathrm{s}[a0,\ a1] = \mathfrak{p}_1[a0,\ a1]$
and $\mathfrak{p}_1[a1,\ a0] \cdot \mathrm{s}[a0,\ a1] = \mathfrak{p}_0[a0,\ a1]$
⟨*proof*⟩

lemma *comp-sym-sym* [*simp*]:
assumes *ide a0* **and** *ide a1*
shows $\mathrm{s}[a1,\ a0] \cdot \mathrm{s}[a0,\ a1] = (a0 \otimes a1)$
⟨*proof*⟩

lemma *sym-inverse-arrows*:
assumes *ide a0* **and** *ide a1*
shows *inverse-arrows* $\mathrm{s}[a0,\ a1]\ \mathrm{s}[a1,\ a0]$
⟨*proof*⟩

lemma *sym-assoc-coherence*:
assumes *ide a* **and** *ide b* **and** *ide c*

shows a[b, c, a] · s[a, b ⊗ c] · a[a, b, c] = (b ⊗ s[a, c]) · a[b, a, c] · (s[a, b] ⊗ c)
⟨*proof*⟩

lemma *sym-naturality*:
assumes «*f0* : *a0* → *b0*» **and** «*f1* : *a1* → *b1*»
shows s[*b0*, *b1*] · (*f0* ⊗ *f1*) = (*f1* ⊗ *f0*) · s[*a0*, *a1*]
⟨*proof*⟩

abbreviation σ
where σ *fg* ≡ s[*cod* (*fst fg*), *cod* (*snd fg*)] · (*fst fg* ⊗ *snd fg*)

interpretation σ: *natural-transformation CC.comp C Prod Prod′* σ
⟨*proof*⟩

lemma σ*-is-natural-transformation*:
shows *natural-transformation CC.comp C Prod Prod′* σ
⟨*proof*⟩

abbreviation *Diag*
where *Diag f* ≡ *if arr f then* (*f*, *f*) *else CC.null*

interpretation Δ: *functor C CC.comp Diag*
⟨*proof*⟩

lemma *functor-Diag*:
shows *functor C CC.comp Diag*
⟨*proof*⟩

interpretation Δ*o*Π: *composite-functor CC.comp C CC.comp Prod Diag* ⟨*proof*⟩
interpretation Π*o*Δ: *composite-functor C CC.comp C Diag Prod* ⟨*proof*⟩

abbreviation π
where $\pi \equiv \lambda(f, g)$. ($\mathfrak{p}_1$[*cod f*, *cod g*] · (*f* ⊗ *g*), $\mathfrak{p}_0$[*cod f*, *cod g*] · (*f* ⊗ *g*))

interpretation π: *transformation-by-components CC.comp CC.comp* Δ*o*Π*.map CC.map* π
⟨*proof*⟩

lemma π*-is-natural-transformation*:
shows *natural-transformation CC.comp CC.comp* Δ*o*Π*.map CC.map* π
⟨*proof*⟩

interpretation δ: *natural-transformation C C map* Π*o*Δ*.map dup*
⟨*proof*⟩

lemma *dup-is-natural-transformation*:
shows *natural-transformation C C map* Π*o*Δ*.map dup*
⟨*proof*⟩

interpretation Δ*o*Π*o*Δ: *composite-functor C CC.comp CC.comp Diag* Δ*o*Π*.map* ⟨*proof*⟩

interpretation ΠoΔoΠ: *composite-functor CC.comp C C Prod* ΠoΔ*.map* ⟨*proof*⟩

interpretation Δoδ: *natural-transformation C CC.comp Diag* ΔoΠoΔ*.map* ‹*Diag* ∘ *dup*›
⟨*proof*⟩

interpretation δoΠ: *natural-transformation CC.comp C Prod* ΠoΔoΠ*.map* ‹*dup* ∘ *Prod*›
⟨*proof*⟩

interpretation πoΔ: *natural-transformation C CC.comp* ΔoΠoΔ*.map Diag* ‹π*.map* ∘ *Diag*›
⟨*proof*⟩

interpretation Πoπ: *natural-transformation CC.comp C* ΠoΔoΠ*.map Prod* ‹*Prod* ∘ π*.map*›
⟨*proof*⟩

interpretation Δoδ-πoΔ: *vertical-composite C CC.comp Diag* ΔoΠoΔ*.map Diag*
‹*Diag* ∘ *dup*› ‹π*.map* ∘ *Diag*›
⟨*proof*⟩
interpretation Πoπ-δoΠ: *vertical-composite CC.comp C Prod* ΠoΔoΠ*.map Prod*
‹*dup* ∘ *Prod*› ‹*Prod* ∘ π*.map*›
⟨*proof*⟩

interpretation ΔΠ: *unit-counit-adjunction CC.comp C Diag Prod dup* π*.map*
⟨*proof*⟩

proposition *induces-unit-counit-adjunction*:
shows *unit-counit-adjunction CC.comp C Diag Prod dup* π*.map*
⟨*proof*⟩

end

22.2 Category with Terminal Object

locale *category-with-terminal-object* =
category +
assumes *has-terminal*: ∃ *t*. *terminal t*

locale *elementary-category-with-terminal-object* =
category C
for *C* :: ′*a comp* (**infixr** · *55*)
and *one* :: ′*a* (**1**)
and *trm* :: ′*a* ⇒ ′*a* (t[-]) +
assumes *ide-one*: *ide* **1**
and *trm-in-hom* [*intro*, *simp*]: *ide a* ⟹ «t[*a*] : *a* → **1**»
and *trm-eqI*: ⟦ *ide a*; «*f* : *a* → **1**» ⟧ ⟹ *f* = t[*a*]
begin

lemma *trm-simps* [*simp*]:
assumes *ide a*
shows *arr* t[*a*] **and** *dom* t[*a*] = *a* **and** *cod* t[*a*] = **1**

⟨*proof*⟩

lemma *trm-one*:
shows t[**1**] = **1**
⟨*proof*⟩

lemma *terminal-one*:
shows *terminal* **1**
⟨*proof*⟩

lemma *trm-naturality*:
assumes *arr f*
shows t[*cod f*] · *f* = t[*dom f*]
⟨*proof*⟩

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

proposition *is-category-with-terminal-object*:
shows *category-with-terminal-object C*
⟨*proof*⟩

definition *τ*
where *τ* = (*λf. if arr f then trm* (*dom f*) *else null*)

lemma *τ-in-hom* [*intro*, *simp*]:
assumes *arr f*
shows «*τ f* : *dom f* → **1**»
⟨*proof*⟩

lemma *τ-simps* [*simp*]:
assumes *arr f*
shows *arr* (*τ f*) **and** *dom* (*τ f*) = *dom f* **and** *cod* (*τ f*) = **1**
⟨*proof*⟩

sublocale Ω: *constant-functor C C* **1**
⟨*proof*⟩

sublocale *τ*: *natural-transformation C C map* Ω.*map τ*
⟨*proof*⟩

end

context *category-with-terminal-object*
begin

definition *some-terminal* ($\mathbf{1}^?$)
where *some-terminal* ≡ *SOME t. terminal t*

definition *some-terminator* ($\mathrm{t}^?[\text{-}]$)
where $\mathrm{t}^?[f] \equiv$ *if arr f then THE t.* «$t : \mathit{dom}\ f \to \mathbf{1}^?$» *else null*

lemma *terminal-some-terminal* [*intro*]:
shows *terminal* $\mathbf{1}^?$
⟨*proof*⟩

lemma *ide-some-terminal*:
shows *ide* $\mathbf{1}^?$
⟨*proof*⟩

lemma *some-trm-in-hom* [*intro*]:
assumes *arr f*
shows «$\mathrm{t}^?[f] : \mathit{dom}\ f \to \mathbf{1}^?$»
⟨*proof*⟩

