Monoidal Categories

Eugene W. Stark

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

September 26, 2023

Abstract

Building on the formalization of basic category theory set out in the author's previous AFP article [6], the present article formalizes some basic aspects of the theory of monoidal categories. Among the notions defined here are monoidal category, monoidal functor, and equivalence of monoidal categories. The main theorems formalized are MacLane's coherence theorem and the constructions of the free monoidal category and free strict monoidal category generated by a given category. The coherence theorem is proved syntactically, using a structurally recursive approach to reduction of terms that might have some novel aspects. We also give proofs of some results given by Etingof *et al* [2], which may prove useful in a formal setting. In particular, we show that the left and right unitors need not be taken as given data in the definition of monoidal category, nor does the definition of monoidal functor need to take as given a specific isomorphism expressing the preservation of the unit object. Our definitions of monoidal category and monoidal functor are stated so as to take advantage of the economy afforded by these facts.

Revisions made subsequent to the first version of this article added material on cartesian monoidal categories; showing that the underlying category of a cartesian monoidal category is a cartesian category, and that every cartesian category extends to a cartesian monoidal category.

Contents

Chapter 1

Introduction

A *monoidal category* is a category C equipped with a binary "tensor product" functor $\otimes : C \times C \to C$, which is associative up to a given natural isomorphism, and an object $\mathcal{I}$ that behaves up to isomorphism like a unit for $\otimes$. The associativity and unit isomorphisms are assumed to satisfy certain axioms known as *coherence conditions*. Monoidal categories were introduced by Bénabou [1] and MacLane [4]. MacLane showed that the axioms for a monoidal category imply that all diagrams in a large class are commutative. This result, known as MacLane's Coherence Theorem, is the first important result in the theory of monoidal categories.

Monoidal categories are important partly because of their ubiquity. The category of sets and functions is monoidal; more generally any category with binary products and a terminal object becomes a monoidal category if we take the categorical product as $\otimes$ and the terminal object as $\mathcal{I}$. The category of vector spaces over a field, with linear maps as morphisms, not only admits monoidal structure with respect to the categorical product, but also with respect to the usual tensor product of vector spaces. Monoidal categories serve as the starting point for enriched category theory in that they provide a setting in which ordinary categories, having "homs in the category of sets," can be generalized to "categories having homs in a monoidal category $\mathcal{V}$". In addition, the theory of monoidal categories can be regarded as a stepping stone to the theory of bicategories, as monoidal categories are the same thing as one-object bicategories.

Building on the formalization of basic category theory set out in the author's previous AFP article [6], the present article formalizes some basic aspects of the theory of monoidal categories. In Chapter 2, we give a definition of the notion of monoidal category and develop consequences of the axioms. We then give a proof of MacLane's coherence theorem. The proof is syntactic: we define a language of terms built from arrows of a given category C using constructors that correspond to formal composition and tensor product as well as to the associativity and unit isomorphisms and their formal inverses, we then define a mapping that interprets terms of the language in an arbitrary monoidal category D via a valuation functor $V : C \to D$, and finally we syntactically characterize a class of equations between terms that hold in any such interpretation. Among these equations are all those that relate formally parallel "canonical" terms, where a term is

canonical if the only arrows of C that are used in its construction are identities. Thus, all formally parallel canonical terms have identical interpretations in any monoidal category, which is the content of MacLane's coherence theorem.

In Chapter 3, we define the notion of a *monoidal functor* between monoidal categories. A monoidal functor from a monoidal category C to a monoidal category D is a functor $F : C \to D$, equipped with additional data that express that the monoidal structure is preserved by F up to natural isomorphism. A monoidal functor is *strict* if it preserves the monoidal structure "on the nose" (*i.e.* the natural isomorphism is an identity). We also define the notion of an *equivalence of monoidal categories*, which is a monoidal functor $F : C \to D$ that is part of an ordinary equivalence of categories between C and D.

In Chapter 4, we use the language of terms defined in Chapter 2 to give a syntactic construction of the free monoidal category $\mathcal{F}C$ generated by a category C. The arrows $\mathcal{F}C$ are defined to be certain equivalence classes of terms, where composition and tensor product, as well as the associativity and unit isomorphisms, are determined by the syntactic operations. After proving that the construction does in fact yield a monoidal category, we establish its freeness: every functor from C to a monoidal category D extends uniquely to a strict monoidal functor from $\mathcal{F}C$ to D. We then consider the subcategory $\mathcal{F}_S C$ of $\mathcal{F}C$ whose arrows are equivalence classes of terms that we call "diagonal." Diagonal terms amount to lists of arrows of C, composition in $\mathcal{F}_S C$ is given by elementwise composition of compatible lists of arrows, and tensor product in $\mathcal{F}_S C$ is given by concatenation of lists. We show that the subcategory $\mathcal{F}_S C$ is monoidally equivalent to the category $\mathcal{F}C$ and in addition that $\mathcal{F}_S C$ is the free strict monoidal category generated by $\mathcal{C}$.

The formalizations of the notions of monoidal category and monoidal functor that we give here are not quite the traditional ones. The traditional definition of monoidal category assumes as given not only an "associator" natural isomorphism, which expresses the associativity of the tensor product, but also left and right "unitor" isomorphisms, which correspond to unit laws. However, as pointed out in [2], it is not necessary to take the unitors as given, because they are uniquely determined by the other structure and the condition that left and right tensoring with the unit object are endo-equivalences. This leads to a definition of monoidal category that requires fewer data to be given and fewer conditions to be verified in applications. As this is likely to be especially important in a formal setting, we adopt this more economical definition and go to the trouble to obtain the unitors as defined notions. A similar situation occurs with the definition of monoidal functor. The traditional definition requires two natural isomorphisms to be given: one that expresses the preservation of tensor product and another that expresses the preservation of the unit object. Once again, as indicated in [2], it is logically unnecessary to take the latter isomorphism as given, since there is a canonical definition of it in terms of the other structure. We adopt the more economical definition of monoidal functor and prove that the traditionally assumed structure can be derived from it.

Finally, the proof of the coherence theorem given here potentially has some novel aspects. A typical syntactic proof of this theorem, such as that described in [5], involves the identification, for each term constructed as a formal tensor product of the unit object $\mathcal{I}$ and "primitive objects" (*i.e.* the elements of a given set of generators), of a "reduction"

isomorphism obtained by composing "basic reductions" in which occurrences of $\mathcal{I}$ are eliminated using components of the left and right unitors and "parentheses are moved to one end" using components of the associator. The construction of these reductions is performed, as in [5], using an approach that can be thought of as the application of an iterative strategy for normalizing a term. My thoughts were initially along these lines, and I did succeed in producing a formal proof of the coherence theorem in this way. However, proving the termination of the reduction strategy was complicated by the necessity of using of a "rank function" on terms, and the lemmas required for the remainder of the proof had to be proved by induction on rank, which was messy. At some point, I realized that it ought to be possible to define reductions in a structurally recursive way, which would permit the lemmas in the rest of the proof to be proved by structural induction, rather than induction on rank. It took some time to find the right definitions, but in the end this approach worked out more simply, and is what is presented here.

Revision Notes

The original version of this document dates from May, 2017. The current version of this document incorporates revisions made in mid-2020 after the release of Isabelle2020. Aside from various minor improvements, the main change was the addition of a new theory, concerning cartesian monoidal categories, which coordinates with material on cartesian categories that was simultaneously added to [6]. The new theory defines "cartesian monoidal category" as an extension of "monoidal category" obtained by adding additional functors, natural transformations, and coherence conditions. The main results proved are that the underlying category of a cartesian monoidal category is a cartesian category, and that every cartesian category extends to a cartesian monoidal category.

Chapter 2

Monoidal Category

In this theory, we define the notion "monoidal category," and develop consequences of the definition. The main result is a proof of MacLane's coherence theorem.

theory *MonoidalCategory*
imports *Category3.EquivalenceOfCategories*
begin

2.1 Monoidal Category

A typical textbook presentation defines a monoidal category to be a category C equipped with (among other things) a binary "tensor product" functor $\otimes$: $C \times C \to C$ and an "associativity" natural isomorphism α, whose components are isomorphisms $\alpha\ (a,\ b,\ c)$: $(a \otimes b) \otimes c \to a \otimes (b \otimes c)$ for objects a, b, and c of C. This way of saying things avoids an explicit definition of the functors that are the domain and codomain of α and, in particular, what category serves as the domain of these functors. The domain category is in fact the product category $C \times C \times C$ and the domain and codomain of α are the functors $T\ o\ (T \times C)$: $C \times C \times C \to C$ and $T\ o\ (C \times T)$: $C \times C \times C \to C$. In a formal development, though, we can't gloss over the fact that $C \times C \times C$ has to mean either $C \times (C \times C)$ or $(C \times C) \times C$, which are not formally identical, and that associativities are somehow involved in the definitions of the functors $T\ o\ (T \times C)$ and $T\ o\ (C \times T)$. Here we use the *binary-endofunctor* locale to codify our choices about what $C \times C \times C$, $T\ o\ (T \times C)$, and $T\ o\ (C \times T)$ actually mean. In particular, we choose $C \times C \times C$ to be $C \times (C \times C)$ and define the functors $T\ o\ (T \times C)$, and $T\ o\ (C \times T)$ accordingly.

Our primary definition for "monoidal category" follows the somewhat non-traditional development in [2]. There a monoidal category is defined to be a category C equipped with a binary *tensor product* functor T: $C \times C \to C$, an *associativity isomorphism*, which is a natural isomorphism α: $T\ o\ (T \times C) \to T\ o\ (C \times T)$, a *unit object* $\mathcal{I}$ of C, and an isomorphism ι: $T\ (\mathcal{I},\ \mathcal{I}) \to \mathcal{I}$, subject to two axioms: the *pentagon axiom*, which expresses the commutativity of certain pentagonal diagrams involving components of α, and the left and right *unit axioms*, which state that the endofunctors $T\ (\mathcal{I},\ -)$ and $T\ (-,$

$\mathcal{I}$) are equivalences of categories. This definition is formalized in the *monoidal-category* locale.

In more traditional developments, the definition of monoidal category involves additional left and right *unitor* isomorphisms λ and ϱ and associated axioms involving their components. However, as is shown in [2] and formalized here, the unitors are uniquely determined by α and their values $\lambda(\mathcal{I})$ and $\varrho(\mathcal{I})$ at $\mathcal{I}$, which coincide. Treating λ and ϱ as defined notions results in a more economical basic definition of monoidal category that requires less data to be given, and has a similar effect on the definition of "monoidal functor." Moreover, in the context of the formalization of categories that we use here, the unit object $\mathcal{I}$ also need not be given separately, as it can be obtained as the codomain of the isomorphism ι.

```
locale monoidal-category =
  category C +
  CC: product-category C C +
  CCC: product-category C CC.comp +
  T: binary-endofunctor C T +
  α: natural-isomorphism CCC.comp C T.ToTC T.ToCT α +
  L: equivalence-functor C C λf. T (cod ι, f) +
  R: equivalence-functor C C λf. T (f, cod ι)
for C :: 'a comp        (infixr · 55)
and T :: 'a * 'a ⇒ 'a
and α :: 'a * 'a * 'a ⇒ 'a
and ι :: 'a +
assumes unit-in-hom-ax: «ι : T (cod ι, cod ι) → cod ι»
and unit-is-iso: iso ι
and pentagon: ⟦ ide a; ide b; ide c; ide d ⟧ ⟹
              T (a, α (b, c, d)) · α (a, T (b, c), d) · T (α (a, b, c), d) =
              α (a, b, T (c, d)) · α (T (a, b), c, d)
begin
```

We now define helpful notation and abbreviations to improve readability. We did not define and use the notation $\otimes$ for the tensor product in the definition of the locale because to define $\otimes$ as a binary operator requires that it be in curried form, whereas for T to be a binary functor requires that it take a pair as its argument.

```
  abbreviation unity :: 'a (ℐ)
  where unity ≡ cod ι

  abbreviation L :: 'a ⇒ 'a
  where L f ≡ T (ℐ, f)

  abbreviation R :: 'a ⇒ 'a
  where R f ≡ T (f, ℐ)

  abbreviation tensor      (infixr ⊗ 53)
  where f ⊗ g ≡ T (f, g)

  abbreviation assoc       (a[-, -, -])
```

where a[a, b, c] $\equiv \alpha\ (a,\ b,\ c)$

In HOL we can just give the definitions of the left and right unitors "up front" without any preliminary work. Later we will have to show that these definitions have the right properties. The next two definitions define the values of the unitors when applied to identities; that is, their components as natural transformations.

definition *lunit* (l[-])
where *lunit* $a \equiv$ *THE* f. «$f : \mathcal{I} \otimes a \to a$» $\wedge\ \mathcal{I} \otimes f = (\iota \otimes a) \cdot$ *inv* a[$\mathcal{I}$, $\mathcal{I}$, a]

definition *runit* (r[-])
where *runit* $a \equiv$ *THE* f. «$f : a \otimes \mathcal{I} \to a$» $\wedge\ f \otimes \mathcal{I} = (a \otimes \iota) \cdot$ a[a, $\mathcal{I}$, $\mathcal{I}$]

We now embark upon a development of the consequences of the monoidal category axioms. One of our objectives is to be able to show that an interpretation of the *monoidal-category* locale induces an interpretation of a locale corresponding to a more traditional definition of monoidal category. Another is to obtain the facts we need to prove the coherence theorem.

lemma *unit-in-hom* [*intro*]:
shows «$\iota : \mathcal{I} \otimes \mathcal{I} \to \mathcal{I}$»
using *unit-in-hom-ax* **by** *force*

lemma *ide-unity* [*simp*]:
shows *ide* $\mathcal{I}$
using *unit-in-hom* **by** *auto*

lemma *tensor-in-hom* [*simp*]:
assumes «$f : a \to b$» **and** «$g : c \to d$»
shows «$f \otimes g : a \otimes c \to b \otimes d$»
using *assms T.preserves-hom CC.arr-char* **by** *simp*

lemma *tensor-in-homI* [*intro*]:
assumes «$f : a \to b$» **and** «$g : c \to d$» **and** $x = a \otimes c$ **and** $y = b \otimes d$
shows «$f \otimes g : x \to y$»
using *assms tensor-in-hom*
by *force*

lemma *arr-tensor* [*simp*]:
assumes *arr* f **and** *arr* g
shows *arr* $(f \otimes g)$
using *assms* **by** *simp*

lemma *dom-tensor* [*simp*]:
assumes «$f : a \to b$» **and** «$g : c \to d$»
shows *dom* $(f \otimes g) = a \otimes c$
using *assms* **by** *fastforce*

lemma *cod-tensor* [*simp*]:
assumes «$f : a \to b$» **and** «$g : c \to d$»
shows *cod* $(f \otimes g) = b \otimes d$

using *assms* **by** *fastforce*

lemma *tensor-preserves-ide* [*simp*]:
assumes *ide a* **and** *ide b*
shows *ide* (*a* ⊗ *b*)
using *assms T.preserves-ide CC.ide-char* **by** *simp*

lemma *tensor-preserves-iso* [*simp*]:
assumes *iso f* **and** *iso g*
shows *iso* (*f* ⊗ *g*)
using *assms* **by** *simp*

lemma *inv-tensor* [*simp*]:
assumes *iso f* **and** *iso g*
shows *inv* (*f* ⊗ *g*) = *inv f* ⊗ *inv g*
using *assms T.preserves-inv* **by** *auto*

lemma *interchange*:
assumes *seq h g* **and** *seq h′ g′*
shows (*h* ⊗ *h′*) · (*g* ⊗ *g′*) = *h* · *g* ⊗ *h′* · *g′*
using *assms T.preserves-comp* [*of* (*h*, *h′*) (*g*, *g′*)] **by** *simp*

lemma *α-simp*:
assumes *arr f* **and** *arr g* **and** *arr h*
shows *α* (*f*, *g*, *h*) = (*f* ⊗ *g* ⊗ *h*) · a[*dom f*, *dom g*, *dom h*]
using *assms α.is-natural-1* [*of* (*f*, *g*, *h*)] **by** *simp*

lemma *assoc-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «a[*a*, *b*, *c*] : (*a* ⊗ *b*) ⊗ *c* → *a* ⊗ *b* ⊗ *c*»
using *assms CCC.in-homE* **by** *auto*

lemma *arr-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* a[*a*, *b*, *c*]
using *assms assoc-in-hom* **by** *simp*

lemma *dom-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *dom* a[*a*, *b*, *c*] = (*a* ⊗ *b*) ⊗ *c*
using *assms assoc-in-hom* **by** *simp*

lemma *cod-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *cod* a[*a*, *b*, *c*] = *a* ⊗ *b* ⊗ *c*
using *assms assoc-in-hom* **by** *simp*

lemma *assoc-naturality*:
assumes *arr f0* **and** *arr f1* **and** *arr f2*

shows a[*cod f0*, *cod f1*, *cod f2*] · ((*f0* ⊗ *f1*) ⊗ *f2*) =
(*f0* ⊗ *f1* ⊗ *f2*) · a[*dom f0*, *dom f1*, *dom f2*]
using *assms* α.*naturality* **by** *auto*

lemma *iso-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *iso* a[*a*, *b*, *c*]
using *assms* α.*preserves-iso* **by** *simp*

The next result uses the fact that the functor L is an equivalence (and hence faithful) to show the existence of a unique solution to the characteristic equation used in the definition of a component l[a] of the left unitor. It follows that l[a], as given by our definition using definite description, satisfies this characteristic equation and is therefore uniquely determined by by ⊗, α, and ι.