lemma *some-trm-simps* [*simp*]:
assumes *arr f*
shows *arr* $\mathrm{t}^?[f]$ **and** *dom* $\mathrm{t}^?[f] = \mathit{dom}\ f$ **and** *cod* $\mathrm{t}^?[f] = \mathbf{1}^?$
⟨*proof*⟩

lemma *some-trm-eqI*:
assumes «$t : \mathit{dom}\ f \to \mathbf{1}^?$»
shows $t = \mathrm{t}^?[f]$
⟨*proof*⟩

proposition *extends-to-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object C* $\mathbf{1}^?$ $(\lambda a.\ \mathrm{t}^?[a])$
⟨*proof*⟩

end

22.3 Cartesian Category

locale *cartesian-category* =
category-with-binary-products +
category-with-terminal-object

locale *category-with-pullbacks-and-terminal-object* =
category-with-pullbacks +
category-with-terminal-object
begin

sublocale *category-with-binary-products C*
⟨*proof*⟩

sublocale *cartesian-category C* ⟨*proof*⟩

end

locale *elementary-cartesian-category* =
elementary-category-with-binary-products +
elementary-category-with-terminal-object
begin

sublocale *cartesian-category C*
⟨*proof*⟩

proposition *is-cartesian-category*:
shows *cartesian-category C*
⟨*proof*⟩

end

context *cartesian-category*
begin

proposition *extends-to-elementary-cartesian-category*:
shows *elementary-cartesian-category C some-pr0 some-pr1* $\mathbf{1}^?$ $(\lambda a.\ \mathrm{t}^?[a])$
⟨*proof*⟩

end

22.3.1 Monoidal Structure

Here we prove some facts that will later allow us to show that an elementary cartesian category is a monoidal category.

context *elementary-cartesian-category*
begin

abbreviation ι
where $\iota \equiv \mathfrak{p}_0[\mathbf{1}, \mathbf{1}]$

lemma *pr-coincidence*:
shows $\iota = \mathfrak{p}_1[\mathbf{1}, \mathbf{1}]$
⟨*proof*⟩

lemma *unit-is-terminal-arr*:
shows *terminal-arr* ι
⟨*proof*⟩

lemma *unit-eq-trm*:
shows $\iota = \mathrm{t}[\mathbf{1} \otimes \mathbf{1}]$
⟨*proof*⟩

lemma *inverse-arrows-ι*:
shows *inverse-arrows* ι $\langle\mathbf{1}, \mathbf{1}\rangle$
⟨*proof*⟩

lemma *ι-is-iso*:
shows *iso* ι
⟨*proof*⟩

lemma *trm-tensor*:
assumes *ide a* **and** *ide b*
shows $\mathrm{t}[a \otimes b] = \iota \cdot (\mathrm{t}[a] \otimes \mathrm{t}[b])$
⟨*proof*⟩

abbreviation *runit* (r[-])
where $\mathrm{r}[a] \equiv \mathfrak{p}_1[a, \mathbf{1}]$

abbreviation *runit'* (r^{-1}[-])
where $\mathrm{r}^{-1}[a] \equiv \langle a, \mathrm{t}[a]\rangle$

abbreviation *lunit* (l[-])
where $\mathrm{l}[a] \equiv \mathfrak{p}_0[\mathbf{1}, a]$

abbreviation *lunit'* (l^{-1}[-])
where $\mathrm{l}^{-1}[a] \equiv \langle \mathrm{t}[a], a\rangle$

lemma *runit-in-hom*:
assumes *ide a*
shows «$\mathrm{r}[a] : a \otimes \mathbf{1} \rightarrow a$»
⟨*proof*⟩

lemma *runit'-in-hom*:
assumes *ide a*
shows «$\mathrm{r}^{-1}[a] : a \rightarrow a \otimes \mathbf{1}$»
⟨*proof*⟩

lemma *lunit-in-hom*:
assumes *ide a*
shows «$\mathrm{l}[a] : \mathbf{1} \otimes a \rightarrow a$»
⟨*proof*⟩

lemma *lunit'-in-hom*:
assumes *ide a*
shows «$\mathrm{l}^{-1}[a] : a \rightarrow \mathbf{1} \otimes a$»
⟨*proof*⟩

lemma *runit-naturality*:
assumes *arr f*
shows $\mathrm{r}[\mathit{cod}\ f] \cdot (f \otimes \mathbf{1}) = f \cdot \mathrm{r}[\mathit{dom}\ f]$
⟨*proof*⟩

lemma *inverse-arrows-runit*:
assumes *ide a*

shows *inverse-arrows* r[a] r^{-1}[a]
⟨*proof*⟩

lemma *lunit-naturality*:
assumes *arr f*
shows C l[*cod f*] ($\mathbf{1} \otimes f$) = $C\, f$ l[*dom f*]
⟨*proof*⟩

lemma *inverse-arrows-lunit*:
assumes *ide a*
shows *inverse-arrows* l[a] l^{-1}[a]
⟨*proof*⟩

lemma *pr-expansion*:
assumes *ide a* **and** *ide b*
shows $\mathfrak{p}_0[a,\ b] = \mathrm{l}[b] \cdot (\mathrm{t}[a] \otimes b)$ **and** $\mathfrak{p}_1[a,\ b] = \mathrm{r}[a] \cdot (a \otimes \mathrm{t}[b])$
⟨*proof*⟩

lemma *comp-lunit-term-dup*:
assumes *ide a*
shows $\mathrm{l}[a] \cdot (\mathrm{t}[a] \otimes a) \cdot \mathrm{d}[a] = a$
⟨*proof*⟩

lemma *comp-runit-term-dup*:
assumes *ide a*
shows $\mathrm{r}[a] \cdot (a \otimes \mathrm{t}[a]) \cdot \mathrm{d}[a] = a$
⟨*proof*⟩

lemma *dup-coassoc*:
assumes *ide a*
shows $\mathrm{a}[a,\ a,\ a] \cdot (\mathrm{d}[a] \otimes a) \cdot \mathrm{d}[a] = (a \otimes \mathrm{d}[a]) \cdot \mathrm{d}[a]$
⟨*proof*⟩

lemma *comp-assoc-tuple*:
assumes «$f0 : a \rightarrow b0$» **and** «$f1 : a \rightarrow b1$» **and** «$f2 : a \rightarrow b2$»
shows $\mathrm{a}[b0,\ b1,\ b2] \cdot \langle\langle f0,\ f1\rangle,\ f2\rangle = \langle f0,\ \langle f1,\ f2\rangle\rangle$
and $\mathrm{a}^{-1}[b0,\ b1,\ b2] \cdot \langle f0,\ \langle f1,\ f2\rangle\rangle = \langle\langle f0,\ f1\rangle,\ f2\rangle$
⟨*proof*⟩

lemma *dup-tensor*:
assumes *ide a* **and** *ide b*
shows $\mathrm{d}[a \otimes b] = \mathrm{a}^{-1}[a,\ b,\ a \otimes b] \cdot (a \otimes \mathrm{a}[b,\ a,\ b]) \cdot (a \otimes \sigma\ (a,\ b) \otimes b) \cdot$
$(a \otimes \mathrm{a}^{-1}[a,\ b,\ b]) \cdot \mathrm{a}[a,\ a,\ b \otimes b] \cdot (\mathrm{d}[a] \otimes \mathrm{d}[b])$
⟨*proof*⟩

lemma *terminal-tensor-one-one*:
shows *terminal* ($\mathbf{1} \otimes \mathbf{1}$)

⟨*proof*⟩

end

22.3.2 Exponentials

The following prepare the way for the definition of cartesian closed categories. The notion of exponential has to be defined in relation to products. Here we use a generic choice of products for this purpose.

context *cartesian-category*
begin

definition *has-as-exponential*
where *has-as-exponential b c x e* ≡
ide b ∧ *ide x* ∧ «*e* : *some-prod x b* → *c*» ∧
(∀ *a g*. *ide a* ∧ «*g* : *some-prod a b* → *c*» ⟶
(∃!*f*. «*f* : *a* → *x*» ∧ *g* = *C e* (*some-prod f b*)))

lemma *has-as-exponentialI* [*intro*]:
assumes *ide b* **and** *ide x* **and** «*e* : *some-prod x b* → *c*»
and ⋀*a g*. ⟦*ide a*; «*g* : *some-prod a b* → *c*»⟧ ⟹ ∃!*f*. «*f* : *a* → *x*» ∧ *g* = *C e* (*some-prod f b*)
shows *has-as-exponential b c x e*
⟨*proof*⟩

lemma *has-as-exponentialE* [*elim*]:
assumes *has-as-exponential b c x e*
and ⟦*ide b*; *ide x*; «*e* : *some-prod x b* → *c*»;
⋀*a g*. ⟦*ide a*; «*g* : *some-prod a b* → *c*»⟧ ⟹ ∃!*f*. «*f* : *a* → *x*» ∧ *g* = *C e* (*some-prod f b*)⟧
⟹ *T*
shows *T*
⟨*proof*⟩

lemma *exponentials-are-isomorphic*:
assumes *has-as-exponential b c x e* **and** *has-as-exponential b c x′ e′*
shows ∃!*h*. «*h* : *x* → *x′*» ∧ *e* = *e′* · *some-prod h b*
and ⋀*h*. ⟦«*h* : *x* → *x′*»; *e* = *e′* · (*some-prod h b*)⟧ ⟹ *iso h*
⟨*proof*⟩

end

22.4 Category with Finite Products

In this last section, we show that the notion "cartesian category", which we defined to be a category with binary products and terminal object, coincides with the notion "category with finite products". Due to the inability to quantify over types in HOL, we content ourselves with defining the latter notion as "has I-indexed products for every finite set I of natural numbers." We can transfer this property to finite sets at other types using the fact that products are preserved under bijections of the index sets.

locale *category-with-finite-products* =
  category C
for *C* :: *'c comp* +
assumes *has-finite-products*: *finite* (*I* :: *nat set*) ⟹ *has-products I*
begin

  lemma *has-finite-products'*:
  assumes *I* ≠ *UNIV*
  shows *finite I* ⟹ *has-products I*
  ⟨*proof*⟩

end

lemma (**in** *category*) *has-binary-products-if*:
assumes *has-products* ({*0*, *1*} :: *nat set*)
shows *has-binary-products*
⟨*proof*⟩

sublocale *category-with-finite-products* ⊆ *category-with-binary-products C*
  ⟨*proof*⟩

proposition (**in** *category-with-finite-products*) *is-category-with-binary-products*$_{CFP}$:
shows *category-with-binary-products C*
  ⟨*proof*⟩

sublocale *category-with-finite-products* ⊆ *category-with-terminal-object C*
⟨*proof*⟩

proposition (**in** *category-with-finite-products*) *is-category-with-terminal-object*$_{CFP}$:
shows *category-with-terminal-object C*
  ⟨*proof*⟩

sublocale *category-with-finite-products* ⊆ *cartesian-category* ⟨*proof*⟩

proposition (**in** *category-with-finite-products*) *is-cartesian-category*$_{CFP}$:
shows *cartesian-category C*
  ⟨*proof*⟩

context *category*
begin

  lemma *binary-product-of-products-is-product*:
  assumes *has-as-product J0 D0 a0* **and** *has-as-product J1 D1 a1*
  and *has-as-binary-product a0 a1 p0 p1*
  and *Collect* (*partial-composition.arr J0*) ∩ *Collect* (*partial-composition.arr J1*) = {}
  and *partial-magma.null J0* = *partial-magma.null J1*
  shows *has-as-product*
      (*discrete-category.comp*

(*Collect* (*partial-composition.arr J0*) ∪ *Collect* (*partial-composition.arr J1*))
(*partial-magma.null J0*))
(*λi. if i* ∈ *Collect* (*partial-composition.arr J0*) *then D0 i*
else if i ∈ *Collect* (*partial-composition.arr J1*) *then D1 i*
else null)
(*dom p0*)
⟨*proof*⟩

end

sublocale *cartesian-category* ⊆ *category-with-finite-products*
⟨*proof*⟩

proposition (**in** *cartesian-category*) *is-category-with-finite-products*:
shows *category-with-finite-products C*
⟨*proof*⟩

end

Chapter 23

Category with Finite Limits

theory *CategoryWithFiniteLimits*
imports *CartesianCategory CategoryWithPullbacks*
begin

In this chapter we define "category with finite limits" and show that such categories coincide with those having pullbacks and a terminal object.

Since we can't quantify over types in HOL, the best we can do at defining the notion "category with finite limits" is to state it for a fixed choice of type (e.g. *nat*) for the arrows of the "diagram shape". However, we then have to go to some trouble to show the existence of finite limits for diagram shapes at other types.

locale *category-with-finite-limits* =
category +
assumes *has-finite-limits*:
⟦ *category* (*J* :: *nat comp*); *finite* (*Collect* (*partial-composition.arr J*)) ⟧
⟹ *has-limits-of-shape J*
begin

We show that a category with finite limits has pullbacks and a terminal object and is therefore also a cartesian category.

interpretation *category-with-pullbacks C*
⟨*proof*⟩

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

sublocale *category-with-pullbacks C* ⟨*proof*⟩

interpretation *category-with-terminal-object C*
⟨*proof*⟩

lemma *is-category-with-terminal-object*:
shows *category-with-terminal-object C*
⟨*proof*⟩

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

sublocale *category-with-finite-products*
⟨*proof*⟩

sublocale *cartesian-category* ⟨*proof*⟩

end

locale *category-with-pullbacks-and-terminal* =
category-with-pullbacks +
category-with-terminal-object

sublocale *category-with-finite-limits* ⊆ *category-with-pullbacks-and-terminal* ⟨*proof*⟩

Conversely, we show that a category with pullbacks and a terminal object also has finite products and equalizers, and therefore has finite limits.

context *category-with-pullbacks-and-terminal*
begin

interpretation *ECP*: *elementary-category-with-pullbacks C some-prj0 some-prj1*
⟨*proof*⟩

abbreviation *some-prj0'*
where *some-prj0' a b* ≡ (*if ide a* ∧ *ide b then some-prj0* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *else null*)

abbreviation *some-prj1'*
where *some-prj1' a b* ≡ (*if ide a* ∧ *ide b then some-prj1* $\mathrm{t}^?[a]$ $\mathrm{t}^?[b]$ *else null*)

interpretation *ECC*: *elementary-category-with-terminal-object C* ‹$\mathbf{1}^?$› ‹$\lambda a.$ $\mathrm{t}^?[a]$›
⟨*proof*⟩
interpretation *ECC*: *elementary-cartesian-category C some-prj0' some-prj1'* ‹$\mathbf{1}^?$› ‹$\lambda a.$ $\mathrm{t}^?[a]$›
⟨*proof*⟩

interpretation *category-with-equalizers C*
⟨*proof*⟩

interpretation *category-with-finite-products C*
⟨*proof*⟩

lemma *has-finite-products*:
shows *category-with-finite-products C*
⟨*proof*⟩

lemma *has-finite-limits*:
shows *category-with-finite-limits C*
⟨*proof*⟩

sublocale *category-with-finite-limits C*
⟨*proof*⟩

end

end

Chapter 24

Cartesian Closed Category

theory *CartesianClosedCategory*
imports *CartesianCategory*
begin

A *cartesian closed category* is a cartesian category such that, for every object b, the functor *prod* - b is a left adjoint functor. A right adjoint to this functor takes each object c to the *exponential exp* b c. The adjunction yields a natural bijection between *hom* (*prod* a b) c and *hom* a (*exp* b c).