lemma *lunit-char*:
assumes *ide a*
shows «l[*a*] : $\mathcal{I}$ ⊗ *a* → *a*» **and** $\mathcal{I}$ ⊗ l[*a*] = (ι ⊗ *a*) · *inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*]
and ∃!*f*. «*f* : $\mathcal{I}$ ⊗ *a* → *a*» ∧ $\mathcal{I}$ ⊗ *f* = (ι ⊗ *a*) · *inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*]
proof −
obtain *F* η ε **where** *L*: *equivalence-of-categories C C F* (λ*f*. $\mathcal{I}$ ⊗ *f*) η ε
using *L.induces-equivalence* **by** *auto*
interpret *L*: *equivalence-of-categories C C F* ‹λ*f*. $\mathcal{I}$ ⊗ *f*› η ε
using *L* **by** *auto*
let *?P* = λ*f*. «*f* : $\mathcal{I}$ ⊗ *a* → *a*» ∧ $\mathcal{I}$ ⊗ *f* = (ι ⊗ *a*) · *inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*]
show ∃!*f*. *?P f*
proof −
have ∃*f*. *?P f*
proof −
have «(ι ⊗ *a*) · *inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*] : $\mathcal{I}$ ⊗ $\mathcal{I}$ ⊗ *a* → $\mathcal{I}$ ⊗ *a*»
proof
show «ι ⊗ *a* : ($\mathcal{I}$ ⊗ $\mathcal{I}$) ⊗ *a* → $\mathcal{I}$ ⊗ *a*»
using *assms ide-in-hom* **by** *blast*
show «*inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*] : $\mathcal{I}$ ⊗ $\mathcal{I}$ ⊗ *a* → ($\mathcal{I}$ ⊗ $\mathcal{I}$) ⊗ *a*»
using *assms* **by** *auto*
qed
moreover have *ide* ($\mathcal{I}$ ⊗ *a*) **using** *assms* **by** *simp*
ultimately show *?thesis*
using *assms L.is-full* **by** *blast*
qed
moreover have ⋀*f f'*. *?P f* ⟹ *?P f'* ⟹ *f* = *f'*
by (*metis L.is-faithful in-homE*)
ultimately show *?thesis* **by** *blast*
qed
hence *1*: *?P* l[*a*]
unfolding *lunit-def* **using** *theI'* [*of ?P*] **by** *auto*
show «l[*a*] : $\mathcal{I}$ ⊗ *a* → *a*» **using** *1* **by** *fast*
show $\mathcal{I}$ ⊗ l[*a*] = (ι ⊗ *a*) · *inv* a[$\mathcal{I}$, $\mathcal{I}$, *a*] **using** *1* **by** *fast*
qed

lemma *lunit-in-hom* [*intro*]:
assumes *ide a*
shows «l[*a*] : $\mathcal{I} \otimes a \to a$»
  using *assms lunit-char*(*1*) **by** *blast*

lemma *arr-lunit* [*simp*]:
assumes *ide a*
shows *arr* l[*a*]
  using *assms lunit-in-hom* **by** *auto*

lemma *dom-lunit* [*simp*]:
assumes *ide a*
shows *dom* l[*a*] = $\mathcal{I} \otimes a$
  using *assms lunit-in-hom* **by** *auto*

lemma *cod-lunit* [*simp*]:
assumes *ide a*
shows *cod* l[*a*] = *a*
  using *assms lunit-in-hom* **by** *auto*

As the right-hand side of the characteristic equation for $\mathcal{I} \otimes$ l[a] is an isomorphism, and the equivalence functor L reflects isomorphisms, it follows that l[a] is an isomorphism.

lemma *iso-lunit* [*simp*]:
assumes *ide a*
shows *iso* l[*a*]
  using *assms lunit-char*(*2*) *unit-is-iso ide-unity isos-compose iso-assoc iso-inv-iso*
    unit-in-hom L.reflects-iso arr-lunit arr-tensor ideD(*1*) *ide-is-iso lunit-in-hom*
    tensor-preserves-iso
  by *metis*

To prove that an arrow f is equal to l[a] we need only show that it is parallel to l[a] and that $\mathcal{I} \otimes f$ satisfies the same characteristic equation as $\mathcal{I} \otimes$ l[a] does.

lemma *lunit-eqI*:
assumes «$f : \mathcal{I} \otimes a \to a$» **and** $\mathcal{I} \otimes f = (\iota \otimes a) \cdot$ *inv* a[$\mathcal{I}$, $\mathcal{I}$, a]
shows f = l[a]
proof −
  have *ide a* **using** *assms*(*1*) **by** *auto*
  thus *?thesis*
    using *assms lunit-char the1-equality* **by** *blast*
qed

The next facts establish the corresponding results for the components of the right unitor.

lemma *runit-char*:
assumes *ide a*
shows «r[*a*] : $a \otimes \mathcal{I} \to a$» **and** r[$a$] $\otimes \mathcal{I} = (a \otimes \iota) \cdot$ a[a, $\mathcal{I}$, $\mathcal{I}$]
and $\exists!f.$ «$f : a \otimes \mathcal{I} \to a$» $\wedge$ $f \otimes \mathcal{I} = (a \otimes \iota) \cdot$ a[a, $\mathcal{I}$, $\mathcal{I}$]
proof −
  obtain F η ε **where** *R*: *equivalence-of-categories C C F* ($\lambda f.\ f \otimes \mathcal{I}$) η ε

```
    using R.induces-equivalence by auto
  interpret R: equivalence-of-categories C C F ‹λf. f ⊗ I› η ε
    using R by auto
  let ?P = λf. «f : a ⊗ I → a» ∧ f ⊗ I = (a ⊗ ι) · a[a, I, I]
  show ∃!f. ?P f
  proof −
    have ∃f. ?P f
    proof −
      have «(a ⊗ ι) · a[a, I, I] : (a ⊗ I) ⊗ I → a ⊗ I»
        using assms by fastforce
      moreover have ide (a ⊗ I) using assms by simp
      ultimately show ?thesis
        using assms R.is-full [of a a ⊗ I (a ⊗ ι) · a[a, I, I]] by blast
    qed
    moreover have ⋀f f'. ?P f ⟹ ?P f' ⟹ f = f'
      by (metis R.is-faithful in-homE)
    ultimately show ?thesis by blast
  qed
  hence 1: ?P r[a] unfolding runit-def using theI' [of ?P] by fast
  show «r[a] : a ⊗ I → a» using 1 by fast
  show r[a] ⊗ I = (a ⊗ ι) · a[a, I, I] using 1 by fast
qed

lemma runit-in-hom [intro]:
assumes ide a
shows «r[a] : a ⊗ I → a»
  using assms runit-char(1) by blast

lemma arr-runit [simp]:
assumes ide a
shows arr r[a]
  using assms runit-in-hom by blast

lemma dom-runit [simp]:
assumes ide a
shows dom r[a] = a ⊗ I
  using assms runit-in-hom by blast

lemma cod-runit [simp]:
assumes ide a
shows cod r[a] = a
  using assms runit-in-hom by blast

lemma runit-eqI:
assumes «f : a ⊗ I → a» and f ⊗ I = (a ⊗ ι) · a[a, I, I]
shows f = r[a]
proof −
  have ide a using assms(1) by auto
  thus ?thesis
```

```
    using assms runit-char the1-equality by blast
  qed

  lemma iso-runit [simp]:
  assumes ide a
  shows iso r[a]
    using assms unit-is-iso iso-inv-iso isos-compose ide-is-iso R.preserves-reflects-arr
          arrI ide-unity iso-assoc runit-char tensor-preserves-iso R.reflects-iso
    by metis
```

We can now show that the components of the left and right unitors have the naturality properties required of a natural transformation.

```
  lemma lunit-naturality:
  assumes arr f
  shows l[cod f] · (I ⊗ f) = f · l[dom f]
  proof −
    interpret α′: inverse-transformation CCC.comp C T.ToTC T.ToCT α ..
    have par: par (l[cod f] · (I ⊗ f)) (f · l[dom f])
      using assms by simp
    moreover have I ⊗ l[cod f] · (I ⊗ f) = I ⊗ f · l[dom f]
    proof −
      have I ⊗ l[cod f] · (I ⊗ f) = ((ι ⊗ cod f) · ((I ⊗ I) ⊗ f)) · inv a[I, I, dom f]
        using assms interchange [of I I I ⊗ f l[cod f]] lunit-char(2)
              α′.naturality [of (I, I, f)] comp-assoc
        by auto
      also have ... = ((I ⊗ f) · (ι ⊗ dom f)) · inv a[I, I, dom f]
        using assms interchange comp-arr-dom comp-cod-arr unit-in-hom by auto
      also have ... = (I ⊗ f) · (I ⊗ l[dom f])
        using assms lunit-char(2) comp-assoc by auto
      also have ... = I ⊗ f · l[dom f]
        using assms interchange L.preserves-comp par by metis
      finally show ?thesis by blast
    qed
    ultimately show l[cod f] · (I ⊗ f) = f · l[dom f]
      using L.is-faithful by metis
  qed

  lemma runit-naturality:
  assumes arr f
  shows r[cod f] · (f ⊗ I) = f · r[dom f]
  proof −
    have par: par (r[cod f] · (f ⊗ I)) (f · r[dom f])
      using assms by force
    moreover have r[cod f] · (f ⊗ I) ⊗ I = f · r[dom f] ⊗ I
    proof −
      have r[cod f] · (f ⊗ I) ⊗ I = (cod f ⊗ ι) · a[cod f, I, I] · ((f ⊗ I) ⊗ I)
        using assms interchange [of I I I ⊗ f r[cod f]] runit-char(2)
              comp-assoc
        by auto
```

```
    also have ... = (cod f ⊗ ι) · (f ⊗ I ⊗ I) ·  a[dom f, I, I]
      using assms α.naturality [of (f, I, I)] by auto
    also have ... = ((cod f ⊗ ι) · (f ⊗ I ⊗ I)) · a[dom f, I, I]
      using comp-assoc by simp
    also have ... = ((f ⊗ I) · (dom f ⊗ ι)) · a[dom f, I, I]
      using assms unit-in-hom interchange comp-arr-dom comp-cod-arr by auto
    also have ... = (f ⊗ I) · (r[dom f] ⊗ I)
      using assms runit-char comp-assoc by auto
    also have ... = f · r[dom f] ⊗ I
      using assms interchange R.preserves-comp par by metis
    finally show ?thesis by blast
  qed
  ultimately show r[cod f] · (f ⊗ I) = f · r[dom f]
    using R.is-faithful by metis
qed
```

The next two definitions extend the unitors to all arrows, not just identities. Unfortunately, the traditional symbol λ for the left unitor is already reserved for a higher purpose, so we have to make do with a poor substitute.

```
abbreviation ł
where ł f ≡ if arr f then f · l[dom f] else null

abbreviation ϱ
where ϱ f ≡ if arr f then f · r[dom f] else null

lemma ł-ide-simp:
assumes ide a
shows ł a = l[a]
  using assms lunit-char comp-cod-arr ide-in-hom by (metis in-homE)

lemma ϱ-ide-simp:
assumes ide a
shows ϱ a = r[a]
  using assms runit-char [of a] comp-cod-arr by auto

end

context monoidal-category
begin

  sublocale ł: natural-transformation C C L map ł
  proof −
    interpret ł: transformation-by-components C C L map ‹λa. l[a]›
      using lunit-in-hom lunit-naturality unit-in-hom-ax L.is-extensional
      by (unfold-locales, auto)
    have ł.map = ł
      using ł.is-natural-1 ł.is-extensional by auto
    thus natural-transformation C C L map ł
      using ł.natural-transformation-axioms by auto
```

qed

sublocale ℓ: *natural-isomorphism C C L map* ℓ
  apply *unfold-locales*
  using *iso-lunit* ℓ*-ide-simp* **by** *simp*

sublocale ϱ: *natural-transformation C C R map* ϱ
proof −
  interpret ϱ: *transformation-by-components C C R map* ‹λ*a*. r[*a*]›
    using *runit-naturality unit-in-hom-ax R.is-extensional*
    by (*unfold-locales*, *auto*)
  have ϱ.*map* = ϱ
    using ϱ.*is-natural-1* ϱ.*is-extensional* **by** *auto*
  thus *natural-transformation C C R map* ϱ
    using ϱ.*natural-transformation-axioms* **by** *auto*
qed

sublocale ϱ: *natural-isomorphism C C R map* ϱ
  apply *unfold-locales*
  using ϱ*-ide-simp* **by** *simp*

sublocale ℓ′: *inverse-transformation C C L map* ℓ **..**
sublocale ϱ′: *inverse-transformation C C R map* ϱ **..**
sublocale α′: *inverse-transformation CCC.comp C T.ToTC T.ToCT* α **..**

abbreviation α′
where α′ ≡ α′.*map*

abbreviation *assoc′* (a^{-1}[-, -, -])
where a^{-1}[*a*, *b*, *c*] ≡ *inv* a[*a*, *b*, *c*]

lemma α′*-ide-simp*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows α′ (*a*, *b*, *c*) = a^{-1}[*a*, *b*, *c*]
  using *assms* α′.*inverts-components inverse-unique* **by** *force*

lemma α′*-simp*:
assumes *arr f* **and** *arr g* **and** *arr h*
shows α′ (*f*, *g*, *h*) = ((*f* ⊗ *g*) ⊗ *h*) · a^{-1}[*dom f*, *dom g*, *dom h*]
  using *assms T.ToTC-simp* α′.*is-natural-1* α′*-ide-simp* **by** *force*

lemma *assoc-inv*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *inverse-arrows* a[*a*, *b*, *c*] a^{-1}[*a*, *b*, *c*]
  using *assms inv-is-inverse* **by** *simp*

lemma *assoc′-in-hom* [*intro*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «a^{-1}[*a*, *b*, *c*] : *a* ⊗ *b* ⊗ *c* → (*a* ⊗ *b*) ⊗ *c*»

using *assms* **by** *auto*

lemma *arr-assoc'* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* $\mathrm{a}^{-1}[a, b, c]$
using *assms* **by** *simp*

lemma *dom-assoc'* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *dom* $\mathrm{a}^{-1}[a, b, c] = a \otimes b \otimes c$
using *assms* **by** *simp*

lemma *cod-assoc'* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *cod* $\mathrm{a}^{-1}[a, b, c] = (a \otimes b) \otimes c$
using *assms* **by** *simp*

lemma *comp-assoc-assoc'* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathrm{a}[a, b, c] \cdot \mathrm{a}^{-1}[a, b, c] = a \otimes (b \otimes c)$
and $\mathrm{a}^{-1}[a, b, c] \cdot \mathrm{a}[a, b, c] = (a \otimes b) \otimes c$
using *assms assoc-inv comp-arr-inv comp-inv-arr* **by** *auto*

lemma *assoc'-naturality*:
assumes *arr f0* **and** *arr f1* **and** *arr f2*
shows $((f0 \otimes f1) \otimes f2) \cdot \mathrm{a}^{-1}[dom\ f0, dom\ f1, dom\ f2] =$
$\mathrm{a}^{-1}[cod\ f0, cod\ f1, cod\ f2] \cdot (f0 \otimes f1 \otimes f2)$
using *assms* $\alpha'.naturality$ **by** *auto*

abbreviation $\mathfrak{l}'$
where $\mathfrak{l}' \equiv \mathfrak{l}'.map$

abbreviation *lunit'* (l^{-1}[-])
where $\mathrm{l}^{-1}[a] \equiv inv\ \mathrm{l}[a]$

lemma $\mathfrak{l}'$*-ide-simp*:
assumes *ide a*
shows $\mathfrak{l}'.map\ a = \mathrm{l}^{-1}[a]$
using *assms* $\mathfrak{l}'.inverts\text{-}components$ $\mathfrak{l}$*-ide-simp inverse-unique* **by** *force*

lemma *lunit-inv*:
assumes *ide a*
shows *inverse-arrows* $\mathrm{l}[a]$ $\mathrm{l}^{-1}[a]$
using *assms inv-is-inverse* **by** *simp*

lemma *lunit'-in-hom* [*intro*]:
assumes *ide a*
shows «$\mathrm{l}^{-1}[a] : a \to \mathcal{I} \otimes a$»
using *assms* **by** *auto*

lemma *comp-lunit-lunit'* [*simp*]:
assumes *ide a*
shows $\mathrm{l}[a] \cdot \mathrm{l}^{-1}[a] = a$
and $\mathrm{l}^{-1}[a] \cdot \mathrm{l}[a] = \mathcal{I} \otimes a$
proof −
  show $\mathrm{l}[a] \cdot \mathrm{l}^{-1}[a] = a$
    using *assms comp-arr-inv lunit-inv* **by** *fastforce*
  show $\mathrm{l}^{-1}[a] \cdot \mathrm{l}[a] = \mathcal{I} \otimes a$
    using *assms comp-arr-inv lunit-inv* **by** *fastforce*
qed

lemma *lunit'-naturality*:
assumes *arr f*
shows $(\mathcal{I} \otimes f) \cdot \mathrm{l}^{-1}[dom\ f] = \mathrm{l}^{-1}[cod\ f] \cdot f$
  using *assms* $\mathfrak{l}'$*.naturality* $\mathfrak{l}'$*-ide-simp* **by** *simp*

abbreviation ϱ'
where $\varrho' \equiv \varrho'.map$

abbreviation *runit'* (r^{-1}[-])
where $\mathrm{r}^{-1}[a] \equiv inv\ \mathrm{r}[a]$

lemma ϱ'*-ide-simp*:
assumes *ide a*
shows $\varrho'.map\ a = \mathrm{r}^{-1}[a]$
  using *assms* ϱ'*.inverts-components* ϱ*-ide-simp inverse-unique* **by** *auto*

lemma *runit-inv*:
assumes *ide a*
shows *inverse-arrows* $\mathrm{r}[a]$ $\mathrm{r}^{-1}[a]$
  using *assms inv-is-inverse* **by** *simp*

lemma *runit'-in-hom* [*intro*]:
assumes *ide a*
shows «$\mathrm{r}^{-1}[a] : a \to a \otimes \mathcal{I}$»
  using *assms* **by** *auto*

lemma *comp-runit-runit'* [*simp*]:
assumes *ide a*
shows $\mathrm{r}[a] \cdot \mathrm{r}^{-1}[a] = a$
and $\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a] = a \otimes \mathcal{I}$
proof −
  show $\mathrm{r}[a] \cdot \mathrm{r}^{-1}[a] = a$
    using *assms runit-inv* **by** *fastforce*
  show $\mathrm{r}^{-1}[a] \cdot \mathrm{r}[a] = a \otimes \mathcal{I}$
    using *assms runit-inv* **by** *fastforce*
qed

lemma *runit′-naturality*:
assumes *arr f*
shows $(f \otimes \mathcal{I}) \cdot \mathrm{r}^{-1}[dom\ f] = \mathrm{r}^{-1}[cod\ f] \cdot f$
using *assms ϱ′.naturality ϱ′-ide-simp* **by** *simp*

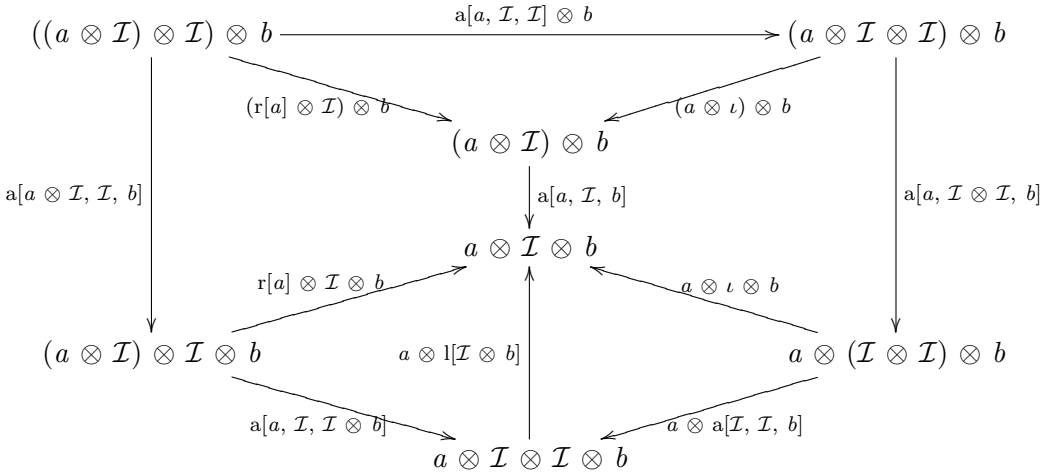
lemma *lunit-commutes-with-L*:
assumes *ide a*
shows $\mathrm{l}[\mathcal{I} \otimes a] = \mathcal{I} \otimes \mathrm{l}[a]$
using *assms lunit-naturality lunit-in-hom iso-lunit iso-is-section*
section-is-mono monoE L.preserves-ide arrI cod-lunit dom-lunit seqI
by *metis*

lemma *runit-commutes-with-R*:
assumes *ide a*
shows $\mathrm{r}[a \otimes \mathcal{I}] = \mathrm{r}[a] \otimes \mathcal{I}$
using *assms runit-naturality runit-in-hom iso-runit iso-is-section*
section-is-mono monoE R.preserves-ide arrI cod-runit dom-runit seqI
by *metis*