locale *cartesian-closed-category* =
cartesian-category +
assumes *left-adjoint-prod*: $\bigwedge b.$ *ide* $b \Longrightarrow$ *left-adjoint-functor* C C ($\lambda x.$ *some-prod* x b)

locale *elementary-cartesian-closed-category* =
elementary-cartesian-category C *pr0 pr1 one trm*
for C :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ‹·› *55*)
and *pr0* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹$\mathfrak{p}_0$[-, -]›)
and *pr1* :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹$\mathfrak{p}_1$[-, -]›)
and *one* :: $'a$ (‹**1**›)
and *trm* :: $'a \Rightarrow 'a$ (‹t[-]›)
and *exp* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *eval* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *curry* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ +
assumes *eval-in-hom*: ⟦ *ide* b; *ide* c ⟧ $\Longrightarrow$ «*eval* b c : *prod* (*exp* b c) $b \rightarrow c$»
and *ide-exp* [*intro*]: ⟦ *ide* b; *ide* c ⟧ $\Longrightarrow$ *ide* (*exp* b c)
and *curry-in-hom*: ⟦ *ide* a; *ide* b; *ide* c; «g : *prod* a $b \rightarrow c$» ⟧
$\Longrightarrow$ «*curry* a b c g : $a \rightarrow$ *exp* b c»
and *uncurry-curry*: ⟦ *ide* a; *ide* b; *ide* c; «g : *prod* a $b \rightarrow c$» ⟧
$\Longrightarrow$ *eval* b c · *prod* (*curry* a b c g) $b = g$
and *curry-uncurry*: ⟦ *ide* a; *ide* b; *ide* c; «h : $a \rightarrow$ *exp* b c» ⟧
$\Longrightarrow$ *curry* a b c (*eval* b c · *prod* h b) $= h$

context *cartesian-closed-category*
begin

interpretation *elementary-cartesian-category C some-pr0 some-pr1* $\langle \mathbf{1}^? \rangle$ $\langle \lambda a.\ \mathrm{t}^?[a] \rangle$
⟨*proof*⟩

lemma *has-exponentials*:
assumes *ide b* **and** *ide c*
shows $\exists\, x\ e.\ ide\ x \wedge «e : prod\ x\ b \rightarrow c» \wedge$
$(\forall\, a\ g.\ ide\ a \wedge «g : prod\ a\ b \rightarrow c» \longrightarrow (\exists!f.\ «f : a \rightarrow x» \wedge g = e \cdot prod\ f\ b))$
⟨*proof*⟩

definition *some-exp*
where $some\text{-}exp\ b\ c \equiv SOME\ x.\ ide\ x \wedge$
$(\exists\, e.\ «e : prod\ x\ b \rightarrow c» \wedge$
$(\forall\, a\ g.\ ide\ a \wedge «g : prod\ a\ b \rightarrow c»$
$\longrightarrow (\exists!f.\ «f : a \rightarrow x» \wedge g = e \cdot prod\ f\ b)))$

definition *some-eval*
where $some\text{-}eval\ b\ c \equiv SOME\ e.\ «e : prod\ (some\text{-}exp\ b\ c)\ b \rightarrow c» \wedge$
$(\forall\, a\ g.\ ide\ a \wedge «g : prod\ a\ b \rightarrow c»$
$\longrightarrow (\exists!f.\ «f : a \rightarrow some\text{-}exp\ b\ c» \wedge g = e \cdot prod\ f\ b))$

definition *some-curry*
where $some\text{-}curry\ a\ b\ c\ g \equiv THE\, f.\ «f : a \rightarrow some\text{-}exp\ b\ c» \wedge g = some\text{-}eval\ b\ c \cdot prod\ f\ b$

lemma *curry-uniqueness*:
assumes *ide b* **and** *ide c*
shows *ide* (*some-exp b c*)
and $«some\text{-}eval\ b\ c : prod\ (some\text{-}exp\ b\ c)\ b \rightarrow c»$
and $[\![\ ide\ a;\ «g : prod\ a\ b \rightarrow c»\ ]\!] \Longrightarrow$
$\exists!f.\ «f : a \rightarrow some\text{-}exp\ b\ c» \wedge g = some\text{-}eval\ b\ c \cdot prod\ f\ b$
⟨*proof*⟩

lemma *ide-exp* [*intro*, *simp*]:
assumes *ide b* **and** *ide c*
shows *ide* (*some-exp b c*)
⟨*proof*⟩

lemma *eval-in-hom* [*intro*]:
assumes *ide b* **and** *ide c* **and** $x = prod\ (some\text{-}exp\ b\ c)\ b$
shows $«some\text{-}eval\ b\ c : x \rightarrow c»$
⟨*proof*⟩

lemma *uncurry-curry*:
assumes *ide a* **and** *ide b* **and** $«g : prod\ a\ b \rightarrow c»$
shows $«some\text{-}curry\ a\ b\ c\ g : a \rightarrow some\text{-}exp\ b\ c» \wedge$
$g = some\text{-}eval\ b\ c \cdot prod\ (some\text{-}curry\ a\ b\ c\ g)\ b$
⟨*proof*⟩

lemma *curry-uncurry*:
assumes *ide b* **and** *ide c* **and** $«h : a \rightarrow some\text{-}exp\ b\ c»$

shows *some-curry a b c* (*some-eval b c* · *prod h b*) = *h*
⟨*proof*⟩

interpretation *elementary-cartesian-closed-category C some-pr0 some-pr1*
‹$\mathbf{1}^?$› ‹λ*a*. $\mathrm{t}^?[a]$› *some-exp some-eval some-curry*
⟨*proof*⟩

lemma *extends-to-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category C some-pr0 some-pr1*
$\mathbf{1}^?$ (λ*a*. $\mathrm{t}^?[a]$) *some-exp some-eval some-curry*
⟨*proof*⟩

lemma *has-as-exponential*:
assumes *ide b* **and** *ide c*
shows *has-as-exponential b c* (*some-exp b c*) (*some-eval b c*)
⟨*proof*⟩

lemma *has-as-exponential-iff*:
shows *has-as-exponential b c x e* ⟷
ide b ∧ «*e* : *some-prod x b* → *c*» ∧
(∃ *h*. «*h* : *x* → *some-exp b c*» ∧ *e* = *some-eval b c* · *some-prod h b* ∧ *iso h*)
⟨*proof*⟩

end

context *elementary-cartesian-closed-category*
begin

lemma *left-adjoint-prod*:
assumes *ide b*
shows *left-adjoint-functor C C* (λ*x*. *x* ⊗ *b*)
⟨*proof*⟩

sublocale *cartesian-category C*
⟨*proof*⟩

sublocale *cartesian-closed-category C*
⟨*proof*⟩

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

end

end

Chapter 25

The Category of Hereditarily Finite Sets

theory *HF-SetCat*
imports *CategoryWithFiniteLimits CartesianClosedCategory HereditarilyFinite.HF*
begin

This theory constructs a category whose objects are in bijective correspondence with the hereditarily finite sets and whose arrows correspond to the functions between such sets. We show that this category is cartesian closed and has finite limits. Note that up to this point we have not constructed any other interpretation for the *cartesian-closed-category* locale, but it is important to have one to ensure that the locale assumptions are consistent.

25.1 Preliminaries

We begin with some preliminary definitions and facts about hereditarily finite sets, which are better targeted toward what we are trying to do here than what already exists in *HereditarilyFinite.HF*.

The following defines when a hereditarily finite set F represents a function from a hereditarily finite set B to a hereditarily finite set C. Specifically, F must be a relation from B to C, whose domain is B, whose range is contained in C, and which is single-valued on its domain.

definition *hfun*
where $hfun\ B\ C\ F \equiv F \leq B * C \wedge hfunction\ F \wedge hdomain\ F = B \wedge hrange\ F \leq C$

lemma *hfunI* [*intro*]:
assumes $F \leq A * B$
and $\bigwedge X.\ X \in A \Longrightarrow \exists! Y.\ \langle X,\ Y\rangle \in F$
and $\bigwedge X\ Y.\ \langle X,\ Y\rangle \in F \Longrightarrow Y \in B$
shows *hfun A B F*
⟨*proof*⟩

lemma *hfunE* [*elim*]:
assumes *hfun B C F*
and $(\bigwedge Y.\ Y \in B \Longrightarrow (\exists !Z.\ \langle Y, Z\rangle \in F) \wedge (\forall Z.\ \langle Y, Z\rangle \in F \longrightarrow Z \in C)) \Longrightarrow T$
shows *T*
⟨*proof*⟩

The hereditarily finite set *hexp B C* represents the collection of all functions from *B* to *C*.

definition *hexp*
where $hexp\ B\ C = \{\!|F \in HPow\ (B * C).\ hfun\ B\ C\ F|\!\}$

lemma *hfun-in-hexp*:
assumes *hfun B C F*
shows $F \in hexp\ B\ C$
⟨*proof*⟩

The function *happ* applies a function *F* from *B* to *C* to an element of *B*, yielding an element of *C*.

abbreviation *happ*
where $happ \equiv app$

lemma *happ-mapsto*:
assumes $F \in hexp\ B\ C$ **and** $Y \in B$
shows $happ\ F\ Y \in C$ **and** $happ\ F\ Y \in hrange\ F$
⟨*proof*⟩

lemma *happ-expansion*:
assumes *hfun B C F*
shows $F = \{\!|XY \in B * C.\ hsnd\ XY = happ\ F\ (hfst\ XY)|\!\}$
⟨*proof*⟩

Function *hlam* takes a function *F* from $A * B$ to *C* to a function *hlam F* from *A* to *hexp B C*.

definition *hlam*
where $hlam\ A\ B\ C\ F =$
$\{\!|XG \in A * hexp\ B\ C.$
$\forall YZ.\ YZ \in hsnd\ XG \longleftrightarrow is\text{-}hpair\ YZ \wedge \langle\langle hfst\ XG, hfst\ YZ\rangle, hsnd\ YZ\rangle \in F|\!\}$

lemma *hfun-hlam*:
assumes $hfun\ (A * B)\ C\ F$
shows $hfun\ A\ (hexp\ B\ C)\ (hlam\ A\ B\ C\ F)$
⟨*proof*⟩

lemma *happ-hlam*:
assumes $X \in A$ **and** $hfun\ (A * B)\ C\ F$
shows $\exists !G.\ \langle X, G\rangle \in hlam\ A\ B\ C\ F$
and $happ\ (hlam\ A\ B\ C\ F)\ X = (THE\ G.\ \langle X, G\rangle \in hlam\ A\ B\ C\ F)$
and $happ\ (hlam\ A\ B\ C\ F)\ X = \{\!|yz \in B * C.\ \langle\langle X, hfst\ yz\rangle, hsnd\ yz\rangle \in F|\!\}$

and $Y \in B \Longrightarrow$ *happ* (*happ* (*hlam* A B C F) X) Y = *happ* F $\langle X,\ Y\rangle$
⟨*proof*⟩

25.2 Construction of the Category

locale *hfsetcat*
begin

We construct the category of hereditarily finite sets and functions simply by applying the generic "set category" construction, using the hereditarily finite sets as the universe, and constraining the collections of such sets that determine objects of the category to those that are finite.

interpretation *setcat* ‹*TYPE*(*hf*)› *finite*
⟨*proof*⟩
interpretation *set-category comp* ‹$\lambda A.$ $A \subseteq$ *Collect terminal* $\wedge$ *finite* (*elem-of* ‘ A)›
⟨*proof*⟩

lemma *set-ide-char*:
shows $A \in$ *set* ‘ *Collect ide* $\longleftrightarrow$ $A \subseteq$ *Univ* $\wedge$ *finite* A
⟨*proof*⟩

lemma *set-ideD*:
assumes *ide a*
shows *set a* $\subseteq$ *Univ* **and** *finite* (*set a*)
⟨*proof*⟩

lemma *ide-mkIdeI* [*intro*]:
assumes $A \subseteq$ *Univ* **and** *finite* A
shows *ide* (*mkIde* A) **and** *set* (*mkIde* A) = A
⟨*proof*⟩

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

We verify that the objects of HF are indeed in bijective correspondence with the hereditarily finite sets.

definition *ide-to-hf*
where *ide-to-hf a* = *HF* (*elem-of* ‘ *set a*)

definition *hf-to-ide*
where *hf-to-ide x* = *mkIde* (*arr-of* ‘ *hfset x*)

lemma *ide-to-hf-mapsto*:
shows *ide-to-hf* $\in$ *Collect ide* $\rightarrow$ *UNIV*
⟨*proof*⟩

lemma *hf-to-ide-mapsto*:
shows *hf-to-ide* $\in$ *UNIV* $\rightarrow$ *Collect ide*

⟨*proof*⟩

lemma *hf-to-ide-ide-to-hf*:
assumes $a \in$ *Collect ide*
shows *hf-to-ide* (*ide-to-hf a*) = *a*
⟨*proof*⟩

lemma *ide-to-hf-hf-to-ide*:
assumes $x \in$ *UNIV*
shows *ide-to-hf* (*hf-to-ide x*) = *x*
⟨*proof*⟩

lemma *bij-betw-ide-hf-set*:
shows *bij-betw ide-to-hf* (*Collect ide*) (*UNIV* :: *hf set*)
⟨*proof*⟩

lemma *ide-implies-finite-set*:
assumes *ide a*
shows *finite* (*set a*) **and** *finite* (*hom unity a*)
⟨*proof*⟩

We establish the connection between the membership relation defined for hereditarily finite sets and the corresponding membership relation associated with the set category.

lemma *arr-of-membI* [*intro*]:
assumes *x* ∈ *ide-to-hf a*
shows *arr-of x* ∈ *set a*
⟨*proof*⟩

lemma *elem-of-membI* [*intro*]:
assumes *ide a* **and** *x* ∈ *set a*
shows *elem-of x* ∈ *ide-to-hf a*
⟨*proof*⟩

We show that each hom-set *hom a b* is in bijective correspondence with the elements of the hereditarily finite set *hfun* (*ide-to-hf a*) (*ide-to-hf b*).

definition *arr-to-hfun*
where *arr-to-hfun f* = ⦃*XY* ∈ *ide-to-hf* (*dom f*) ∗ *ide-to-hf* (*cod f*).
hsnd XY = *elem-of* (*Fun f* (*arr-of* (*hfst XY*)))⦄

definition *hfun-to-arr*
where *hfun-to-arr B C F* =
mkArr (*arr-of* ‘ *hfset B*) (*arr-of* ‘ *hfset C*) (λ*x*. *arr-of* (*happ F* (*elem-of x*)))

lemma *hfun-arr-to-hfun*:
assumes *arr f*
shows *hfun* (*ide-to-hf* (*dom f*)) (*ide-to-hf* (*cod f*)) (*arr-to-hfun f*)
⟨*proof*⟩

lemma *arr-to-hfun-in-hexp*:

assumes *arr f*
shows *arr-to-hfun f* ∈ *hexp* (*ide-to-hf* (*dom f*)) (*ide-to-hf* (*cod f*))
⟨*proof*⟩

lemma *hfun-to-arr-in-hom*:
assumes *hfun B C F*
shows «*hfun-to-arr B C F* : *hf-to-ide B* → *hf-to-ide C*»
⟨*proof*⟩

The comprehension notation from *HereditarilyFinite.HF* interferes in an unfortunate way with the restriction notation from *HOL−Library.FuncSet*, making it impossible to use both in the present context.