The components of the left and right unitors are related via a "triangle" diagram that also involves the associator. The proof follows [2], Proposition 2.2.3.

lemma *triangle*:
assumes *ide a* **and** *ide b*
shows $(a \otimes \mathrm{l}[b]) \cdot \mathrm{a}[a, \mathcal{I}, b] = \mathrm{r}[a] \otimes b$
proof −

We show that the lower left triangle in the following diagram commutes.



have ∗: $(a \otimes \mathrm{l}[\mathcal{I} \otimes b]) \cdot \mathrm{a}[a, \mathcal{I}, \mathcal{I} \otimes b] = \mathrm{r}[a] \otimes \mathcal{I} \otimes b$
proof −
have *1*: $((a \otimes \mathrm{l}[\mathcal{I} \otimes b]) \cdot \mathrm{a}[a, \mathcal{I}, \mathcal{I} \otimes b]) \cdot \mathrm{a}[a \otimes \mathcal{I}, \mathcal{I}, b]$
$= (\mathrm{r}[a] \otimes \mathcal{I} \otimes b) \cdot \mathrm{a}[a \otimes \mathcal{I}, \mathcal{I}, b]$

```
  proof −
    have ((a ⊗ l[ℐ ⊗ b]) · a[a, ℐ, ℐ ⊗ b]) · a[a ⊗ ℐ, ℐ, b] =
          ((a ⊗ l[ℐ ⊗ b]) · (a ⊗ a[ℐ, ℐ, b])) · a[a, ℐ ⊗ ℐ, b] · (a[a, ℐ, ℐ] ⊗ b)
      using assms pentagon comp-assoc by auto
    also have ... = (a ⊗ ((ℐ ⊗ l[b]) · a[ℐ, ℐ, b])) · a[a, ℐ ⊗ ℐ, b] · (a[a, ℐ, ℐ] ⊗ b)
      using assms interchange lunit-commutes-with-L by simp
    also have ... = ((a ⊗ (ι ⊗ b)) · a[a, ℐ ⊗ ℐ, b]) · (a[a, ℐ, ℐ] ⊗ b)
      using assms lunit-char unit-in-hom comp-arr-dom comp-assoc by auto
    also have ... = (a[a, ℐ, b] · ((a ⊗ ι) ⊗ b)) · (a[a, ℐ, ℐ] ⊗ b)
      using assms unit-in-hom assoc-naturality [of a ι b] by fastforce
    also have ... = a[a, ℐ, b] · ((r[a] ⊗ ℐ) ⊗ b)
      using assms unit-in-hom interchange runit-char(2) comp-assoc by auto
    also have ... = (r[a] ⊗ ℐ ⊗ b) · a[a ⊗ ℐ, ℐ, b]
      using assms assoc-naturality [of r[a] ℐ b] by simp
    finally show ?thesis by blast
  qed
  show ?thesis
  proof −
    have epi a[a ⊗ ℐ, ℐ, b]
      using assms iso-assoc iso-is-retraction retraction-is-epi by simp
    thus ?thesis
      using 1 assms epiE [of a[a ⊗ ℐ, ℐ, b] (a ⊗ l[ℐ ⊗ b]) · a[a, ℐ, ℐ ⊗ b]]
      by fastforce
  qed
qed
```

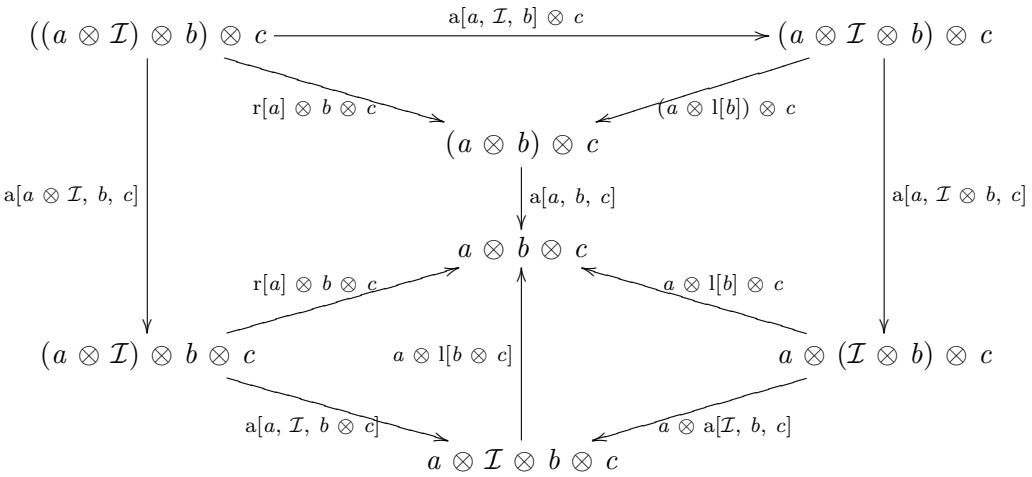
In [2] it merely states that the preceding result suffices "because any object of C is isomorphic to one of the form $\mathcal{I} \otimes b$." However, it seems a little bit more involved than that to formally transport the equation $(*)$ along the isomorphism $l[b]$ from $\mathcal{I} \otimes b$ to b.

```
  have (a ⊗ l[b]) · a[a, ℐ, b] = ((a ⊗ l[b]) · (a ⊗ l[ℐ ⊗ b]) · (a ⊗ ℐ ⊗ l⁻¹[b])) ·
                                 (a ⊗ ℐ ⊗ l[b]) · a[a, ℐ, ℐ ⊗ b] · ((a ⊗ ℐ) ⊗ l⁻¹[b])
  proof −
    have a[a, ℐ, b] = (a ⊗ ℐ ⊗ l[b]) · a[a, ℐ, ℐ ⊗ b] · ((a ⊗ ℐ) ⊗ l⁻¹[b])
    proof −
      have (a ⊗ ℐ ⊗ l[b]) · a[a, ℐ, ℐ ⊗ b] · ((a ⊗ ℐ) ⊗ l⁻¹[b])
            = ((a ⊗ ℐ ⊗ l[b]) · (a ⊗ ℐ ⊗ l⁻¹[b])) · a[a, ℐ, b]
        using assms assoc-naturality [of a ℐ l⁻¹[b]] comp-assoc by simp
      also have ... = a[a, ℐ, b]
        using assms inv-is-inverse interchange comp-cod-arr by simp
      finally show ?thesis by auto
    qed
    moreover have a ⊗ l[b] = (a ⊗ l[b]) · (a ⊗ l[ℐ ⊗ b]) · (a ⊗ ℐ ⊗ l⁻¹[b])
      using assms lunit-commutes-with-L comp-arr-dom interchange by auto
    ultimately show ?thesis by argo
  qed
  also have ... = (a ⊗ l[b]) · (a ⊗ l[ℐ ⊗ b]) · ((a ⊗ ℐ ⊗ l⁻¹[b]) · (a ⊗ ℐ ⊗ l[b])) ·
                  a[a, ℐ, ℐ ⊗ b] · ((a ⊗ ℐ) ⊗ l⁻¹[b])
    using assms comp-assoc by auto
  also have ... = (a ⊗ l[b]) · ((a ⊗ l[ℐ ⊗ b]) · a[a, ℐ, ℐ ⊗ b]) · ((a ⊗ ℐ) ⊗ l⁻¹[b])
```

using *assms interchange comp-cod-arr comp-assoc* **by** *auto*
also have ... = r[a] ⊗ b
using *assms* ∗ *interchange runit-char*(*1*) *comp-arr-dom comp-cod-arr* **by** *auto*
finally show *?thesis* **by** *blast*
qed

lemma *lunit-tensor-gen*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows (a ⊗ l[b ⊗ c]) · (a ⊗ a[$\mathcal{I}$, b, c]) = a ⊗ l[b] ⊗ c
proof −

We show that the lower right triangle in the following diagram commutes.



have ((a ⊗ l[b ⊗ c]) · (a ⊗ a[$\mathcal{I}$, b, c])) · (a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c)) =
(((a ⊗ l[b ⊗ c]) · a[a, $\mathcal{I}$, b ⊗ c]) · a[a ⊗ $\mathcal{I}$, b, c]
using *assms pentagon comp-assoc* **by** *simp*
also have ... = (r[a] ⊗ (b ⊗ c)) · a[a ⊗ $\mathcal{I}$, b, c]
using *assms triangle* **by** *auto*
also have ... = a[a, b, c] · ((r[a] ⊗ b) ⊗ c)
using *assms assoc-naturality* [*of* r[a] b c] **by** *auto*
also have ... = (a[a, b, c] · ((a ⊗ l[b]) ⊗ c)) · (a[a, $\mathcal{I}$, b] ⊗ c)
using *assms triangle interchange comp-assoc* **by** *auto*
also have ... = (a ⊗ (l[b] ⊗ c)) · (a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c))
using *assms assoc-naturality* [*of a* l[b] c] *comp-assoc* **by** *auto*
finally have *1*: ((a ⊗ l[b ⊗ c]) · (a ⊗ a[$\mathcal{I}$, b, c])) · a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c)
= (a ⊗ (l[b] ⊗ c)) · a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c)
by *blast*

The result follows by cancelling the isomorphism a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c)

have *2*: *iso* (a[a, $\mathcal{I}$ ⊗ b, c] · (a[a, $\mathcal{I}$, b] ⊗ c))
using *assms isos-compose* **by** *simp*
moreover have

```
    seq ((a ⊗ l[b ⊗ c]) · (a ⊗ a[I, b, c])) (a[a, I ⊗ b, c] · (a[a, I, b] ⊗ c))
    using assms by auto
  moreover have seq (a ⊗ (l[b] ⊗ c)) (a[a, I ⊗ b, c] · (a[a, I, b] ⊗ c))
    using assms by auto
  ultimately show ?thesis
    using 1 2 assms iso-is-retraction retraction-is-epi
        epiE [of a[a, I ⊗ b, c] · (a[a, I, b] ⊗ c)
                 (a ⊗ l[b ⊗ c]) · (a ⊗ a[I, b, c]) a ⊗ l[b] ⊗ c]
    by auto
qed
```

The following result is quoted without proof as Theorem 7 of [3] where it is attributed to MacLane [4]. It also appears as [5], Exercise 1, page 161. I did not succeed within a few hours to construct a proof following MacLane's hint. The proof below is based on [2], Proposition 2.2.4.

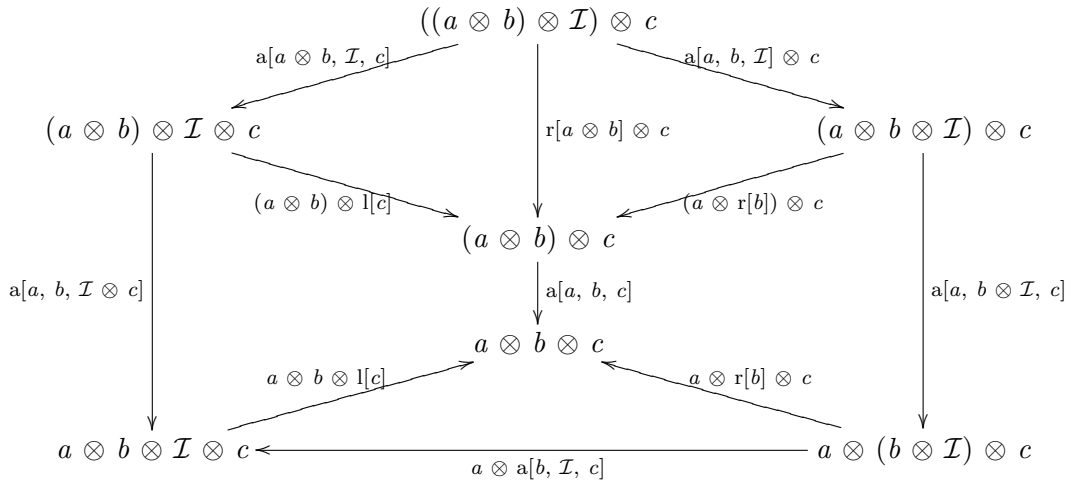
```
lemma lunit-tensor':
assumes ide a and ide b
shows l[a ⊗ b] · a[I, a, b] = l[a] ⊗ b
proof −
  have I ⊗ (l[a ⊗ b] · a[I, a, b]) = I ⊗ (l[a] ⊗ b)
    using assms interchange [of I I] lunit-tensor-gen by simp
  moreover have par (l[a ⊗ b] · a[I, a, b]) (l[a] ⊗ b)
    using assms by simp
  ultimately show ?thesis
    using assms L.is-faithful [of l[a ⊗ b] · a[I, a, b] l[a] ⊗ b] by simp
qed

lemma lunit-tensor:
assumes ide a and ide b
shows l[a ⊗ b] = (l[a] ⊗ b) · a⁻¹[I, a, b]
  using assms lunit-tensor' invert-side-of-triangle by simp
```

We next show the corresponding result for the right unitor.

```
lemma runit-tensor-gen:
assumes ide a and ide b and ide c
shows r[a ⊗ b] ⊗ c = ((a ⊗ r[b]) ⊗ c) · (a[a, b, I] ⊗ c)
proof −
```

We show that the upper right triangle in the following diagram commutes.



have r[$a \otimes b$] $\otimes$ c = (($a \otimes b$) $\otimes$ l[c]) $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *assms triangle* **by** *simp*
also have ... = (a^{-1}[a, b, c] $\cdot$ ($a \otimes b \otimes$ l[c]) $\cdot$ a[a, b, $\mathcal{I} \otimes c$]) $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *assms assoc-naturality* [*of a b* l[c]] *comp-arr-dom comp-cod-arr*
invert-side-of-triangle(*1*)
by *force*
also have ... = a^{-1}[a, b, c] $\cdot$ ($a \otimes b \otimes$ l[c]) $\cdot$ a[a, b, $\mathcal{I} \otimes c$] $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *comp-assoc* **by** *force*
also have ... = a^{-1}[a, b, c] $\cdot$ (($a \otimes$ (r[b] $\otimes$ c)) $\cdot$ ($a \otimes$ a^{-1}[b, $\mathcal{I}$, c])) $\cdot$
a[a, b, $\mathcal{I} \otimes c$] $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *assms triangle* [*of b c*] *interchange invert-side-of-triangle*(*2*) **by** *force*
also have ... = (((($a \otimes$ r[b]) $\otimes$ c) $\cdot$ a^{-1}[a, $b \otimes \mathcal{I}$, c]) $\cdot$ ($a \otimes$ a^{-1}[b, $\mathcal{I}$, c]) $\cdot$
a[a, b, $\mathcal{I} \otimes c$] $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *assms assoc'-naturality* [*of a* r[b] c] *comp-assoc* **by** *force*
also have ... = (($a \otimes$ r[b]) $\otimes$ c) $\cdot$ a^{-1}[a, $b \otimes \mathcal{I}$, c] $\cdot$ ($a \otimes$ a^{-1}[b, $\mathcal{I}$, c]) $\cdot$
a[a, b, $\mathcal{I} \otimes c$] $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c]
using *comp-assoc* **by** *simp*
also have ... = (($a \otimes$ r[b]) $\otimes$ c) $\cdot$ (a[a, b, $\mathcal{I}$] $\otimes$ c)
using *assms pentagon invert-side-of-triangle*(*1*)
invert-side-of-triangle(*1*)
[*of* a[a, b, $\mathcal{I} \otimes c$] $\cdot$ a[$a \otimes b$, $\mathcal{I}$, c] $a \otimes$ a[b, $\mathcal{I}$, c]
a[a, $b \otimes \mathcal{I}$, c] $\cdot$ (a[a, b, $\mathcal{I}$] $\otimes$ c)]
by *force*
finally show *?thesis* **by** *blast*
qed

lemma *runit-tensor*:
assumes *ide a* **and** *ide b*
shows r[$a \otimes b$] = ($a \otimes$ r[b]) $\cdot$ a[a, b, $\mathcal{I}$]
proof −
have (($a \otimes$ r[b]) $\cdot$ a[a, b, $\mathcal{I}$]) $\otimes$ $\mathcal{I}$ = r[$a \otimes b$] $\otimes$ $\mathcal{I}$

```
    using assms interchange [of ℐ ℐ] runit-tensor-gen by simp
  moreover have par ((a ⊗ r[b]) · a[a, b, ℐ]) r[a ⊗ b]
    using assms by simp
  ultimately show ?thesis
    using assms R.is-faithful [of (a ⊗ r[b]) · (a[a, b, ℐ]) r[a ⊗ b]]
    by fastforce
qed

lemma runit-tensor′:
assumes ide a and ide b
shows r[a ⊗ b] · a⁻¹[a, b, ℐ] = a ⊗ r[b]
  using assms runit-tensor invert-side-of-triangle by force
```

Sometimes inverted forms of the triangle and pentagon axioms are useful.

```
lemma triangle′:
assumes ide a and ide b
shows (a ⊗ l[b]) = (r[a] ⊗ b) · a⁻¹[a, ℐ, b]
proof −
  have (r[a] ⊗ b) · a⁻¹[a, ℐ, b] = ((a ⊗ l[b]) · a[a, ℐ, b]) · a⁻¹[a, ℐ, b]
    using assms triangle by auto
  also have ... = (a ⊗ l[b])
    using assms comp-arr-dom comp-assoc by auto
  finally show ?thesis by auto
qed

lemma pentagon′:
assumes ide a and ide b and ide c and ide d
shows ((a⁻¹[a, b, c] ⊗ d) · a⁻¹[a, b ⊗ c, d]) · (a ⊗ a⁻¹[b, c, d])
        = a⁻¹[a ⊗ b, c, d] · a⁻¹[a, b, c ⊗ d]
proof −
  have ((a⁻¹[a, b, c] ⊗ d) · a⁻¹[a, b ⊗ c, d]) · (a ⊗ a⁻¹[b, c, d])
        = inv ((a ⊗ a[b, c, d]) · (a[a, b ⊗ c, d] · (a[a, b, c] ⊗ d)))
    using assms isos-compose inv-comp by simp
  also have ... = inv (a[a, b, c ⊗ d] · a[a ⊗ b, c, d])
    using assms pentagon by auto
  also have ... = a⁻¹[a ⊗ b, c, d] · a⁻¹[a, b, c ⊗ d]
    using assms inv-comp by simp
  finally show ?thesis by auto
qed
```

The following non-obvious fact is Corollary 2.2.5 from [2]. The statement that $\mathrm{l}[\mathcal{I}] = \mathrm{r}[\mathcal{I}]$ is Theorem 6 from [3]. MacLane [5] does not show this, but assumes it as an axiom.

```
lemma unitor-coincidence:
shows l[ℐ] = ι and r[ℐ] = ι
proof −
  have l[ℐ] ⊗ ℐ = (ℐ ⊗ l[ℐ]) · a[ℐ, ℐ, ℐ]
    using lunit-tensor′ [of ℐ ℐ] lunit-commutes-with-L [of ℐ] by simp
  moreover have r[ℐ] ⊗ ℐ = (ℐ ⊗ l[ℐ]) · a[ℐ, ℐ, ℐ]
    using triangle [of ℐ ℐ] by simp
```

```
  moreover have ι ⊗ I = (I ⊗ l[I]) · a[I, I, I]
    using lunit-char comp-arr-dom unit-in-hom comp-assoc by auto
  ultimately have l[I] ⊗ I = ι ⊗ I ∧ r[I] ⊗ I = ι ⊗ I
    by argo
  moreover have par l[I] ι ∧ par r[I] ι
    using unit-in-hom by force
  ultimately have 1: l[I] = ι ∧ r[I] = ι
    using R.is-faithful by metis
  show l[I] = ι using 1 by auto
  show r[I] = ι using 1 by auto
qed

lemma unit-triangle:
shows ι ⊗ I = (I ⊗ ι) · a[I, I, I]
and (ι ⊗ I) · a⁻¹[I, I, I] = I ⊗ ι
  using triangle [of I I] triangle' [of I I] unitor-coincidence by auto
```

The only isomorphism that commutes with ι is $\mathcal{I}$.

```
lemma iso-commuting-with-unit-equals-unity:
assumes «f : I → I» and iso f and f · ι = ι · (f ⊗ f)
shows f = I
proof −
  have I ⊗ f = I ⊗ I
  proof −
    have f ⊗ f = f ⊗ I
      by (metis assms(1,3) iso-cancel-left runit-naturality seqE seqI' unit-in-hom-ax
          unit-is-iso unitor-coincidence(2))
    thus ?thesis
      by (metis assms(1−2) R.preserves-comp comp-cod-arr comp-inv-arr' ideD(1) ide-unity
          in-homE interchange)
  qed
  moreover have par f I
    using assms by auto
  ultimately show f = I
    using L.is-faithful by metis
qed

end
```

We now show that the unit ι of a monoidal category is unique up to a unique isomorphism (Proposition 2.2.6 of [2]).

```
locale monoidal-category-with-alternate-unit =
  monoidal-category C T α ι +
  C₁: monoidal-category C T α ι₁
for C :: 'a comp      (infixr · 55)
and T :: 'a * 'a ⇒ 'a
and α :: 'a * 'a * 'a ⇒ 'a
and ι :: 'a
and ι₁ :: 'a
```

begin

no-notation $C_1.tensor$ (**infixr** $\otimes$ *53*)
no-notation $C_1.unity$ ($\mathcal{I}$)
no-notation $C_1.lunit$ (l[-])
no-notation $C_1.runit$ (r[-])
no-notation $C_1.assoc$ (a[-, -, -])
no-notation $C_1.assoc'$ (a^{-1}[-, -, -])

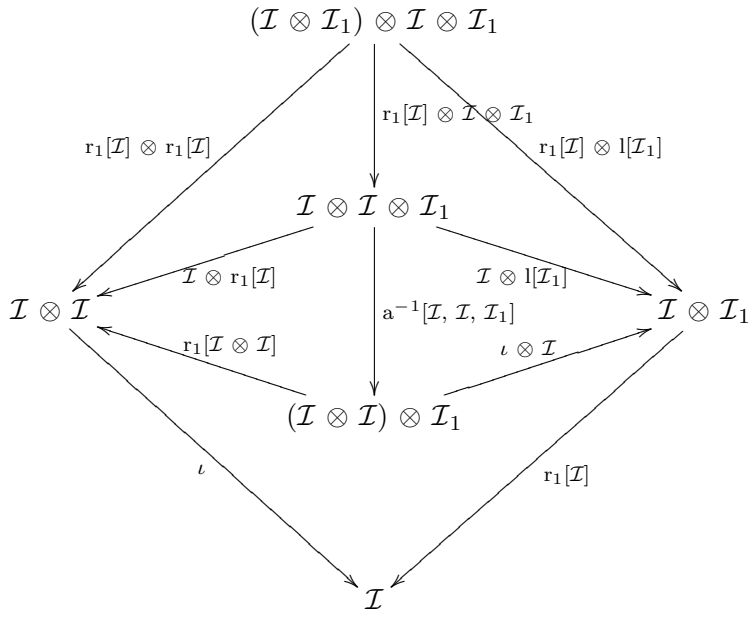
notation $C_1.tensor$ (**infixr** $\otimes_1$ *53*)
notation $C_1.unity$ ($\mathcal{I}_1$)
notation $C_1.lunit$ (l_1[-])
notation $C_1.runit$ (r_1[-])
notation $C_1.assoc$ (a_1[-, -, -])
notation $C_1.assoc'$ ($\mathrm{a}_1{}^{-1}$[-, -, -])

definition i
where $i \equiv \mathrm{l}[\mathcal{I}_1] \cdot inv\ \mathrm{r}_1[\mathcal{I}]$

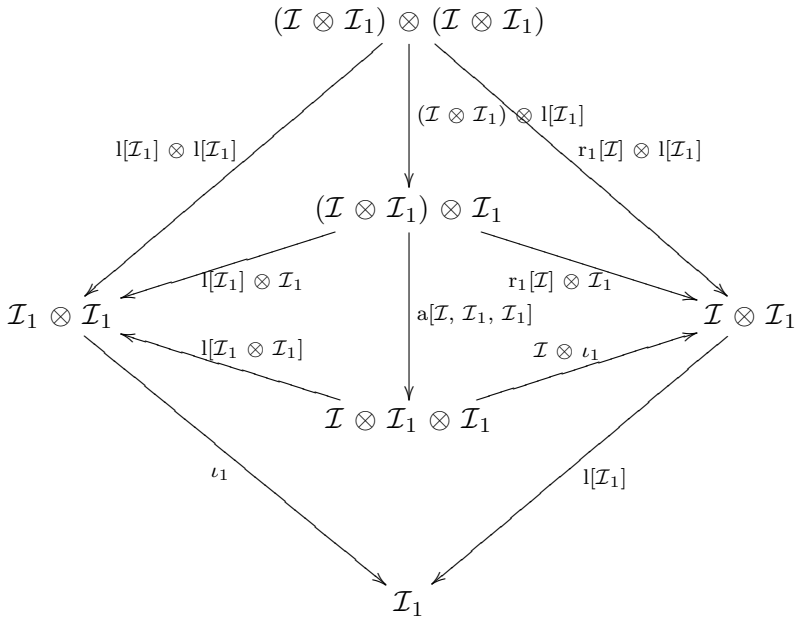
lemma *iso-i*:
shows «$i : \mathcal{I} \to \mathcal{I}_1$» **and** *iso i*
proof −
  show «$i : \mathcal{I} \to \mathcal{I}_1$»
    using $C_1.iso\text{-}runit$ *inv-in-hom i-def* **by** *auto*
  show *iso i*
    using *iso-lunit* $C_1.iso\text{-}runit$ *isos-compose i-def* **by** *simp*
qed

The following is Exercise 2.2.7 of [2].

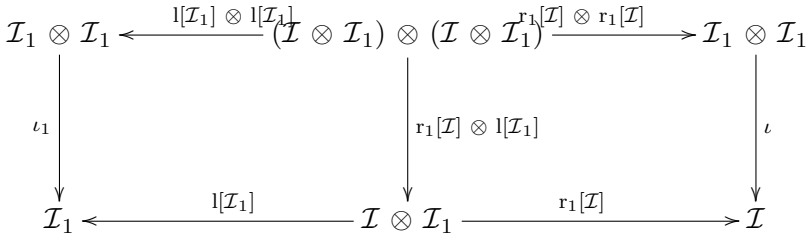
lemma *i-maps-ι-to-ι₁*:
shows $i \cdot \iota = \iota_1 \cdot (i \otimes i)$
proof −
  have *1*: $inv\ \mathrm{r}_1[\mathcal{I}] \cdot \iota = (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1]) \cdot (inv\ \mathrm{r}_1[\mathcal{I}] \otimes inv\ \mathrm{r}_1[\mathcal{I}])$
  proof −
    have $\iota \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{r}_1[\mathcal{I}]) = \mathrm{r}_1[\mathcal{I}] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1])$
    proof −



have $\iota \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{r}_1[\mathcal{I}]) = \iota \cdot (\mathcal{I} \otimes \mathrm{r}_1[\mathcal{I}]) \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using *interchange comp-cod-arr comp-arr-dom* **by** *simp*
also have $... = \iota \cdot (\mathrm{r}_1[\mathcal{I} \otimes \mathcal{I}] \cdot \mathrm{a}^{-1}[\mathcal{I}, \mathcal{I}, \mathcal{I}_1]) \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using C_1*.runit-tensor'* **by** *auto*
also have $... = (\iota \cdot \mathrm{r}_1[\mathcal{I} \otimes \mathcal{I}]) \cdot \mathrm{a}^{-1}[\mathcal{I}, \mathcal{I}, \mathcal{I}_1] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using *comp-assoc* **by** *auto*
also have $... = (\mathrm{r}_1[\mathcal{I}] \cdot (\iota \otimes \mathcal{I}_1)) \cdot \mathrm{a}^{-1}[\mathcal{I}, \mathcal{I}, \mathcal{I}_1] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using C_1*.runit-naturality* [*of* ι] *unit-in-hom* **by** *fastforce*
also have $... = \mathrm{r}_1[\mathcal{I}] \cdot ((\iota \otimes \mathcal{I}_1) \cdot \mathrm{a}^{-1}[\mathcal{I}, \mathcal{I}, \mathcal{I}_1]) \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using *comp-assoc* **by** *auto*
also have $... = \mathrm{r}_1[\mathcal{I}] \cdot (\mathcal{I} \otimes \mathrm{l}[\mathcal{I}_1]) \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I} \otimes \mathcal{I}_1)$
using *lunit-tensor lunit-commutes-with-L unitor-coincidence* **by** *simp*
also have $... = \mathrm{r}_1[\mathcal{I}] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1])$
using *interchange comp-arr-dom comp-cod-arr* **by** *simp*
finally show *?thesis* **by** *blast*
qed
moreover have *seq* ι $(\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{r}_1[\mathcal{I}])$ $\wedge$ *seq* $\mathrm{r}_1[\mathcal{I}]$ $(\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1])$
using *unit-in-hom* **by** *fastforce*
moreover have *iso* $\mathrm{r}_1[\mathcal{I}]$ $\wedge$ *iso* $(\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{r}_1[\mathcal{I}])$
using C_1*.iso-runit tensor-preserves-iso* **by** *force*
ultimately show *?thesis*
using *invert-opposite-sides-of-square inv-tensor* **by** *metis*
qed
have *2*: $\mathrm{l}[\mathcal{I}_1] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1]) = \iota_1 \cdot (\mathrm{l}[\mathcal{I}_1] \otimes \mathrm{l}[\mathcal{I}_1])$
proof −



have $\mathrm{l}[\mathcal{I}_1] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathrm{l}[\mathcal{I}_1]) = \mathrm{l}[\mathcal{I}_1] \cdot (\mathrm{r}_1[\mathcal{I}] \otimes \mathcal{I}_1) \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using *interchange comp-arr-dom comp-cod-arr* **by** *force*
also have $\ldots = \mathrm{l}[\mathcal{I}_1] \cdot ((\mathcal{I} \otimes \iota_1) \cdot \mathrm{a}[\mathcal{I}, \mathcal{I}_1, \mathcal{I}_1]) \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using C_1*.runit-tensor* C_1*.unitor-coincidence* C_1*.runit-commutes-with-R* **by** *simp*
also have $\ldots = (\mathrm{l}[\mathcal{I}_1] \cdot (\mathcal{I} \otimes \iota_1)) \cdot \mathrm{a}[\mathcal{I}, \mathcal{I}_1, \mathcal{I}_1] \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using *comp-assoc* **by** *fastforce*
also have $\ldots = (\iota_1 \cdot \mathrm{l}[\mathcal{I}_1 \otimes \mathcal{I}_1]) \cdot \mathrm{a}[\mathcal{I}, \mathcal{I}_1, \mathcal{I}_1] \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using *lunit-naturality* [*of* ι_1] C_1*.unit-in-hom lunit-commutes-with-L* **by** *fastforce*
also have $\ldots = \iota_1 \cdot (\mathrm{l}[\mathcal{I}_1 \otimes \mathcal{I}_1] \cdot \mathrm{a}[\mathcal{I}, \mathcal{I}_1, \mathcal{I}_1]) \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using *comp-assoc* **by** *force*
also have $\ldots = \iota_1 \cdot (\mathrm{l}[\mathcal{I}_1] \otimes \mathcal{I}_1) \cdot ((\mathcal{I} \otimes \mathcal{I}_1) \otimes \mathrm{l}[\mathcal{I}_1])$
using *lunit-tensor'* **by** *auto*
also have $\ldots = \iota_1 \cdot (\mathrm{l}[\mathcal{I}_1] \otimes \mathrm{l}[\mathcal{I}_1])$
using *interchange comp-arr-dom comp-cod-arr* **by** *simp*
finally show *?thesis* **by** *blast*
qed
show *?thesis*
proof −



have $i \cdot \iota = \mathrm{l}[\mathcal{I}_1] \cdot \mathit{inv}\ \mathrm{r}_1[\mathcal{I}] \cdot \iota$

```
    using i-def comp-assoc by auto
  also have ... = (l[I₁] · (r₁[I] ⊗ l[I₁])) · (inv r₁[I] ⊗ inv r₁[I])
    using 1 comp-assoc by simp
  also have ... = ι₁ · (l[I₁] ⊗ l[I₁]) · (inv r₁[I] ⊗ inv r₁[I])
    using 2 comp-assoc by fastforce
  also have ... = ι₁ · (i ⊗ i)
    using interchange i-def by simp
  finally show ?thesis by blast
 qed
qed

lemma inv-i-iso-ι:
assumes «f : I → I₁» and iso f and f · ι = ι₁ · (f ⊗ f)
shows «inv i · f : I → I» and iso (inv i · f)
and (inv i · f) · ι = ι · (inv i · f ⊗ inv i · f)
proof −
 show 1: «inv i · f : I → I»
   using assms iso-i inv-in-hom by blast
 show iso (inv i · f)
   using assms 1 iso-i inv-in-hom
   by (intro isos-compose, auto)
 show (inv i · f) · ι = ι · (inv i · f ⊗ inv i · f)
 proof −
   have (inv i · f) · ι = (inv i · ι₁) · (f ⊗ f)
     using assms iso-i comp-assoc by auto
   also have ... = (ι · (inv i ⊗ inv i)) · (f ⊗ f)
     by (metis unit-in-hom-ax i-maps-ι-to-ι₁ invert-opposite-sides-of-square iso-i
         inv-tensor tensor-preserves-iso seqI')
   also have ... = ι · (inv i · f ⊗ inv i · f)
     using assms 1 iso-i interchange comp-assoc by fastforce
   finally show ?thesis by blast
 qed
qed

lemma unit-unique-upto-unique-iso:
shows ∃!f. «f : I → I₁» ∧ iso f ∧ f · ι = ι₁ · (f ⊗ f)
proof
 show «i : I → I₁» ∧ iso i ∧ i · ι = ι₁ · (i ⊗ i)
   using iso-i i-maps-ι-to-ι₁ by auto
 show ⋀f. «f : I → I₁» ∧ iso f ∧ f · ι = ι₁ · (f ⊗ f) ⟹ f = i
 proof −
   fix f
   assume f: «f : I → I₁» ∧ iso f ∧ f · ι = ι₁ · (f ⊗ f)
   have inv i · f = I
     using f inv-i-iso-ι iso-commuting-with-unit-equals-unity by blast
   hence ide (C (inv i) f)
     using iso-i by simp
   thus f = i
     using section-retraction-of-iso(2) [of inv i f] inverse-arrow-unique inv-is-inverse
```

iso-i
by *blast*
qed
qed

end

2.2 Elementary Monoidal Category

Although the economy of data assumed by *monoidal-category* is useful for general results, to establish interpretations it is more convenient to work with a traditional definition of monoidal category. The following locale provides such a definition. It permits a monoidal category to be specified by giving the tensor product and the components of the associator and unitors, which are required only to satisfy elementary conditions that imply functoriality and naturality, without having to worry about extensionality or formal interpretations for the various functors and natural transformations.