lemma *Fun-char*:
assumes *arr f*
shows *Fun f* = *restrict* (*λx. arr-of* (*happ* (*arr-to-hfun f*) (*elem-of x*))) (*Dom f*)
⟨*proof*⟩

lemma *Fun-hfun-to-arr*:
assumes *hfun B C F*
shows *Fun* (*hfun-to-arr B C F*) = *restrict* (*λx. arr-of* (*happ F* (*elem-of x*))) (*arr-of* ‘ *hfset B*)
⟨*proof*⟩

lemma *arr-of-img-hfset-ide-to-hf*:
assumes *ide a*
shows *arr-of* ‘ *hfset* (*ide-to-hf a*) = *set a*
⟨*proof*⟩

lemma *hfun-to-arr-arr-to-hfun*:
assumes *arr f*
shows *hfun-to-arr* (*ide-to-hf* (*dom f*)) (*ide-to-hf* (*cod f*)) (*arr-to-hfun f*) = *f*
⟨*proof*⟩

lemma *arr-to-hfun-hfun-to-arr*:
assumes *hfun B C F*
shows *arr-to-hfun* (*hfun-to-arr B C F*) = *F*
⟨*proof*⟩

lemma *bij-betw-hom-hfun*:
assumes *ide a* **and** *ide b*
shows *bij-betw arr-to-hfun* (*hom a b*) {*F. hfun* (*ide-to-hf a*) (*ide-to-hf b*) *F*}
⟨*proof*⟩

We next relate composition of arrows in the category to the corresponding operation on hereditarily finite sets.

definition *hcomp*
where *hcomp G F* =
⦃*XZ* ∈ *hdomain F* ∗ *hrange G. hsnd XZ* = *happ G* (*happ F* (*hfst XZ*))⦄

lemma *hfun-hcomp*:
assumes *hfun A B F* **and** *hfun B C G*
shows *hfun A C (hcomp G F)*
⟨*proof*⟩

lemma *arr-to-hfun-comp*:
assumes *seq g f*
shows *arr-to-hfun (comp g f) = hcomp (arr-to-hfun g) (arr-to-hfun f)*
⟨*proof*⟩

lemma *hfun-to-arr-hcomp*:
assumes *hfun A B F* **and** *hfun B C G*
shows *hfun-to-arr A C (hcomp G F) = comp (hfun-to-arr B C G) (hfun-to-arr A B F)*
⟨*proof*⟩

25.3 Binary Products

The category of hereditarily finite sets has binary products, given by cartesian product of sets in the usual way.

definition *prod*
where *prod a b = hf-to-ide (ide-to-hf a * ide-to-hf b)*

definition *pr0*
where *pr0 a b = (if ide a ∧ ide b then*
mkArr (set (prod a b)) (set b) (λx. arr-of (hsnd (elem-of x)))
else null)

definition *pr1*
where *pr1 a b = (if ide a ∧ ide b then*
mkArr (set (prod a b)) (set a) (λx. arr-of (hfst (elem-of x)))
else null)

definition *tuple*
where *tuple f g = mkArr (set (dom f)) (set (prod (cod f) (cod g)))*
(λx. arr-of (hpair (elem-of (Fun f x)) (elem-of (Fun g x))))

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

lemma *pr1-in-hom [intro]*:
assumes *ide a* **and** *ide b*
shows «*pr1 a b : prod a b → a*»
⟨*proof*⟩

lemma *pr1-simps [simp]*:
assumes *ide a* **and** *ide b*

shows *arr* (*pr1 a b*) **and** *dom* (*pr1 a b*) = *prod a b* **and** *cod* (*pr1 a b*) = *a*
⟨*proof*⟩

lemma *pr0-in-hom* [*intro*]:
assumes *ide a* **and** *ide b*
shows «*pr0 a b* : *prod a b* → *b*»
⟨*proof*⟩

lemma *pr0-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* (*pr0 a b*) **and** *dom* (*pr0 a b*) = *prod a b* **and** *cod* (*pr0 a b*) = *b*
⟨*proof*⟩

lemma *arr-of-tuple-elem-of-membI*:
assumes *span f g* **and** *x* ∈ *Dom f*
shows *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩ ∈ *set* (*prod* (*cod f*) (*cod g*))
⟨*proof*⟩

lemma *tuple-in-hom* [*intro*]:
assumes *span f g*
shows «*tuple f g* : *dom f* → *prod* (*cod f*) (*cod g*)»
⟨*proof*⟩

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

lemma *Fun-pr1*:
assumes *ide a* **and** *ide b*
shows *Fun* (*pr1 a b*) = *restrict* (λ*x*. *arr-of* (*hfst* (*elem-of x*))) (*set* (*prod a b*))
⟨*proof*⟩

lemma *Fun-pr0*:
assumes *ide a* **and** *ide b*
shows *Fun* (*pr0 a b*) = *restrict* (λ*x*. *arr-of* (*hsnd* (*elem-of x*))) (*set* (*prod a b*))
⟨*proof*⟩

lemma *Fun-tuple*:
assumes *span f g*
shows *Fun* (*tuple f g*) = *restrict* (λ*x*. *arr-of* ⟨*elem-of* (*Fun f x*), *elem-of* (*Fun g x*)⟩) (*Dom f*)
⟨*proof*⟩

lemma *pr1-tuple*:
assumes *span f g*
shows *comp* (*pr1* (*cod f*) (*cod g*)) (*tuple f g*) = *f*
⟨*proof*⟩

lemma *pr0-tuple*:
assumes *span f g*
shows *comp* (*pr0* (*cod f*) (*cod g*)) (*tuple f g*) = *g*
⟨*proof*⟩

lemma *tuple-pr*:
assumes *ide a* **and** *ide b* **and** «*h* : *dom h* → *prod a b*»
shows *tuple* (*comp* (*pr1 a b*) *h*) (*comp* (*pr0 a b*) *h*) = *h*
⟨*proof*⟩

interpretation *HF'*: *elementary-category-with-binary-products comp pr0 pr1*
⟨*proof*⟩

For reasons of economy of locale parameters, the notion *prod* is a defined notion of the *elementary-category-with-binary-products* locale. However, we need to be able to relate this notion to that of cartesian product of hereditarily finite sets, which we have already used to give a definition of *prod*. The locale assumptions for *elementary-cartesian-closed-category* refer specifically to *HF'.prod*, even though in the end the notion itself does not depend on that choice. To be able to show that the locale assumptions of *elementary-cartesian-closed-category* are satisfied, we need to use a choice of products that we can relate to the cartesian product of hereditarily finite sets. We therefore need to show that our previously defined *prod* coincides (on objects) with the one defined in the *elementary-category-with-binary-products* locale; *i.e. HF'.prod.* Note that the latter is defined for all arrows, not just identity arrows, so we need to use that for the subsequent definitions and proofs.

lemma *prod-ide-eq*:
assumes *ide a* **and** *ide b*
shows *prod a b* = *HF'.prod a b*
⟨*proof*⟩

lemma *tuple-span-eq*:
assumes *span f g*
shows *tuple f g* = *HF'.tuple f g*
⟨*proof*⟩

25.4 Exponentials

We now turn our attention to exponentials.

definition *exp*
where *exp b c* = *hf-to-ide* (*hexp* (*ide-to-hf b*) (*ide-to-hf c*))

definition *eval*
where *eval b c* = *mkArr* (*set* (*HF'.prod* (*exp b c*) *b*)) (*set c*)
(*λx. arr-of* (*happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*))))

definition Λ

where Λ *a b c f* = *mkArr* (*set a*) (*set* (*exp b c*))
(λ*x*. *arr-of* (*happ* (*hlam* (*ide-to-hf a*) (*ide-to-hf b*) (*ide-to-hf c*)
(*arr-to-hfun f*))
(*elem-of x*)))