locale *elementary-monoidal-category* =
category C
for $C :: 'a\ comp$ (**infixr** $\cdot$ *55*)
and $tensor :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\otimes$ *53*)
and $unity :: 'a$ ($\mathcal{I}$)
and $lunit :: 'a \Rightarrow 'a$ (l[-])
and $runit :: 'a \Rightarrow 'a$ (r[-])
and $assoc :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (a[-, -, -]) +
assumes *ide-unity* [*simp*]: $ide\ \mathcal{I}$
and *iso-lunit*: $ide\ a \Longrightarrow iso\ \mathrm{l}[a]$
and *iso-runit*: $ide\ a \Longrightarrow iso\ \mathrm{r}[a]$
and *iso-assoc*: $\llbracket ide\ a; ide\ b; ide\ c \rrbracket \Longrightarrow iso\ \mathrm{a}[a, b, c]$
and *tensor-in-hom* [*simp*]: $\llbracket \langle\!\langle f : a \to b \rangle\!\rangle; \langle\!\langle g : c \to d \rangle\!\rangle \rrbracket \Longrightarrow \langle\!\langle f \otimes g : a \otimes c \to b \otimes d \rangle\!\rangle$
and *tensor-preserves-ide*: $\llbracket ide\ a; ide\ b \rrbracket \Longrightarrow ide\ (a \otimes b)$
and *interchange*: $\llbracket seq\ g\ f; seq\ g'\ f' \rrbracket \Longrightarrow (g \otimes g') \cdot (f \otimes f') = g \cdot f \otimes g' \cdot f'$
and *lunit-in-hom* [*simp*]: $ide\ a \Longrightarrow \langle\!\langle \mathrm{l}[a] : \mathcal{I} \otimes a \to a \rangle\!\rangle$
and *lunit-naturality*: $arr\ f \Longrightarrow \mathrm{l}[cod\ f] \cdot (\mathcal{I} \otimes f) = f \cdot \mathrm{l}[dom\ f]$
and *runit-in-hom* [*simp*]: $ide\ a \Longrightarrow \langle\!\langle \mathrm{r}[a] : a \otimes \mathcal{I} \to a \rangle\!\rangle$
and *runit-naturality*: $arr\ f \Longrightarrow \mathrm{r}[cod\ f] \cdot (f \otimes \mathcal{I}) = f \cdot \mathrm{r}[dom\ f]$
and *assoc-in-hom* [*simp*]:
$\llbracket ide\ a; ide\ b; ide\ c \rrbracket \Longrightarrow \langle\!\langle \mathrm{a}[a, b, c] : (a \otimes b) \otimes c \to a \otimes b \otimes c \rangle\!\rangle$
and *assoc-naturality*:
$\llbracket arr\ f0; arr\ f1; arr\ f2 \rrbracket \Longrightarrow \mathrm{a}[cod\ f0, cod\ f1, cod\ f2] \cdot ((f0 \otimes f1) \otimes f2)$
$= (f0 \otimes (f1 \otimes f2)) \cdot \mathrm{a}[dom\ f0, dom\ f1, dom\ f2]$
and *triangle*: $\llbracket ide\ a; ide\ b \rrbracket \Longrightarrow (a \otimes \mathrm{l}[b]) \cdot \mathrm{a}[a, \mathcal{I}, b] = \mathrm{r}[a] \otimes b$
and *pentagon*: $\llbracket ide\ a; ide\ b; ide\ c; ide\ d \rrbracket \Longrightarrow$
$(a \otimes \mathrm{a}[b, c, d]) \cdot \mathrm{a}[a, b \otimes c, d] \cdot (\mathrm{a}[a, b, c] \otimes d)$
$= \mathrm{a}[a, b, c \otimes d] \cdot \mathrm{a}[a \otimes b, c, d]$

An interpretation for the *monoidal-category* locale readily induces an interpretation for the *elementary-monoidal-category* locale.

context *monoidal-category*

begin

lemma *induces-elementary-monoidal-category*:
shows *elementary-monoidal-category C tensor* $\mathcal{I}$ *lunit runit assoc*
using *iso-assoc tensor-preserves-ide assoc-in-hom tensor-in-hom*
assoc-naturality lunit-naturality runit-naturality lunit-in-hom runit-in-hom
iso-lunit iso-runit interchange pentagon triangle
by *unfold-locales auto*

end

context *elementary-monoidal-category*
begin

interpretation *CC*: *product-category C C* **..**
interpretation *CCC*: *product-category C CC.comp* **..**

definition $T :: 'a * 'a \Rightarrow 'a$
where $T\ f \equiv$ *if CC.arr f then* $(fst\ f \otimes snd\ f)$ *else null*

lemma *T-simp* [*simp*]:
assumes *arr f* **and** *arr g*
shows $T\ (f, g) = f \otimes g$
using *assms T-def* **by** *simp*

lemma *arr-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *arr* $(f \otimes g)$
using *assms tensor-in-hom* **by** *blast*

lemma *dom-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *dom* $(f \otimes g) =$ *dom f* $\otimes$ *dom g*
using *assms tensor-in-hom* **by** *blast*

lemma *cod-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *cod* $(f \otimes g) =$ *cod f* $\otimes$ *cod g*
using *assms tensor-in-hom* **by** *blast*

interpretation *T*: *binary-endofunctor C T*
using *interchange T-def*
apply *unfold-locales*
apply *auto*[4]
by (*elim CC.seqE*, *auto*)

lemma *binary-endofunctor-T*:
shows *binary-endofunctor C T* **..**

interpretation *ToTC*: *functor CCC.comp C T.ToTC*
using *T.functor-ToTC* **by** *auto*

interpretation *ToCT*: *functor CCC.comp C T.ToCT*
using *T.functor-ToCT* **by** *auto*

definition α
where $\alpha\ f \equiv$ *if CCC.arr f*
then (*fst f* $\otimes$ (*fst* (*snd f*)) $\otimes$ *snd* (*snd f*))) $\cdot$
a[*dom* (*fst f*), *dom* (*fst* (*snd f*)), *dom* (*snd* (*snd f*))]
else null

lemma α*-ide-simp* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\alpha\ (a,\ b,\ c) =$ a[a, b, c]
unfolding α*-def* **using** *assms assoc-in-hom comp-cod-arr*
by (*metis CC.arrI*$_{PC}$ *CCC.arrI*$_{PC}$ *fst-conv ide-char in-homE snd-conv*)

lemma *arr-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* a[a, b, c]
using *assms assoc-in-hom* **by** *blast*

lemma *dom-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *dom* a[a, b, c] = ($a \otimes b$) $\otimes$ c
using *assms assoc-in-hom* **by** *blast*

lemma *cod-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *cod* a[a, b, c] = $a \otimes b \otimes c$
using *assms assoc-in-hom* **by** *blast*

interpretation α: *natural-isomorphism CCC.comp C T.ToTC T.ToCT* α
proof −
interpret α: *transformation-by-components CCC.comp C T.ToTC T.ToCT* α
apply *unfold-locales*
unfolding α*-def T.ToTC-def T.ToCT-def T-def*
using *comp-arr-dom comp-cod-arr assoc-naturality*
by *simp-all*
interpret α: *natural-isomorphism CCC.comp C T.ToTC T.ToCT* α*.map*
using *iso-assoc* α*.map-simp-ide assoc-in-hom tensor-preserves-ide* α*-def*
by (*unfold-locales*, *auto*)
have $\alpha = \alpha$*.map*
using *assoc-naturality* α*-def comp-cod-arr T.ToTC-def T-def* α*.map-def* **by** *auto*
thus *natural-isomorphism CCC.comp C T.ToTC T.ToCT* α
using α*.natural-isomorphism-axioms* **by** *simp*
qed

interpretation α': *inverse-transformation CCC.comp C T.ToTC T.ToCT* α **..**

interpretation *L*: *functor C C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$›
  using *T.fixing-ide-gives-functor-1* **by** *auto*

interpretation *R*: *functor C C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$›
  using *T.fixing-ide-gives-functor-2* **by** *auto*

interpretation ι: *natural-isomorphism C C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$› *map*
    ‹$\lambda f.$ *if arr f then f* · l[*dom f*] *else null*›
proof −
  interpret ι: *transformation-by-components C C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$› *map* ‹$\lambda a.$ l[a]›
    using *lunit-naturality* **by** (*unfold-locales*, *auto*)
  interpret ι: *natural-isomorphism C C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$› *map* ι.*map*
    using *iso-lunit* **by** (*unfold-locales*, *simp*)
  have ι.*map* = ($\lambda f.$ *if arr f then f* · l[*dom f*] *else null*)
    using ι.*map-def lunit-naturality* **by** *fastforce*
  thus *natural-isomorphism C C* ($\lambda f.\ T\ (\mathcal{I},\ f)$) *map* ($\lambda f.$ *if arr f then f* · l[*dom f*] *else null*)
    using ι.*natural-isomorphism-axioms* **by** *force*
qed

interpretation ϱ: *natural-isomorphism C C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$› *map*
    ‹$\lambda f.$ *if arr f then f* · r[*dom f*] *else null*›
proof −
  interpret ϱ: *transformation-by-components C C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$› *map* ‹$\lambda a.$ r[a]›
    using *runit-naturality* **by** (*unfold-locales*, *auto*)
  interpret ϱ: *natural-isomorphism C C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$› *map* ϱ.*map*
    using *iso-runit* ϱ.*map-simp-ide* **by** (*unfold-locales*, *simp*)
  have ($\lambda f.$ *if arr f then f* · r[*dom f*] *else null*) = ϱ.*map*
    using ϱ.*map-def runit-naturality T-simp* **by** *fastforce*
  thus *natural-isomorphism C C* ($\lambda f.\ T\ (f,\ \mathcal{I})$) *map* ($\lambda f.$ *if arr f then f* · r[*dom f*] *else null*)
    using ϱ.*natural-isomorphism-axioms* **by** *force*
qed

The endofunctors $\lambda f.\ T\ (\mathcal{I},\ f)$ and $\lambda f.\ T\ (f,\ \mathcal{I})$ are equivalence functors, due to the existence of the unitors.

interpretation *L*: *equivalence-functor C C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$›
proof −
  interpret *endofunctor C* ‹$\lambda f.\ T\ (\mathcal{I},\ f)$› **..**
  show *equivalence-functor C C* ($\lambda f.\ T\ (\mathcal{I},\ f)$)
    using *isomorphic-to-identity-is-equivalence* ι.*natural-isomorphism-axioms* **by** *simp*
qed

interpretation *R*: *equivalence-functor C C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$›
proof −
  interpret *endofunctor C* ‹$\lambda f.\ T\ (f,\ \mathcal{I})$› **..**
  show *equivalence-functor C C* ($\lambda f.\ T\ (f,\ \mathcal{I})$)
    using *isomorphic-to-identity-is-equivalence* ϱ.*natural-isomorphism-axioms* **by** *simp*
qed

To complete an interpretation of the *monoidal-category* locale, we define $\iota \equiv \mathrm{l}[\mathcal{I}]$. We could also have chosen $\iota \equiv \varrho\,[\mathcal{I}]$ as the two are equal, though to prove that requires some work yet.

```
definition ι
where ι ≡ l[ℐ]

lemma ι-in-hom:
shows «ι : ℐ ⊗ ℐ → ℐ»
  using lunit-in-hom ι-def by simp

lemma induces-monoidal-category:
shows monoidal-category C T α ι
proof −
  have 1: «ι : ℐ ⊗ ℐ → ℐ»
    using lunit-in-hom ι-def by simp
  interpret L: equivalence-functor C C ‹λf. T (cod ι, f)›
  proof −
    have (λf. T (ℐ, f)) = (λf. T (cod ι, f)) using 1 by fastforce
    thus equivalence-functor C C (λf. T (cod ι, f))
      using L.equivalence-functor-axioms T-def by simp
  qed
  interpret R: equivalence-functor C C ‹λf. T (f, cod ι)›
  proof −
    have (λf. T (f, ℐ)) = (λf. T (f, cod ι)) using 1 by fastforce
    thus equivalence-functor C C (λf. T (f, cod ι))
      using R.equivalence-functor-axioms T-def by simp
  qed
  show ?thesis
  proof
    show «ι : T (cod ι, cod ι) → cod ι» using 1 by fastforce
    show iso ι using iso-lunit ι-def by simp
    show ⋀a b c d. ⟦ ide a; ide b; ide c; ide d ⟧ ⟹
                   T (a, α (b, c, d)) · α (a, T (b, c), d) · T (α (a, b, c), d)
                     = α (a, b, T (c, d)) · α (T (a, b), c, d)
      using pentagon tensor-preserves-ide by simp
  qed
qed

interpretation MC: monoidal-category C T α ι
  using induces-monoidal-category by auto
```

We now show that the notions defined in the interpretation *MC* agree with their counterparts in the present locale. These facts are needed if we define an interpretation for the *elementary-monoidal-category* locale, use it to obtain the induced interpretation for *monoidal-category*, and then want to transfer facts obtained in the induced interpretation back to the original one.

```
lemma ℐ-agreement:
shows MC.unity = ℐ
```

by (*metis ι-def ide-unity in-homE lunit-in-hom*)

lemma *L-agreement*:
shows $MC.L = (\lambda f.\ T\ (\mathcal{I}, f))$
using *ι-in-hom* **by** *auto*

lemma *R-agreement*:
shows $MC.R = (\lambda f.\ T\ (f, \mathcal{I}))$
using *ι-in-hom* **by** *auto*

We wish to show that the components of the unitors $MC.\mathfrak{l}$ and $MC.\varrho$ defined in the induced interpretation MC agree with those given by the parameters *lunit* and *runit* to the present locale. To avoid a lengthy development that repeats work already done in the *monoidal-category* locale, we establish the agreement in a special case and then use the properties already shown for MC to prove the general case. In particular, we first show that $\mathrm{l}[\mathcal{I}] = MC.lunit\ MC.unity$ and $\mathrm{r}[\mathcal{I}] = MC.runit\ MC.unity$, from which it follows by facts already proved for MC that both are equal to ι. We then show that for an arbitrary identity a the arrows $\mathrm{l}[a]$ and $\mathrm{r}[a]$ satisfy the equations that uniquely characterize the components $MC.lunit\ a$ and $MC.runit\ a$, respectively, and are therefore equal to those components.

lemma *unitor-coincidence*:
shows $\mathrm{l}[\mathcal{I}] = \iota$ **and** $\mathrm{r}[\mathcal{I}] = \iota$
proof −
have $\mathrm{r}[\mathcal{I}] = MC.runit\ MC.unity$
by (*metis* (*no-types*, *lifting*) *MC.arr-runit MC.runit-eqI MC.unitor-coincidence*(*2*)
T-simp $\mathcal{I}$*-agreement ι-def α-ide-simp ideD*(*1*) *ide-unity iso-is-arr iso-runit runit-in-hom triangle*)
moreover have $\mathrm{l}[\mathcal{I}] = MC.lunit\ MC.unity$
using *MC.unitor-coincidence*(*1*) *ι-def* **by** *force*
ultimately have *1*: $\mathrm{l}[\mathcal{I}] = \iota \wedge \mathrm{r}[\mathcal{I}] = \iota$
using *MC.unitor-coincidence* **by** *simp*
show $\mathrm{l}[\mathcal{I}] = \iota$ **using** *1* **by** *simp*
show $\mathrm{r}[\mathcal{I}] = \iota$ **using** *1* **by** *simp*
qed

lemma *lunit-char*:
assumes *ide a*
shows $\mathcal{I} \otimes \mathrm{l}[a] = (\iota \otimes a) \cdot inv\ \mathrm{a}[\mathcal{I}, \mathcal{I}, a]$
by (*metis MC.iso-assoc α-ide-simp ι-in-hom arrI arr-tensor assms ideD*(*1*)
ide-unity invert-side-of-triangle(*2*) *triangle unitor-coincidence*(*2*))

lemma *runit-char*:
assumes *ide a*
shows $\mathrm{r}[a] \otimes \mathcal{I} = (a \otimes \iota) \cdot \mathrm{a}[a, \mathcal{I}, \mathcal{I}]$
using *assms triangle ι-def* **by** *simp*

lemma $\mathfrak{l}$*-agreement*:
shows $MC.\mathfrak{l} = (\lambda f.\ if\ arr\ f\ then\ f \cdot \mathrm{l}[dom\ f]\ else\ null)$

```
  proof
    fix f
    have ¬ arr f ⟹ MC.𝔩 f = null by simp
    moreover have arr f ⟹ MC.𝔩 f = f · l[dom f]
    proof −
      have ⋀a. ide a ⟹ l[a] = MC.lunit a
        using 𝓘-agreement T-def lunit-char ι-in-hom iso-lunit
        apply (intro MC.lunit-eqI)
         apply auto
        by blast
      thus ?thesis
        by (metis ide-dom ext seqE)
    qed
    ultimately show MC.𝔩 f = (if arr f then f · l[dom f] else null) by simp
  qed

  lemma ϱ-agreement:
  shows MC.ϱ = (λf. if arr f then f · r[dom f] else null)
  proof
    fix f
    have ¬ arr f ⟹ MC.ϱ f = null by simp
    moreover have arr f ⟹ MC.ϱ f = f · r[dom f]
    proof −
      have ⋀a. ide a ⟹ r[a] = MC.runit a
        using 𝓘-agreement T-def runit-char ι-in-hom iso-runit
        apply (intro MC.runit-eqI)
         apply auto
        by blast
      thus ?thesis
        by (metis ide-dom local.ext seqE)
    qed
    ultimately show MC.ϱ f = (if arr f then f · r[dom f] else null) by simp
  qed

  lemma lunit-agreement:
  assumes ide a
  shows MC.lunit a = l[a]
    using assms comp-cod-arr 𝔩-agreement
    by (metis (no-types, lifting) MC.𝔩-ide-simp ide-char in-homE lunit-in-hom)

  lemma runit-agreement:
  assumes ide a
  shows MC.runit a = r[a]
    using assms comp-cod-arr ϱ-agreement
    by (metis (no-types, lifting) MC.ϱ-ide-simp ide-char in-homE runit-in-hom)

end
```

2.3 Strict Monoidal Category

A monoidal category is *strict* if the components of the associator and unitors are all identities.

locale *strict-monoidal-category* =
  monoidal-category +
assumes *strict-assoc*: ⟦ *ide a0*; *ide a1*; *ide a2* ⟧ ⟹ *ide* a[*a0*, *a1*, *a2*]
and *strict-lunit*: *ide a* ⟹ l[*a*] = *a*
and *strict-runit*: *ide a* ⟹ r[*a*] = *a*
begin

  lemma *strict-unit*:
  shows *ι* = *I*
    using *strict-lunit unitor-coincidence*(*1*) **by** *auto*

  lemma *tensor-assoc* [*simp*]:
  assumes *arr f0* **and** *arr f1* **and** *arr f2*
  shows (*f0* ⊗ *f1*) ⊗ *f2* = *f0* ⊗ *f1* ⊗ *f2*
    by (*metis CC.arrI*$_{PC}$ *CCC.arrI*$_{PC}$ *α'.preserves-reflects-arr α'-simp assms*(*1*−*3*)
        assoc'-naturality ide-cod ide-dom inv-ide comp-arr-ide comp-ide-arr strict-assoc)

end

2.4 Opposite Monoidal Category

The *opposite* of a monoidal category has the same underlying category, but the arguments to the tensor product are reversed and the associator is inverted and its arguments reversed.

locale *opposite-monoidal-category* =
  C: *monoidal-category C T*$_C$ *α*$_C$ *ι*
for *C* :: *'a comp* (**infixr** · *55*)
and *T*$_C$ :: *'a* ∗ *'a* ⇒ *'a*
and *α*$_C$ :: *'a* ∗ *'a* ∗ *'a* ⇒ *'a*
and *ι* :: *'a*
begin

  abbreviation *T*
  where *T f* ≡ *T*$_C$ (*snd f*, *fst f*)

  abbreviation *α*
  where *α f* ≡ *C.α'* (*snd* (*snd f*), *fst* (*snd f*), *fst f*)

end

sublocale *opposite-monoidal-category* ⊆ *monoidal-category C T α ι*
proof −
  interpret *T*: *binary-endofunctor C T*

```
    using C.T.is-extensional C.CC.seq-char C.interchange by (unfold-locales, auto)
  interpret ToTC: functor C.CCC.comp C T.ToTC
    using T.functor-ToTC by auto
  interpret ToCT: functor C.CCC.comp C T.ToCT
    using T.functor-ToCT by auto
  interpret α: natural-transformation C.CCC.comp C T.ToTC T.ToCT α
    using C.α'.is-extensional C.CCC.dom-char C.CCC.cod-char T.ToTC-def T.ToCT-def
          C.α'-simp C.α'.naturality
    by (unfold-locales) auto
  interpret α: natural-isomorphism C.CCC.comp C T.ToTC T.ToCT α
    using C.α'.components-are-iso by (unfold-locales, simp)
  interpret L: equivalence-functor C C ‹λf. T (C.cod ι, f)›
    using C.R.equivalence-functor-axioms by simp
  interpret R: equivalence-functor C C ‹λf. T (f, C.cod ι)›
    using C.L.equivalence-functor-axioms by simp
  show monoidal-category C T α ι
    using C.unit-is-iso C.pentagon' C.comp-assoc
    by (unfold-locales) auto
qed

context opposite-monoidal-category
begin

  lemma lunit-simp:
  assumes C.ide a
  shows lunit a = C.runit a
    using assms lunit-char C.iso-assoc by (intro C.runit-eqI, auto)

  lemma runit-simp:
  assumes C.ide a
  shows runit a = C.lunit a
    using assms runit-char C.iso-assoc by (intro C.lunit-eqI, auto)

end
```

2.5 Monoidal Language

In this section we assume that a category C is given, and we define a formal syntax of terms constructed from arrows of C using function symbols that correspond to unity, composition, tensor, the associator and its formal inverse, and the left and right unitors and their formal inverses. We will use this syntax to state and prove the coherence theorem and then to construct the free monoidal category generated by C.

```
locale monoidal-language =
  C: category C
  for C :: 'a comp                    (infixr · 55)
begin

  datatype (discs-sels) 't term =
```

```
  Prim 't                          (⟨-⟩)
| Unity                            (𝓘)
| Tensor 't term 't term           (infixr ⊗ 53)
| Comp 't term 't term             (infixr ⋅ 55)
| Lunit 't term                    (𝗹[-])
| Lunit' 't term                   (𝗹⁻¹[-])
| Runit 't term                    (𝗿[-])
| Runit' 't term                   (𝗿⁻¹[-])
| Assoc 't term 't term 't term    (𝗮[-, -, -])
| Assoc' 't term 't term 't term   (𝗮⁻¹[-, -, -])

lemma not-is-Tensor-Unity:
shows ¬ is-Tensor Unity
  by simp
```

We define formal domain and codomain functions on terms.

```
primrec Dom :: 'a term ⇒ 'a term
where Dom ⟨f⟩ = ⟨C.dom f⟩
    | Dom 𝓘 = 𝓘
    | Dom (t ⊗ u) = (Dom t ⊗ Dom u)
    | Dom (t ⋅ u) = Dom u
    | Dom 𝗹[t] = (𝓘 ⊗ Dom t)
    | Dom 𝗹⁻¹[t] = Dom t
    | Dom 𝗿[t] = (Dom t ⊗ 𝓘)
    | Dom 𝗿⁻¹[t] = Dom t
    | Dom 𝗮[t, u, v] = ((Dom t ⊗ Dom u) ⊗ Dom v)
    | Dom 𝗮⁻¹[t, u, v] = (Dom t ⊗ (Dom u ⊗ Dom v))

primrec Cod :: 'a term ⇒ 'a term
where Cod ⟨f⟩ = ⟨C.cod f⟩
    | Cod 𝓘 = 𝓘
    | Cod (t ⊗ u) = (Cod t ⊗ Cod u)
    | Cod (t ⋅ u) = Cod t
    | Cod 𝗹[t] = Cod t
    | Cod 𝗹⁻¹[t] = (𝓘 ⊗ Cod t)
    | Cod 𝗿[t] = Cod t
    | Cod 𝗿⁻¹[t] = (Cod t ⊗ 𝓘)
    | Cod 𝗮[t, u, v] = (Cod t ⊗ (Cod u ⊗ Cod v))
    | Cod 𝗮⁻¹[t, u, v] = ((Cod t ⊗ Cod u) ⊗ Cod v)
```

A term is a "formal arrow" if it is constructed from arrows of C in such a way that composition is applied only to formally composable pairs of terms.

```
primrec Arr :: 'a term ⇒ bool
where Arr ⟨f⟩ = C.arr f
    | Arr 𝓘 = True
    | Arr (t ⊗ u) = (Arr t ∧ Arr u)
    | Arr (t ⋅ u) = (Arr t ∧ Arr u ∧ Dom t = Cod u)
    | Arr 𝗹[t] = Arr t
    | Arr 𝗹⁻¹[t] = Arr t
```

| *Arr* $\mathbf{r}[t] = Arr\ t$
| *Arr* $\mathbf{r}^{-1}[t] = Arr\ t$
| *Arr* $\mathbf{a}[t, u, v] = (Arr\ t \wedge Arr\ u \wedge Arr\ v)$
| *Arr* $\mathbf{a}^{-1}[t, u, v] = (Arr\ t \wedge Arr\ u \wedge Arr\ v)$

abbreviation *Par* :: *'a term* ⇒ *'a term* ⇒ *bool*
where $Par\ t\ u \equiv Arr\ t \wedge Arr\ u \wedge Dom\ t = Dom\ u \wedge Cod\ t = Cod\ u$

abbreviation *Seq* :: *'a term* ⇒ *'a term* ⇒ *bool*
where $Seq\ t\ u \equiv Arr\ t \wedge Arr\ u \wedge Dom\ t = Cod\ u$

abbreviation *Hom* :: *'a term* ⇒ *'a term* ⇒ *'a term set*
where $Hom\ a\ b \equiv \{\ t.\ Arr\ t \wedge Dom\ t = a \wedge Cod\ t = b\ \}$

A term is a "formal identity" if it is constructed from identity arrows of C and $\boldsymbol{\mathcal{I}}$ using only the $\otimes$ operator.

primrec *Ide* :: *'a term* ⇒ *bool*
where $Ide\ \langle f \rangle = C.ide\ f$
| $Ide\ \boldsymbol{\mathcal{I}} = True$
| $Ide\ (t \otimes u) = (Ide\ t \wedge Ide\ u)$
| $Ide\ (t \cdot u) = False$
| $Ide\ \mathbf{l}[t] = False$
| $Ide\ \mathbf{l}^{-1}[t] = False$
| $Ide\ \mathbf{r}[t] = False$
| $Ide\ \mathbf{r}^{-1}[t] = False$
| $Ide\ \mathbf{a}[t, u, v] = False$
| $Ide\ \mathbf{a}^{-1}[t, u, v] = False$

lemma *Ide-implies-Arr* [*simp*]:
shows $Ide\ t \Longrightarrow Arr\ t$
by (*induct t*) *auto*

lemma *Arr-implies-Ide-Dom*:
shows $Arr\ t \Longrightarrow Ide\ (Dom\ t)$
by (*induct t*) *auto*

lemma *Arr-implies-Ide-Cod*:
shows $Arr\ t \Longrightarrow Ide\ (Cod\ t)$
by (*induct t*) *auto*

lemma *Ide-in-Hom* [*simp*]:
shows $Ide\ t \Longrightarrow t \in Hom\ t\ t$
by (*induct t*) *auto*

A formal arrow is "canonical" if the only arrows of C used in its construction are identities.

primrec *Can* :: *'a term* ⇒ *bool*
where $Can\ \langle f \rangle = C.ide\ f$
| $Can\ \boldsymbol{\mathcal{I}} = True$

```
  | Can (t ⊗ u) = (Can t ∧ Can u)
  | Can (t ⋆ u) = (Can t ∧ Can u ∧ Dom t = Cod u)
  | Can l[t] = Can t
  | Can l⁻¹[t] = Can t
  | Can r[t] = Can t
  | Can r⁻¹[t] = Can t
  | Can a[t, u, v] = (Can t ∧ Can u ∧ Can v)
  | Can a⁻¹[t, u, v] = (Can t ∧ Can u ∧ Can v)

lemma Ide-implies-Can:
shows Ide t ⟹ Can t
  by (induct t) auto

lemma Can-implies-Arr:
shows Can t ⟹ Arr t
  by (induct t) auto
```

We next define the formal inverse of a term. This is only sensible for formal arrows built using only isomorphisms of C; in particular, for canonical formal arrows.

```
primrec Inv :: 'a term ⇒ 'a term
where Inv ⟨f⟩ = ⟨C.inv f⟩
  | Inv 𝓘 = 𝓘
  | Inv (t ⊗ u) = (Inv t ⊗ Inv u)
  | Inv (t ⋆ u) = (Inv u ⋆ Inv t)
  | Inv l[t] = l⁻¹[Inv t]
  | Inv l⁻¹[t] = l[Inv t]
  | Inv r[t] = r⁻¹[Inv t]
  | Inv r⁻¹[t] = r[Inv t]
  | Inv a[t, u, v] = a⁻¹[Inv t, Inv u, Inv v]
  | Inv a⁻¹[t, u, v] = a[Inv t, Inv u, Inv v]

lemma Inv-preserves-Ide:
shows Ide t ⟹ Ide (Inv t)
  by (induct t) auto

lemma Inv-preserves-Can:
assumes Can t
shows Can (Inv t) and Dom (Inv t) = Cod t and Cod (Inv t) = Dom t
proof −
  have 0: Can t ⟹ Can (Inv t) ∧ Dom (Inv t) = Cod t ∧ Cod (Inv t) = Dom t
    by (induct t) auto
  show Can (Inv t) using assms 0 by blast
  show Dom (Inv t) = Cod t using assms 0 by blast
  show Cod (Inv t) = Dom t using assms 0 by blast
qed

lemma Inv-in-Hom [simp]:
assumes Can t
shows Inv t ∈ Hom (Cod t) (Dom t)
```

using *assms Inv-preserves-Can Can-implies-Arr* **by** *simp*

lemma *Inv-Ide* [*simp*]:
assumes *Ide a*
shows *Inv a* = *a*
using *assms* **by** (*induct a*) *auto*

lemma *Inv-Inv* [*simp*]:
assumes *Can t*
shows *Inv* (*Inv t*) = *t*
using *assms* **by** (*induct t*) *auto*

We call a term "diagonal" if it is either $\boldsymbol{\mathcal{I}}$ or it is constructed from arrows of C using only the $\otimes$ operator associated to the right. Essentially, such terms are lists of arrows of C, where $\boldsymbol{\mathcal{I}}$ represents the empty list and $\otimes$ is used as the list constructor. We call them "diagonal" because terms can regarded as defining "interconnection matrices" of arrows connecting "inputs" to "outputs", and from this point of view diagonal terms correspond to diagonal matrices. The matrix point of view is suggestive for the extension of the results presented here to the symmetric monoidal and cartesian monoidal cases.

fun *Diag* :: *'a term* ⇒ *bool*
where *Diag* $\boldsymbol{\mathcal{I}}$ = *True*
| *Diag* ⟨*f*⟩ = *C.arr f*
| *Diag* (⟨*f*⟩ ⊗ *u*) = (*C.arr f* ∧ *Diag u* ∧ *u* ≠ $\boldsymbol{\mathcal{I}}$)
| *Diag* - = *False*

lemma *Diag-TensorE*:
assumes *Diag* (*Tensor t u*)
shows ⟨*un-Prim t*⟩ = *t* **and** *C.arr* (*un-Prim t*) **and** *Diag t* **and** *Diag u* **and** *u* ≠ $\boldsymbol{\mathcal{I}}$
proof −
have *1*: *t* = ⟨*un-Prim t*⟩ ∧ *C.arr* (*un-Prim t*) ∧ *Diag t* ∧ *Diag u* ∧ *u* ≠ $\boldsymbol{\mathcal{I}}$
using *assms* **by** (*cases t*; *simp*; *cases u*; *simp*)
show ⟨*un-Prim t*⟩ = *t* **using** *1* **by** *simp*
show *C.arr* (*un-Prim t*) **using** *1* **by** *simp*
show *Diag t* **using** *1* **by** *simp*
show *Diag u* **using** *1* **by** *simp*
show *u* ≠ $\boldsymbol{\mathcal{I}}$ **using** *1* **by** *simp*
qed

lemma *Diag-implies-Arr*:
shows *Diag t* ⟹ *Arr t*
apply (*induct t*, *simp-all*)
by (*simp add*: *Diag-TensorE*)

lemma *Dom-preserves-Diag*:
shows *Diag t* ⟹ *Diag* (*Dom t*)
apply (*induct t*, *simp-all*)
proof −
fix *u v*
assume *I2*: *Diag v* ⟹ *Diag* (*Dom v*)

```
  assume uv: Diag (u ⊗ v)
  show Diag (Dom u ⊗ Dom v)
  proof −
    have 1: is-Prim (Dom u) ∧ C.arr (un-Prim (Dom u)) ∧
          Dom u = ⟨C.dom (un-Prim u)⟩
      using uv by (cases u; simp; cases v, simp-all)
    have 2: Diag v ∧ v ≠ 𝓘 ∧ ¬ is-Comp v ∧ ¬ is-Lunit' v ∧ ¬ is-Runit' v
      using uv by (cases u; simp; cases v, simp-all)
    have Diag (Dom v) ∧ Dom v ≠ 𝓘
      using 2 I2 by (cases v, simp-all)
    thus ?thesis using 1 by force
  qed
qed

lemma Cod-preserves-Diag:
shows Diag t ⟹ Diag (Cod t)
apply (induct t, simp-all)
proof −
  fix u v
  assume I2: Diag v ⟹ Diag (Cod v)
  assume uv: Diag (u ⊗ v)
  show Diag (Cod u ⊗ Cod v)
  proof −
    have 1: is-Prim (Cod u) ∧ C.arr (un-Prim (Cod u)) ∧ Cod u = ⟨C.cod (un-Prim u)⟩
      using uv by (cases u; simp; cases v; simp)
    have 2: Diag v ∧ v ≠ 𝓘 ∧ ¬ is-Comp v ∧ ¬ is-Lunit' v ∧ ¬ is-Runit' v
      using uv by (cases u; simp; cases v; simp)
    have Diag (Cod v) ∧ Cod v ≠ 𝓘
      using I2 2 by (cases v, simp-all)
    thus ?thesis using 1 by force
  qed
qed

lemma Inv-preserves-Diag:
assumes Can t and Diag t
shows Diag (Inv t)
proof −
  have Can t ∧ Diag t ⟹ Diag (Inv t)
    apply (induct t, simp-all)
    by (metis (no-types, lifting) Can.simps(1) Inv.simps(1) Inv.simps(2) Diag.simps(3)
        Inv-Inv Diag-TensorE(1) C.inv-ide)
  thus ?thesis using assms by blast
qed
```

The following function defines the "dimension" of a term, which is the number of arrows of $(\cdot)$ it contains. For diagonal terms, this is just the length of the term when regarded as a list of arrows of $(\cdot)$. Alternatively, if a term is regarded as defining an interconnection matrix, then the dimension is the number of inputs (or outputs).

```
primrec dim :: 'a term ⇒ nat
```

where *dim* ⟨*f*⟩ = *1*
| *dim* $\mathcal{I}$ = *0*
| *dim* (*t* ⊗ *u*) = (*dim t* + *dim u*)
| *dim* (*t* ⋅ *u*) = *dim t*
| *dim* **l**[*t*] = *dim t*
| *dim* $\mathbf{l}^{-1}$[*t*] = *dim t*
| *dim* **r**[*t*] = *dim t*
| *dim* $\mathbf{r}^{-1}$[*t*] = *dim t*
| *dim* **a**[*t*, *u*, *v*] = *dim t* + *dim u* + *dim v*
| *dim* $\mathbf{a}^{-1}$[*t*, *u*, *v*] = *dim t* + *dim u* + *dim v*

The following function defines a tensor product for diagonal terms. If terms are regarded as lists, this is just list concatenation. If terms are regarded as matrices, this corresponds to constructing a block diagonal matrix.

fun *TensorDiag* (**infixr** ⌊⊗⌋ *53*)
where $\mathcal{I}$ ⌊⊗⌋ *u* = *u*
| *t* ⌊⊗⌋ $\mathcal{I}$ = *t*
| ⟨*f*⟩ ⌊⊗⌋ *u* = ⟨*f*⟩ ⊗ *u*
| (*t* ⊗ *u*) ⌊⊗⌋ *v* = *t* ⌊⊗⌋ (*u* ⌊⊗⌋ *v*)
| *t* ⌊⊗⌋ *u* = *undefined*

lemma *TensorDiag-Prim* [*simp*]:
assumes *t* ≠ $\mathcal{I}$
shows ⟨*f*⟩ ⌊⊗⌋ *t* = ⟨*f*⟩ ⊗ *t*
using *assms* **by** (*cases t*, *simp-all*)

lemma *TensorDiag-term-Unity* [*simp*]:
shows *t* ⌊⊗⌋ $\mathcal{I}$ = *t*
by (*cases t* = $\mathcal{I}$; *cases t*, *simp-all*)

lemma *TensorDiag-Diag*:
assumes *Diag* (*t* ⊗ *u*)
shows *t* ⌊⊗⌋ *u* = *t* ⊗ *u*
using *assms TensorDiag-Prim* **by** (*cases t*, *simp-all*)