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

lemma *hfset-ide-to-hf*:
assumes *ide a*
shows *hfset* (*ide-to-hf a*) = *elem-of* ' *set a*
⟨*proof*⟩

lemma *eval-in-hom* [*intro*]:
assumes *ide b* **and** *ide c*
shows *in-hom* (*eval b c*) (*HF'.prod* (*exp b c*) *b*) *c*
⟨*proof*⟩

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

lemma *hlam-arr-to-hfun-in-hexp*:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom f* (*prod a b*) *c*
shows *hlam* (*ide-to-hf a*) (*ide-to-hf b*) (*ide-to-hf c*) (*arr-to-hfun f*)
∈ *hexp* (*ide-to-hf a*) (*ide-to-hf* (*exp b c*))
⟨*proof*⟩

lemma *lam-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom f* (*prod a b*) *c*
shows *in-hom* (Λ *a b c f*) *a* (*exp b c*)
⟨*proof*⟩

lemma *lam-simps* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom f* (*prod a b*) *c*
shows *arr* (Λ *a b c f*)
and *dom* (Λ *a b c f*) = *a*
and *cod* (Λ *a b c f*) = *exp b c*
⟨*proof*⟩

lemma *Fun-lam*:

assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom f* (*prod a b*) *c*
shows *Fun* (Λ *a b c f*) =
restrict (λ*x*. *arr-of* (*happ* (*hlam* (*ide-to-hf a*) (*ide-to-hf b*) (*ide-to-hf c*) (*arr-to-hfun f*))
(*elem-of x*)))
(*set a*)
⟨*proof*⟩

lemma *Fun-eval*:
assumes *ide b* **and** *ide c*
shows *Fun* (*eval b c*) = *restrict* (λ*x*. *arr-of* (*happ* (*hfst* (*elem-of x*)) (*hsnd* (*elem-of x*))))
(*set* (*HF′.prod* (*exp b c*) *b*))
⟨*proof*⟩

lemma *Fun-prod*:
assumes *arr f* **and** *arr g* **and** *x* ∈ *set* (*prod* (*dom f*) (*dom g*))
shows *Fun* (*HF′.prod f g*) *x* = *arr-of* ⟨*elem-of* (*Fun f* (*arr-of* (*hfst* (*elem-of x*)))),
elem-of (*Fun g* (*arr-of* (*hsnd* (*elem-of x*))))⟩
⟨*proof*⟩

lemma *prod-in-terms-of-tuple*:
assumes *arr f* **and** *arr g*
shows *HF′.prod f g* =
tuple (*comp f* (*pr1* (*dom f*) (*dom g*))) (*comp g* (*pr0* (*dom f*) (*dom g*)))
⟨*proof*⟩

lemma *eval-prod-lam*:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom g* (*prod a b*) *c*
shows *comp* (*eval b c*) (*HF′.prod* (Λ *a b c g*) *b*) = *g*
⟨*proof*⟩

lemma *lam-eval-prod*:
assumes *ide a* **and** *ide b* **and** *ide c*
and *in-hom h a* (*exp b c*)
shows Λ *a b c* (*comp* (*eval b c*) (*HF′.prod h b*)) = *h*
⟨*proof*⟩

25.5 The Main Results

interpretation *cartesian-closed-category comp*
⟨*proof*⟩

theorem *is-cartesian-closed-category*:
shows *cartesian-closed-category comp*
⟨*proof*⟩

theorem *is-category-with-finite-limits*:
shows *category-with-finite-limits comp*

⟨*proof*⟩

end

end
theory *HF-SetCat-Interp*
imports *HF-SetCat*
begin

Here we demonstrate the possibility of making a top-level interpretation of the *ZFC-set-cat* locale. See theory *SetCat-Interp* for further discussion on why we do this.

interpretation *HF-Sets*: *hfsetcat* ⟨*proof*⟩

end

Chapter 26

ZFC SetCat

In the statement and proof of the Yoneda Lemma given in theory *Yoneda*, we sidestepped the issue, of not having a category of "all" sets, by axiomatizing the notion of a "set category", showing that for every category we could obtain a hom-functor into a set category at a higher type, and then proving the Yoneda lemma for that particular hom-functor. This is perhaps the best we can do within HOL, because HOL does not provide any type that contains a universe of sets with the closure properties usually associated with a category *Set* of sets and functions between them. However, a significant aspect of category theory involves considering "all" algebraic structures of a particular kind as the objects of a "large" category having nice closure or completeness properties. Being able to consider a category of sets that is "small-complete", or a cartesian closed category of sets and functions that includes some infinite sets as objects, are basic examples of this kind of situation.

The purpose of this section is to demonstrate that, although it cannot be done in pure HOL, if we are willing to accept the existence of a type V whose inhabitants correspond to sets satisfying the axioms of ZFC, then it is possible to construct, for example, the "large" category of sets and functions as it is usually understood in category theory. Moreover, assuming the existence of such a type is essentially all we have to do; all the category theory we have developed so far still applies. Specifically, what we do in this section is to use theory *ZFC-in-HOL*, which provides an axiomatization of a set-theoretic universe V, to construct a "set category" *ZFC-SetCat*, whose objects correspond to V-sets, whose arrows correspond to functions between V-sets, and which has the small-completeness property traditionally ascribed to the category of all small sets and functions between them.

theory *ZFC-SetCat*
imports *ZFC-in-HOL.ZFC-Cardinals Limit*
begin

The following locale constructs the category of classes and functions between them and shows that it is small complete. The category is obtained simply as the replete set category at type V. This is not yet the category of sets we want, because it contains objects corresponding to "large" V-sets.

locale *ZFC-class-cat*
begin

sublocale *replete-setcat* ‹*TYPE*(*V*)› ⟨*proof*⟩

lemma *admits-small-V-tupling*:
assumes *small* (*I* :: *V set*)
shows *admits-tupling I*
⟨*proof*⟩

corollary *admits-small-tupling*:
assumes *small I*
shows *admits-tupling I*
⟨*proof*⟩

lemma *has-small-products*:
assumes *small* (*I* :: *'i set*) **and** *I* ≠ *UNIV*
shows *has-products I*
⟨*proof*⟩

theorem *has-small-limits*:
assumes *small* (*UNIV* :: *'i set*)
shows *has-limits* (*undefined* :: *'i*)
⟨*proof*⟩

end

We now construct the desired category of small sets and functions between them, as a full subcategory of the category of classes and functions. To show that this subcategory is small complete, we show that the inclusion creates small products; that is, a small product of objects corresponding to small sets itself corresponds to a small set.

locale *ZFC-set-cat*
begin

interpretation *Cls*: *ZFC-class-cat* ⟨*proof*⟩

definition *setp*
where *setp A* ≡ *A* ⊆ *Cls.Univ* ∧ *small A*

sublocale *sub-set-category Cls.comp* ‹λ*A*. *A* ⊆ *Cls.Univ*› *setp*
⟨*proof*⟩

lemma *is-sub-set-category*:
shows *sub-set-category Cls.comp* (λ*A*. *A* ⊆ *Cls.Univ*) *setp*
⟨*proof*⟩

interpretation *incl*: *full-inclusion-functor Cls.comp* ‹λ*a*. *Cls.ide a* ∧ *setp* (*Cls.set a*)›
⟨*proof*⟩

The following functions establish a bijection between the identities of the category and the elements of type V; which in turn are in bijective correspondence with small V-sets.

definition *V-of-ide* :: *V setcat.arr* ⇒ *V*
where *V-of-ide a* ≡ *ZFC-in-HOL.set* (*Cls.DN* ‘ *Cls.set a*)

definition *ide-of-V* :: *V* ⇒ *V setcat.arr*
where *ide-of-V A* ≡ *Cls.mkIde* (*Cls.UP* ‘ *elts A*)

lemma *bij-betw-ide-V*:
shows *V-of-ide* ∈ *Collect ide* → *UNIV*
and *ide-of-V* ∈ *UNIV* → *Collect ide*
and [*simp*]: *ide a* ⟹ *ide-of-V* (*V-of-ide a*) = *a*
and [*simp*]: *V-of-ide* (*ide-of-V A*) = *A*
and *bij-betw V-of-ide* (*Collect ide*) *UNIV*
and *bij-betw ide-of-V UNIV* (*Collect ide*)
⟨*proof*⟩