lemma *TensorDiag-preserves-Diag*:
assumes *Diag t* **and** *Diag u*
shows *Diag* (*t* ⌊⊗⌋ *u*)
and *Dom* (*t* ⌊⊗⌋ *u*) = *Dom t* ⌊⊗⌋ *Dom u*
and *Cod* (*t* ⌊⊗⌋ *u*) = *Cod t* ⌊⊗⌋ *Cod u*
proof −
have *0*: ⋀*u*. ⟦ *Diag t*; *Diag u* ⟧ ⟹
Diag (*t* ⌊⊗⌋ *u*) ∧ *Dom* (*t* ⌊⊗⌋ *u*) = *Dom t* ⌊⊗⌋ *Dom u* ∧
Cod (*t* ⌊⊗⌋ *u*) = *Cod t* ⌊⊗⌋ *Cod u*
apply (*induct t*, *simp-all*)
proof −
fix *f* :: *'a* **and** *u* :: *'a term*
assume *f*: *C.arr f*
assume *u*: *Diag u*

```
    show Diag (⟨f⟩ ⌊⊗⌋ u) ∧ Dom (⟨f⟩ ⌊⊗⌋ u) = ⟨C.dom f⟩ ⌊⊗⌋ Dom u ∧
                          Cod (⟨f⟩ ⌊⊗⌋ u) = ⟨C.cod f⟩ ⌊⊗⌋ Cod u
      using u f by (cases u, simp-all)
    next
    fix u v w
    assume I1: ⋀u. Diag v ⟹ Diag u ⟹ Diag (v ⌊⊗⌋ u) ∧
                                      Dom (v ⌊⊗⌋ u) = Dom v ⌊⊗⌋ Dom u ∧
                                      Cod (v ⌊⊗⌋ u) = Cod v ⌊⊗⌋ Cod u
    assume I2: ⋀u. Diag w ⟹  Diag u ⟹ Diag (w ⌊⊗⌋ u) ∧
                                       Dom (w ⌊⊗⌋ u) = Dom w ⌊⊗⌋ Dom u ∧
                                       Cod (w ⌊⊗⌋ u) = Cod w ⌊⊗⌋ Cod u
    assume vw: Diag (v ⊗ w)
    assume u: Diag u
    show Diag ((v ⊗ w) ⌊⊗⌋ u) ∧
         Dom ((v ⊗ w) ⌊⊗⌋ u) = (Dom v ⊗ Dom w) ⌊⊗⌋ Dom u ∧
         Cod ((v ⊗ w) ⌊⊗⌋ u) = (Cod v ⊗ Cod w) ⌊⊗⌋ Cod u
    proof −
      have v: v = ⟨un-Prim v⟩ ∧ Diag v
        using vw Diag-implies-Arr Diag-TensorE [of v w] by force
      have w: Diag w
        using vw by (simp add: Diag-TensorE)
      have u = 𝓘 ⟹ ?thesis by (simp add: vw)
      moreover have u ≠ 𝓘 ⟹ ?thesis
        using u v w I1 I2 Dom-preserves-Diag [of u] Cod-preserves-Diag [of u]
        by (cases u, simp-all)
      ultimately show ?thesis by blast
    qed
  qed
  show Diag (t ⌊⊗⌋ u)  using assms 0 by blast
  show Dom (t ⌊⊗⌋ u) = Dom t ⌊⊗⌋ Dom u using assms 0 by blast
  show Cod (t ⌊⊗⌋ u) = Cod t ⌊⊗⌋ Cod u using assms 0 by blast
qed

lemma TensorDiag-in-Hom:
assumes Diag t and Diag u
shows t ⌊⊗⌋ u ∈ Hom (Dom t ⌊⊗⌋ Dom u) (Cod t ⌊⊗⌋ Cod u)
  using assms TensorDiag-preserves-Diag Diag-implies-Arr by simp

lemma Dom-TensorDiag:
assumes Diag t and Diag u
shows Dom (t ⌊⊗⌋ u) = Dom t ⌊⊗⌋ Dom u
  using assms TensorDiag-preserves-Diag(2) by simp

lemma Cod-TensorDiag:
assumes Diag t and Diag u
shows Cod (t ⌊⊗⌋ u) = Cod t ⌊⊗⌋ Cod u
  using assms TensorDiag-preserves-Diag(3) by simp

lemma not-is-Tensor-TensorDiagE:
```

assumes ¬ *is-Tensor* (*t* ⌊⊗⌋ *u*) **and** *Diag t* **and** *Diag u*
and $t \neq \boldsymbol{\mathcal{I}}$ **and** $u \neq \boldsymbol{\mathcal{I}}$
shows *False*
proof −
have ⟦ ¬ *is-Tensor* (*t* ⌊⊗⌋ *u*); *Diag t*; *Diag u*; $t \neq \boldsymbol{\mathcal{I}}$; $u \neq \boldsymbol{\mathcal{I}}$ ⟧ ⟹ *False*
apply (*induct t*, *simp-all*)
proof −
fix *v w*
assume *I2*: ¬ *is-Tensor* (*w* ⌊⊗⌋ *u*) ⟹ *Diag w* ⟹ $w \neq \boldsymbol{\mathcal{I}}$ ⟹ *False*
assume *1*: ¬ *is-Tensor* ((*v* ⊗ *w*) ⌊⊗⌋ *u*)
assume *vw*: *Diag* (*v* ⊗ *w*)
assume *u*: *Diag u*
assume *2*: $u \neq \boldsymbol{\mathcal{I}}$
show *False*
proof −
have *v*: *v* = ⟨*un-Prim v*⟩
using *vw Diag-TensorE* [*of v w*] **by** *force*
have *w*: *Diag w* ∧ $w \neq \boldsymbol{\mathcal{I}}$
using *vw Diag-TensorE* [*of v w*] **by** *simp*
have (*v* ⊗ *w*) ⌊⊗⌋ *u* = *v* ⊗ (*w* ⌊⊗⌋ *u*)
proof −
have (*v* ⊗ *w*) ⌊⊗⌋ *u* = *v* ⌊⊗⌋ (*w* ⌊⊗⌋ *u*)
using *u 2* **by** (*cases u*, *simp-all*)
also have ... = *v* ⊗ (*w* ⌊⊗⌋ *u*)
using *u v w I2 TensorDiag-Prim not-is-Tensor-Unity* **by** *metis*
finally show *?thesis* **by** *simp*
qed
thus *?thesis* **using** *1* **by** *simp*
qed
qed
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *TensorDiag-assoc*:
assumes *Diag t* **and** *Diag u* **and** *Diag v*
shows (*t* ⌊⊗⌋ *u*) ⌊⊗⌋ *v* = *t* ⌊⊗⌋ (*u* ⌊⊗⌋ *v*)
proof −
have ⋀*n t u v*. ⟦ *dim t* = *n*; *Diag t*; *Diag u*; *Diag v* ⟧ ⟹
(*t* ⌊⊗⌋ *u*) ⌊⊗⌋ *v* = *t* ⌊⊗⌋ (*u* ⌊⊗⌋ *v*)
proof −
fix *n*
show ⋀*t u v*. ⟦ *dim t* = *n*; *Diag t*; *Diag u*; *Diag v* ⟧ ⟹
(*t* ⌊⊗⌋ *u*) ⌊⊗⌋ *v* = *t* ⌊⊗⌋ (*u* ⌊⊗⌋ *v*)
proof (*induction n rule*: *nat-less-induct*)
fix *n* :: *nat* **and** *t* :: *'a term* **and** *u v*
assume *I*: ∀ *m*<*n*. ∀ *t u v*. *dim t* = *m* ⟶ *Diag t* ⟶ *Diag u* ⟶ *Diag v* ⟶
(*t* ⌊⊗⌋ *u*) ⌊⊗⌋ *v* = *t* ⌊⊗⌋ (*u* ⌊⊗⌋ *v*)
assume *dim*: *dim t* = *n*
assume *t*: *Diag t*

assume u: *Diag u*
assume v: *Diag v*
show $(t \lfloor\otimes\rfloor\ u) \lfloor\otimes\rfloor\ v = t \lfloor\otimes\rfloor\ (u \lfloor\otimes\rfloor\ v)$
proof −
have $t = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis* **by** *simp*
moreover have $u = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis* **by** *simp*
moreover have $v = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis* **by** *simp*
moreover have $t \neq \boldsymbol{\mathcal{I}} \wedge u \neq \boldsymbol{\mathcal{I}} \wedge v \neq \boldsymbol{\mathcal{I}} \wedge$ *is-Prim t* $\Longrightarrow$ *?thesis*
using *v* **by** (*cases t*, *simp-all*, *cases u*, *simp-all*; *cases v*, *simp-all*)
moreover have $t \neq \boldsymbol{\mathcal{I}} \wedge u \neq \boldsymbol{\mathcal{I}} \wedge v \neq \boldsymbol{\mathcal{I}} \wedge$ *is-Tensor t* $\Longrightarrow$ *?thesis*
proof (*cases t*; *simp*)
fix w :: *'a term* **and** x :: *'a term*
assume *1*: $u \neq \boldsymbol{\mathcal{I}} \wedge v \neq \boldsymbol{\mathcal{I}}$
assume *2*: $t = (w \otimes x)$
show $((w \otimes x) \lfloor\otimes\rfloor\ u) \lfloor\otimes\rfloor\ v = (w \otimes x) \lfloor\otimes\rfloor\ (u \lfloor\otimes\rfloor\ v)$
proof −
have *w*: $w = \langle$*un-Prim w*$\rangle$
using *t 1 2 Diag-TensorE* [*of w x*] **by** *auto*
have *x*: *Diag x*
using *t w 1 2* **by** (*cases w*, *simp-all*)
have $((w \otimes x) \lfloor\otimes\rfloor\ u) \lfloor\otimes\rfloor\ v = (w \lfloor\otimes\rfloor\ (x \lfloor\otimes\rfloor\ u)) \lfloor\otimes\rfloor\ v$
using *u v w x 1 2* **by** (*cases u*, *simp-all*)
also have ... $= (w \otimes (x \lfloor\otimes\rfloor\ u)) \lfloor\otimes\rfloor\ v$
using *t w u 1 2 TensorDiag-Prim not-is-Tensor-TensorDiagE Diag-TensorE not-is-Tensor-Unity*
by *metis*
also have ... $= w \lfloor\otimes\rfloor\ ((x \lfloor\otimes\rfloor\ u) \lfloor\otimes\rfloor\ v)$
using *u v w x 1* **by** (*cases u*, *simp-all*; *cases v*, *simp-all*)
also have ... $= w \lfloor\otimes\rfloor\ (x \lfloor\otimes\rfloor\ (u \lfloor\otimes\rfloor\ v))$
proof −
have *dim x* < *dim t*
using *w 2* **by** (*cases w*, *simp-all*)
thus *?thesis*
using *u v x dim I* **by** *simp*
qed
also have ... $= (w \otimes x) \lfloor\otimes\rfloor\ (u \lfloor\otimes\rfloor\ v)$
proof −
have *3*: *is-Tensor* $(u \lfloor\otimes\rfloor\ v)$
using *u v 1 not-is-Tensor-TensorDiagE* **by** *auto*
obtain u' :: *'a term* **and** v' **where** *uv'*: $u \lfloor\otimes\rfloor\ v = u' \otimes v'$
using *3 is-Tensor-def* **by** *blast*
thus *?thesis* **by** *simp*
qed
finally show *?thesis* **by** *simp*
qed
qed
moreover have $t = \boldsymbol{\mathcal{I}} \vee$ *is-Prim t* $\vee$ *is-Tensor t*
using *t* **by** (*cases t*, *simp-all*)
ultimately show *?thesis* **by** *blast*

```
    qed
  qed
 qed
 thus ?thesis using assms by blast
qed

lemma TensorDiag-preserves-Ide:
assumes Ide t and Ide u and Diag t and Diag u
shows Ide (t ⌊⊗⌋ u)
 using assms
 by (metis (mono-tags, lifting) Arr-implies-Ide-Dom Ide-in-Hom Diag-implies-Arr
     TensorDiag-preserves-Diag(1) TensorDiag-preserves-Diag(2) mem-Collect-eq)

lemma TensorDiag-preserves-Can:
assumes Can t and Can u and Diag t and Diag u
shows Can (t ⌊⊗⌋ u)
proof −
 have ⋀u. ⟦ Can t ∧ Diag t; Can u ∧ Diag u ⟧ ⟹ Can (t ⌊⊗⌋ u)
 proof (induct t; simp)
  show ⋀x u. C.ide x ∧ C.arr x ⟹ Can u ∧ Diag u ⟹ Can (⟨x⟩ ⌊⊗⌋ u)
   by (metis Ide.simps(1) Ide.simps(2) Ide-implies-Can Diag.simps(2) TensorDiag-Prim
       TensorDiag-preserves-Ide Can.simps(3))
  show ⋀t1 t2 u. (⋀u. Diag t1 ⟹ Can u ∧ Diag u ⟹ Can (t1 ⌊⊗⌋ u)) ⟹
                 (⋀u. Diag t2 ⟹ Can u ∧ Diag u ⟹ Can (t2 ⌊⊗⌋ u)) ⟹
                 Can t1 ∧ Can t2 ∧ Diag (t1 ⊗ t2) ⟹ Can u ∧ Diag u ⟹
                 Can ((t1 ⊗ t2) ⌊⊗⌋ u)
   by (metis Diag-TensorE(3) Diag-TensorE(4) TensorDiag-Diag TensorDiag-assoc
       TensorDiag-preserves-Diag(1))
 qed
 thus ?thesis using assms by blast
qed

lemma Inv-TensorDiag:
assumes Can t and Can u and Diag t and Diag u
shows Inv (t ⌊⊗⌋ u) = Inv t ⌊⊗⌋ Inv u
proof −
 have ⋀u. ⟦ Can t ∧ Diag t; Can u ∧ Diag u ⟧ ⟹ Inv (t ⌊⊗⌋ u) = Inv t ⌊⊗⌋ Inv u
 proof (induct t, simp-all)
  fix f u
  assume f: C.ide f ∧ C.arr f
  assume u: Can u ∧ Diag u
  show Inv (⟨f⟩ ⌊⊗⌋ u) = ⟨f⟩ ⌊⊗⌋ Inv u
   using f u by (cases u, simp-all)
  next
  fix t u v
  assume I1: ⋀v. ⟦ Diag t; Can v ∧ Diag v ⟧ ⟹ Inv (t ⌊⊗⌋ v) = Inv t ⌊⊗⌋ Inv v
  assume I2: ⋀v. ⟦ Diag u; Can v ∧ Diag v ⟧ ⟹ Inv (u ⌊⊗⌋ v) = Inv u ⌊⊗⌋ Inv v
  assume tu: Can t ∧ Can u ∧ Diag (t ⊗ u)
  have t: Can t ∧ Diag t
```

```
    using tu Diag-TensorE [of t u] by force
  have u: Can u ∧ Diag u
    using t tu by (cases t, simp-all)
  assume v: Can v ∧ Diag v
  show Inv ((t ⊗ u) ⌊⊗⌋ v) = (Inv t ⊗ Inv u) ⌊⊗⌋ Inv v
  proof −
    have v = Unity ⟹ ?thesis by simp
    moreover have v ≠ Unity ⟹ ?thesis
    proof −
      assume 1: v ≠ Unity
      have Inv ((t ⊗ u) ⌊⊗⌋ v) = Inv (t ⌊⊗⌋ (u ⌊⊗⌋ v))
        using 1 by (cases v, simp-all)
      also have ... = Inv t ⌊⊗⌋ Inv (u ⌊⊗⌋ v)
        using t u v I1 TensorDiag-preserves-Diag TensorDiag-preserves-Can
              Inv-preserves-Diag Inv-preserves-Can
        by simp
      also have ... = Inv t ⌊⊗⌋ (Inv u ⌊⊗⌋ Inv v)
        using t u v I2 TensorDiag-preserves-Diag TensorDiag-preserves-Can
              Inv-preserves-Diag Inv-preserves-Can
        by simp
      also have ... = (Inv t ⊗ Inv u) ⌊⊗⌋ Inv v
        using v 1 by (cases v, simp-all)
      finally show ?thesis by blast
    qed
    ultimately show ?thesis by blast
  qed
 qed
 thus ?thesis using assms by blast
qed
```

The following function defines composition for compatible diagonal terms, by "pushing the composition down" to arrows of C.

```
fun CompDiag :: 'a term ⇒ 'a term ⇒ 'a term      (infixr ⌊·⌋ 55)
where 𝓘 ⌊·⌋ u = u
    | ⟨f⟩ ⌊·⌋ ⟨g⟩ = ⟨f · g⟩
    | (u ⊗ v) ⌊·⌋ (w ⊗ x) = (u ⌊·⌋ w ⊗ v ⌊·⌋ x)
    | t ⌊·⌋ 𝓘 = t
    | t ⌊·⌋ - = undefined · undefined
```

Note that the last clause above is not relevant to diagonal terms. We have chosen a provably non-diagonal value in order to validate associativity.

```
lemma CompDiag-preserves-Diag:
assumes Diag t and Diag u and Dom t = Cod u
shows Diag (t ⌊·⌋ u)
and Dom (t ⌊·⌋ u) = Dom u
and Cod (t ⌊·⌋ u) = Cod t
proof −
  have 0: ⋀u. ⟦ Diag t; Diag u; Dom t = Cod u ⟧ ⟹
              Diag (t ⌊·⌋ u) ∧ Dom (t ⌊·⌋ u) = Dom u ∧ Cod (t ⌊·⌋ u) = Cod t
```

proof (*induct t*, *simp-all add*: *Diag-TensorE*)
fix *f u*
assume *f*: *C.arr f*
assume *u*: *Diag u*
assume *1*: ⟨*C.dom f*⟩ = *Cod u*
show *Diag* (⟨*f*⟩ ⌊·⌋ *u*) ∧ *Dom* (⟨*f*⟩ ⌊·⌋ *u*) = *Dom u* ∧ *Cod* (⟨*f*⟩ ⌊·⌋ *u*) = ⟨*C.cod f*⟩
using *f u 1* **by** (*cases u*, *simp-all*)
next
fix *u v w*
assume *I2*: ⋀*u*. ⟦ *Diag u*; *Dom w* = *Cod u* ⟧ ⟹
Diag (*w* ⌊·⌋ *u*) ∧ *Dom* (*w* ⌊·⌋ *u*) = *Dom u* ∧ *Cod* (*w* ⌊·⌋ *u*) = *Cod w*
assume *vw*: *Diag* (*v* ⊗ *w*)
have *v*: *Diag v*
using *vw Diag-TensorE* [*of v w*] **by** *force*
have *w*: *Diag w*
using *vw Diag-TensorE* [*of v w*] **by** *force*
assume *u*: *Diag u*
assume *1*: (*Dom v* ⊗ *Dom w*) = *Cod u*
show *Diag* ((*v* ⊗ *w*) ⌊·⌋ *u*) ∧ *Dom* ((*v* ⊗ *w*) ⌊·⌋ *u*) = *Dom u* ∧
Cod ((*v* ⊗ *w*) ⌊·⌋ *u*) = *Cod v* ⊗ *Cod w*
using *u v w 1*
proof (*cases u*, *simp-all*)
fix *x y*
assume *2*: *u* = *Tensor x y*
have *4*: *is-Prim x* ∧ *x* = ⟨*un-Prim x*⟩ ∧ *C.arr* (*un-Prim x*) ∧ *Diag y* ∧ *y* ≠ 𝓘
using *u 2* **by** (*cases x*, *cases y*, *simp-all*)
have *5*: *is-Prim v* ∧ *v* = ⟨*un-Prim v*⟩ ∧ *C.arr* (*un-Prim v*) ∧ *Diag w* ∧ *w* ≠ 𝓘
using *v w vw* **by** (*cases v*, *simp-all*)
have *6*: *C.dom* (*un-Prim v*) = *C.cod* (*un-Prim x*) ∧ *Dom w* = *Cod y*
using *1 2 4 5* **apply** (*cases u*, *simp-all*)
by (*metis Cod.simps*(*1*) *Dom.simps*(*1*) *term.simps*(*1*))
have (*v* ⊗ *w*) ⌊·⌋ *u* = ⟨*un-Prim v* · *un-Prim x*⟩ ⊗ *w* ⌊·⌋ *y*
using *2 4 5 6 CompDiag.simps*(*2*) [*of un-Prim v un-Prim x*] **by** *simp*
moreover have *Diag* (⟨*un-Prim v* · *un-Prim x*⟩ ⊗ *w* ⌊·⌋ *y*)
proof −
have *Diag* (*w* ⌊·⌋ *y*)
using *I2 4 5 6* **by** *simp*
thus *?thesis*
using *4 5 6 Diag.simps*(*3*) [*of un-Prim v* · *un-Prim x* (*w* ⌊·⌋ *y*)]
by (*cases w*; *cases y*) *auto*
qed
ultimately show *Diag* (*v* ⌊·⌋ *x* ⊗ *w* ⌊·⌋ *y*) ∧
Dom (*v* ⌊·⌋ *x*) = *Dom x* ∧ *Dom* (*w* ⌊·⌋ *y*) = *Dom y* ∧
Cod (*v* ⌊·⌋ *x*) = *Cod v* ∧ *Cod* (*w* ⌊·⌋ *y*) = *Cod w*
using *4 5 6 I2*
by (*metis* (*full-types*) *C.cod-comp C.dom-comp Cod.simps*(*1*) *CompDiag.simps*(*2*)
Dom.simps(*1*) *C.seqI*)
qed
qed