Next, we establish bijections between the hom-sets of the category and certain subsets of V whose elements represent functions.

definition *V-of-arr* :: *V setcat.arr* ⇒ *V*
where *V-of-arr f* ≡ *VLambda* (*V-of-ide* (*dom f*)) (*Cls.DN o Cls.Fun f o Cls.UP*)

definition *arr-of-V* :: *V setcat.arr* ⇒ *V setcat.arr* ⇒ *V* ⇒ *V setcat.arr*
where *arr-of-V a b F* ≡ *Cls.mkArr* (*Cls.set a*) (*Cls.set b*) (*Cls.UP o app F o Cls.DN*)

definition *vfun*
where *vfun A B f* ≡ *f* ∈ *elts* (*VPow* (*vtimes A B*)) ∧ *elts A* = *Domain* (*pairs f*) ∧
single-valued (*pairs f*)

lemma *small-Collect-vfun*:
shows *small* (*Collect* (*vfun A B*))
⟨*proof*⟩

lemma *vfunI*:
assumes *f* ∈ *elts A* → *elts B*
shows *vfun A B* (*VLambda A f*)
⟨*proof*⟩

lemma *app-vfun-mapsto*:
assumes *vfun A B F*
shows *app F* ∈ *elts A* → *elts B*
⟨*proof*⟩

lemma *bij-betw-hom-vfun*:
shows *V-of-arr* ∈ *hom a b* → *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))
and ⟦*ide a*; *ide b*⟧ ⟹ *arr-of-V a b* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)) → *hom a b*
and *f* ∈ *hom a b* ⟹ *arr-of-V a b* (*V-of-arr f*) = *f*
and ⟦*ide a*; *ide b*; *F* ∈ *Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))⟧
⟹ *V-of-arr* (*arr-of-V a b F*) = *F*

and ⟦*ide a*; *ide b*⟧
⟹ *bij-betw V-of-arr* (*hom a b*) (*Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*)))
and ⟦*ide a*; *ide b*⟧
⟹ *bij-betw* (*arr-of-V a b*) (*Collect* (*vfun* (*V-of-ide a*) (*V-of-ide b*))) (*hom a b*)
⟨*proof*⟩

lemma *small-hom*:
shows *small* (*hom a b*)
⟨*proof*⟩

We can now show that the inclusion of the subcategory into the ambient category *Cls* creates small products. To do this, we consider a product in *Cls* of objects of the subcategory indexed by a small set *I*. Since *Cls* is a replete set category, by a previous result we know that the elements of a product object *p* in *Cls* correspond to its points; that is, to the elements of *hom unity p*. The elements of *hom unity p* in turn correspond to *I*-tuples. By carrying out the construction of the set of *I*-tuples in *V* and exploiting the bijections between homs of the subcatgory and *V*-sets, we can obtain an injection of *hom unity p* to the extension of a *V*-set, thus showing *hom unity p* is small. Since *hom unity p* is small, it determines an object of the subcategory, which must then be a product in the subcategory, in view of the fact that the subcategory is full.

lemma *has-small-V-products*:
assumes *small* (*I* :: *V set*)
shows *has-products I*
⟨*proof*⟩

corollary *has-small-products*:
assumes *small I* **and** *I* ≠ *UNIV*
shows *has-products I*
⟨*proof*⟩

theorem *has-small-limits*:
assumes *category* (*J* :: *'j comp*) **and** *small* (*Collect* (*partial-composition.arr J*))
shows *has-limits-of-shape J*
⟨*proof*⟩

sublocale *concrete-set-category comp setp UNIV Cls.UP*
⟨*proof*⟩

lemma *is-concrete-set-category*:
shows *concrete-set-category comp setp UNIV Cls.UP*
⟨*proof*⟩

end

In pure HOL (without ZFC), we were able to show that every category *C* has a "hom functor", but there was necessarily a dependence of the target set category of the hom functor on the arrow type of *C*. Using the construction of the present theory, we can now show that every "locally small" category *C* has a hom functor, whose target is the same set category for all such *C*. To obtain such a hom functor requires a choice, for each hom-set *hom a b* of *C*, of an injection of *hom a b* to the extension of a *V*-set.

locale *locally-small-category* =
category +
assumes *locally-small*: ⟦*ide a*; *ide b*⟧ ⟹ *small* (*hom b a*)
begin

interpretation *Cop*: *dual-category C* ⟨*proof*⟩
interpretation *CopxC*: *product-category Cop.comp C* ⟨*proof*⟩
interpretation *S*: *ZFC-set-cat* ⟨*proof*⟩

definition *Hom*
where *Hom* ≡ λ(*b*, *a*). *S.UP o* (*SOME* φ. φ ‘ *hom b a* ∈ *range elts* ∧ *inj-on* φ (*hom b a*))

interpretation *Hom*: *hom-functor C S.comp S.setp Hom*
⟨*proof*⟩

lemma *has-ZFC-hom-functor*:
shows *hom-functor C S.comp S.setp Hom*
⟨*proof*⟩

Using this result, we can now state a more traditional version of the Yoneda Lemma in which the target category of the Yoneda functor is the same for all locally small categories.

interpretation *Y*: *yoneda-functor C S.comp S.setp Hom*
⟨*proof*⟩

theorem *ZFC-yoneda-lemma*:
assumes *ide a* **and** *functor Cop.comp S.comp F*
shows ∃φ. *bij-betw* φ (*S.set* (*F a*)) {τ. *natural-transformation Cop.comp S.comp* (*Y.Y a*) *F* τ}
⟨*proof*⟩

end

end
theory *ZFC-SetCat-Interp*
imports *ZFC-SetCat*
begin

Here we demonstrate the possibility of making a top-level interpretation of the *ZFC-set-cat* locale

interpretation *ZFCClsCat*: *ZFC-class-cat* ⟨*proof*⟩
interpretation *ZFCSetCat*: *ZFC-set-cat* ⟨*proof*⟩

To clarify that the category *ZFCSetCat* is what it is supposed to be, we offer the following summary results.

The set of terminal objects of *ZFCSetCat* is in bijective correspondence with the elements of type *V*.

lemma *bij-betw-terminals-and-V*:
shows *bij-betw ZFCSetCat.DN ZFCSetCat.Univ* (*UNIV* :: *V set*)
⟨*proof*⟩

The set of elements of any object of ZFCSetCat is a small subset of the set of terminal objects.

lemma *ide-implies-small-set*:
assumes *ZFCSetCat.ide a*
shows *small* (*ZFCSetCat.set a*) **and** *ZFCSetCat.set a* ⊆ *ZFCSetCat.Univ*
⟨*proof*⟩

Every small set (at an arbitrary type) is in bijective correspondence with the set of elements of some object of *ZFCSetCat*.

lemma *small-implies-bij-to-set*:
assumes *small A*
shows ∃ *a* φ. *ZFCSetCat.ide a* ∧ *bij-betw* φ *A* (*ZFCSetCat.set a*)
⟨*proof*⟩

For objects a and b of ZFCSetCat, the arrows from a to b are in bijective correspondence with the extensional functions between the underlying sets of terminal objects.

lemma *bij-betw-hom-and-ext-funcset*:
assumes *ZFCSetCat.ide a* **and** *ZFCSetCat.ide b*
shows *bij-betw ZFCSetCat.Fun* (*ZFCSetCat.hom a b*) (*ZFCSetCat.set a* $\rightarrow_E$ *ZFCSetCat.set b*)
⟨*proof*⟩

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].