show *Diag* (*t* ⌊·⌋ *u*) **using** *assms 0* **by** *blast*
show *Dom* (*t* ⌊·⌋ *u*) = *Dom u* **using** *assms 0* **by** *blast*
show *Cod* (*t* ⌊·⌋ *u*) = *Cod t* **using** *assms 0* **by** *blast*
qed

lemma *CompDiag-in-Hom*:
assumes *Diag t* **and** *Diag u* **and** *Dom t* = *Cod u*
shows *t* ⌊·⌋ *u* ∈ *Hom* (*Dom u*) (*Cod t*)
using *assms CompDiag-preserves-Diag Diag-implies-Arr* **by** *simp*

lemma *Dom-CompDiag*:
assumes *Diag t* **and** *Diag u* **and** *Dom t* = *Cod u*
shows *Dom* (*t* ⌊·⌋ *u*) = *Dom u*
using *assms CompDiag-preserves-Diag*(*2*) **by** *simp*

lemma *Cod-CompDiag*:
assumes *Diag t* **and** *Diag u* **and** *Dom t* = *Cod u*
shows *Cod* (*t* ⌊·⌋ *u*) = *Cod t*
using *assms CompDiag-preserves-Diag*(*3*) **by** *simp*

lemma *CompDiag-Cod-Diag* [*simp*]:
assumes *Diag t*
shows *Cod t* ⌊·⌋ *t* = *t*
proof −
have *Diag t* ⟹ *Cod t* ⌊·⌋ *t* = *t*
using *C.comp-cod-arr*
apply (*induct t*, *auto*)
by (*auto simp add*: *Diag-TensorE*)
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *CompDiag-Diag-Dom* [*simp*]:
assumes *Diag t*
shows *t* ⌊·⌋ *Dom t* = *t*
proof −
have *Diag t* ⟹ *t* ⌊·⌋ *Dom t* = *t*
using *C.comp-arr-dom*
apply (*induct t*, *auto*)
by (*auto simp add*: *Diag-TensorE*)
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *CompDiag-Ide-Diag* [*simp*]:
assumes *Diag t* **and** *Ide a* **and** *Dom a* = *Cod t*
shows *a* ⌊·⌋ *t* = *t*
using *assms Ide-in-Hom* **by** *simp*

lemma *CompDiag-Diag-Ide* [*simp*]:
assumes *Diag t* **and** *Ide a* **and** *Dom t* = *Cod a*

shows $t \lfloor\cdot\rfloor\ a = t$
using *assms Ide-in-Hom* **by** *auto*

lemma *CompDiag-assoc*:
assumes *Diag t* **and** *Diag u* **and** *Diag v*
and *Dom t = Cod u* **and** *Dom u = Cod v*
shows $(t \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = t \lfloor\cdot\rfloor\ (u \lfloor\cdot\rfloor\ v)$
proof −
have $\bigwedge u\ v.\ [\![\ \mathit{Diag}\ t;\ \mathit{Diag}\ u;\ \mathit{Diag}\ v;\ \mathit{Dom}\ t = \mathit{Cod}\ u;\ \mathit{Dom}\ u = \mathit{Cod}\ v\ ]\!] \Longrightarrow$
$(t \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = t \lfloor\cdot\rfloor\ (u \lfloor\cdot\rfloor\ v)$
proof (*induct t*, *simp-all*)
fix *f u v*
assume *f*: *C.arr f*
assume *u*: *Diag u*
assume *v*: *Diag v*
assume *1*: $\langle C.\mathit{dom}\ f\rangle = \mathit{Cod}\ u$
assume *2*: *Dom u = Cod v*
show $(\langle f\rangle \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = \langle f\rangle \lfloor\cdot\rfloor\ (u \lfloor\cdot\rfloor\ v)$
using *C.comp-assoc* **by** (*cases u*, *simp-all*; *cases v*, *simp-all*)
next
fix *u v w x*
assume *I1*: $\bigwedge u\ v.\ [\![\ \mathit{Diag}\ w;\ \mathit{Diag}\ u;\ \mathit{Diag}\ v;\ \mathit{Dom}\ w = \mathit{Cod}\ u;\ \mathit{Dom}\ u = \mathit{Cod}\ v\ ]\!] \Longrightarrow$
$(w \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = w \lfloor\cdot\rfloor\ (u \lfloor\cdot\rfloor\ v)$
assume *I2*: $\bigwedge u\ v.\ [\![\ \mathit{Diag}\ x;\ \mathit{Diag}\ u;\ \mathit{Diag}\ v;\ \mathit{Dom}\ x = \mathit{Cod}\ u;\ \mathit{Dom}\ u = \mathit{Cod}\ v\ ]\!] \Longrightarrow$
$(x \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = x \lfloor\cdot\rfloor\ (u \lfloor\cdot\rfloor\ v)$
assume *wx*: *Diag* $(w \otimes x)$
assume *u*: *Diag u*
assume *v*: *Diag v*
assume *1*: $(\mathit{Dom}\ w \otimes \mathit{Dom}\ x) = \mathit{Cod}\ u$
assume *2*: *Dom u = Cod v*
show $((w \otimes x) \lfloor\cdot\rfloor\ u) \lfloor\cdot\rfloor\ v = (w \otimes x) \lfloor\cdot\rfloor\ u \lfloor\cdot\rfloor\ v$
proof −
have *w*: *Diag w*
using *wx Diag-TensorE* **by** *blast*
have *x*: *Diag x*
using *wx Diag-TensorE* **by** *blast*
have *is-Tensor u*
using *u 1* **by** (*cases u*) *simp-all*
thus *?thesis*
using *u v* **apply** (*cases u*, *simp-all*, *cases v*, *simp-all*)
proof −
fix *u1 u2 v1 v2*
assume *3*: $u = (u1 \otimes u2)$
assume *4*: $v = (v1 \otimes v2)$
show $(w \lfloor\cdot\rfloor\ u1) \lfloor\cdot\rfloor\ v1 = w \lfloor\cdot\rfloor\ u1 \lfloor\cdot\rfloor\ v1 \wedge$
$(x \lfloor\cdot\rfloor\ u2) \lfloor\cdot\rfloor\ v2 = x \lfloor\cdot\rfloor\ u2 \lfloor\cdot\rfloor\ v2$
proof −
have *Diag u1* ∧ *Diag u2*
using *u 3 Diag-TensorE* **by** *blast*

```
        moreover have Diag v1 ∧ Diag v2
          using v 4 Diag-TensorE by blast
        ultimately show ?thesis using w x I1 I2 1 2 3 4 by simp
      qed
    qed
  qed
  thus ?thesis using assms by blast
qed

lemma CompDiag-preserves-Ide:
assumes Ide t and Ide u and Diag t and Diag u and Dom t = Cod u
shows Ide (t ⌊·⌋ u)
proof −
  have ⋀u. ⟦ Ide t; Ide u; Diag t; Diag u; Dom t = Cod u ⟧ ⟹ Ide (CompDiag t u)
    by (induct t; simp)
  thus ?thesis using assms by blast
qed

lemma CompDiag-preserves-Can:
assumes Can t and Can u and Diag t and Diag u and Dom t = Cod u
shows Can (t ⌊·⌋ u)
proof −
  have ⋀u. ⟦ Can t ∧ Diag t; Can u ∧ Diag u; Dom t = Cod u ⟧ ⟹ Can (t ⌊·⌋ u)
  proof (induct t, simp-all)
    fix t u v
    assume I1: ⋀v. ⟦ Diag t; Can v ∧ Diag v; Dom t = Cod v ⟧ ⟹ Can (t ⌊·⌋ v)
    assume I2: ⋀v. ⟦ Diag u; Can v ∧ Diag v; Dom u = Cod v ⟧ ⟹ Can (u ⌊·⌋ v)
    assume tu: Can t ∧ Can u ∧ Diag (t ⊗ u)
    have t: Can t ∧ Diag t
      using tu Diag-TensorE by blast
    have u: Can u ∧ Diag u
      using tu Diag-TensorE by blast
    assume v: Can v ∧ Diag v
    assume 1: (Dom t ⊗ Dom u) = Cod v
    show Can ((t ⊗ u) ⌊·⌋ v)
    proof −
      have 2: (Dom t ⊗ Dom u) = Cod v using 1 by simp
      show ?thesis
        using v 2
      proof (cases v; simp)
        fix w x
        assume wx: v = (w ⊗ x)
        have Can w ∧ Diag w using v wx Diag-TensorE [of w x] by auto
        moreover have Can x ∧ Diag x using v wx Diag-TensorE [of w x] by auto
        moreover have Dom t = Cod w using 2 wx by simp
        moreover have ux: Dom u = Cod x using 2 wx by simp
        ultimately show Can (t ⌊·⌋ w) ∧ Can (u ⌊·⌋ x)
          using t u I1 I2 by simp
```

```
    qed
  qed
 qed
 thus ?thesis using assms by blast
qed

lemma Inv-CompDiag:
assumes Can t and Can u and Diag t and Diag u and Dom t = Cod u
shows Inv (t ⌊·⌋ u) = Inv u ⌊·⌋ Inv t
proof −
 have ⋀u. ⟦ Can t ∧ Diag t; Can u ∧ Diag u; Dom t = Cod u ⟧ ⟹
       Inv (t ⌊·⌋ u) = Inv u ⌊·⌋ Inv t
 proof (induct t, simp-all)
  show ⋀x u. ⟦ C.ide x ∧ C.arr x; Can u ∧ Diag u; ⟨x⟩ = Cod u ⟧ ⟹
             Inv u = Inv u ⌊·⌋ Inv (Cod u)
   by (metis CompDiag-Diag-Dom Inv-Ide Inv-preserves-Can(2) Inv-preserves-Diag
       Ide.simps(1))
  show ⋀u. Can u ∧ Diag u ⟹ 𝓘 = Cod u ⟹ Inv u = Inv u ⌊·⌋ 𝓘
   by (simp add: Inv-preserves-Can(2) Inv-preserves-Diag)
  fix t u v
  assume tu: Can t ∧ Can u ∧ Diag (t ⊗ u)
  have t: Can t ∧ Diag t
   using tu Diag-TensorE by blast
  have u: Can u ∧ Diag u
   using tu Diag-TensorE by blast
  assume I1: ⋀v. ⟦ Diag t; Can v ∧ Diag v; Dom t = Cod v ⟧ ⟹
                 Inv (t ⌊·⌋ v) = Inv v ⌊·⌋ Inv t
  assume I2: ⋀v. ⟦ Diag u; Can v ∧ Diag v; Dom u = Cod v ⟧ ⟹
                 Inv (u ⌊·⌋ v) = Inv v ⌊·⌋ Inv u
  assume v: Can v ∧ Diag v
  assume 1: (Dom t ⊗ Dom u) = Cod v
  show Inv ((t ⊗ u) ⌊·⌋ v) = Inv v ⌊·⌋ (Inv t ⊗ Inv u)
   using v 1
  proof (cases v, simp-all)
   fix w x
   assume wx: v = (w ⊗ x)
   have Can w ∧ Diag w using v wx Diag-TensorE [of w x] by auto
   moreover have Can x ∧ Diag x using v wx Diag-TensorE [of w x] by auto
   moreover have Dom t = Cod w using wx 1 by simp
   moreover have Dom u = Cod x using wx 1 by simp
   ultimately show Inv (t ⌊·⌋ w) = Inv w ⌊·⌋ Inv t ∧
                   Inv (u ⌊·⌋ x) = Inv x ⌊·⌋ Inv u
    using t u I1 I2 by simp
  qed
 qed
 thus ?thesis using assms by blast
qed

lemma Can-and-Diag-implies-Ide:
```

assumes *Can t* **and** *Diag t*
shows *Ide t*
proof −
  have ⟦ *Can t*; *Diag t* ⟧ ⟹ *Ide t*
    apply (*induct t*, *simp-all*)
    by (*simp add*: *Diag-TensorE*)
  thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *CompDiag-Can-Inv* [*simp*]:
assumes *Can t* **and** *Diag t*
shows *t* ⌊·⌋ *Inv t* = *Cod t*
  using *assms Can-and-Diag-implies-Ide Ide-in-Hom* **by** *simp*

lemma *CompDiag-Inv-Can* [*simp*]:
assumes *Can t* **and** *Diag t*
shows *Inv t* ⌊·⌋ *t* = *Dom t*
  using *assms Can-and-Diag-implies-Ide Ide-in-Hom* **by** *simp*

The next fact is a syntactic version of the interchange law, for diagonal terms.

lemma *CompDiag-TensorDiag*:
assumes *Diag t* **and** *Diag u* **and** *Diag v* **and** *Diag w*
and *Seq t v* **and** *Seq u w*
shows (*t* ⌊⊗⌋ *u*) ⌊·⌋ (*v* ⌊⊗⌋ *w*) = (*t* ⌊·⌋ *v*) ⌊⊗⌋ (*u* ⌊·⌋ *w*)
proof −
  have ⋀*u v w*. ⟦ *Diag t*; *Diag u*; *Diag v*; *Diag w*; *Seq t v*; *Seq u w* ⟧ ⟹
                (*t* ⌊⊗⌋ *u*) ⌊·⌋ (*v* ⌊⊗⌋ *w*) = (*t* ⌊·⌋ *v*) ⌊⊗⌋ (*u* ⌊·⌋ *w*)
  proof (*induct t*, *simp-all*)
    fix *u v w*
    assume *u*: *Diag u*
    assume *v*: *Diag v*
    assume *w*: *Diag w*
    assume *uw*: *Seq u w*
    show *Arr v* ∧ 𝓘 = *Cod v* ⟹ *u* ⌊·⌋ (*v* ⌊⊗⌋ *w*) = *v* ⌊⊗⌋ (*u* ⌊·⌋ *w*)
      using *u v w uw* **by** (*cases v*) *simp-all*
    show ⋀*f*. ⟦ *C.arr f*; *Arr v* ∧ ⟨*C.dom f*⟩ = *Cod v* ⟧ ⟹
              (⟨*f*⟩ ⌊⊗⌋ *u*) ⌊·⌋ (*v* ⌊⊗⌋ *w*) = (⟨*f*⟩ ⌊·⌋ *v*) ⌊⊗⌋ (*u* ⌊·⌋ *w*)
    proof −
      fix *f*
      assume *f*: *C.arr f*
      assume *1*: *Arr v* ∧ ⟨*C.dom f*⟩ = *Cod v*
      show (⟨*f*⟩ ⌊⊗⌋ *u*) ⌊·⌋ (*v* ⌊⊗⌋ *w*) = (⟨*f*⟩ ⌊·⌋ *v*) ⌊⊗⌋ (*u* ⌊·⌋ *w*)
      proof −
        have *2*: *v* = ⟨*un-Prim v*⟩ ∧ *C.arr* (*un-Prim v*) **using** *v 1* **by** (*cases v*) *simp-all*
        have *u* = 𝓘 ⟹ *?thesis*
          using *v w uw 1 2 Cod.simps*(*3*) *CompDiag-Cod-Diag Dom.simps*(*2*)
                TensorDiag-Prim TensorDiag-term-Unity TensorDiag-preserves-Diag(*3*)
          by (*cases w*) *simp-all*
        moreover have *u* ≠ 𝓘 ⟹ *?thesis*

```
    proof −
      assume 3: u ≠ 𝓘
      hence 4: w ≠ 𝓘 using u w uw by (cases u, simp-all; cases w, simp-all)
      have (⟨f⟩ ⌊⊗⌋ u) ⌊·⌋ (v ⌊⊗⌋ w) = (⟨f⟩ ⊗ u) ⌊·⌋ (v ⊗ w)
      proof −
        have ⟨f⟩ ⌊⊗⌋ u = ⟨f⟩ ⊗ u
          using u f 3 TensorDiag-Diag by (cases u) simp-all
        moreover have v ⌊⊗⌋ w = v ⊗ w
          using w 2 4 TensorDiag-Diag by (cases v, simp-all; cases w, simp-all)
        ultimately show ?thesis by simp
      qed
      also have 5: ... = (⟨f⟩ ⌊·⌋ v) ⊗ (u ⌊·⌋ w) by simp
      also have ... = (⟨f⟩ ⌊·⌋ v) ⌊⊗⌋ (u ⌊·⌋ w)
        using f u w uw 1 2 3 4 5
              TensorDiag-Diag TensorDiag-Prim TensorDiag-preserves-Diag(1)
              CompDiag-preserves-Diag(1)
        by (metis Cod.simps(3) Dom.simps(1) Dom.simps(3) Diag.simps(2))
      finally show ?thesis by blast
    qed
    ultimately show ?thesis by blast
  qed
qed
fix t1 t2
assume I2: ⋀u v w. ⟦ Diag t2; Diag u; Diag v; Diag w;
                     Arr v ∧ Dom t2 = Cod v; Seq u w ⟧ ⟹
                     (t2 ⌊⊗⌋ u) ⌊·⌋ (v ⌊⊗⌋ w) = (t2 ⌊·⌋ v) ⌊⊗⌋ (u ⌊·⌋ w)
assume t12: Diag (t1 ⊗ t2)
have t1: t1 = ⟨un-Prim t1⟩ ∧ C.arr (un-Prim t1) ∧ Diag t1
  using t12 by (cases t1) simp-all
have t2: Diag t2 ∧ t2 ≠ 𝓘
  using t12 by (cases t1) simp-all
assume 1: Arr t1 ∧ Arr t2 ∧ Arr v ∧ Dom t1 ⊗ Dom t2 = Cod v
show ((t1 ⊗ t2) ⌊⊗⌋ u) ⌊·⌋ (v ⌊⊗⌋ w) = ((t1 ⊗ t2) ⌊·⌋ v) ⌊⊗⌋ (u ⌊·⌋ w)
proof −
  have u = 𝓘 ⟹ ?thesis
    using w uw TensorDiag-term-Unity CompDiag-Cod-Diag by (cases w) simp-all
  moreover have u ≠ 𝓘 ⟹ ?thesis
  proof −
    assume u': u ≠ 𝓘
    hence w': w ≠ 𝓘 using u w uw by (cases u; simp; cases w; simp)
    show ?thesis
      using 1 v
    proof (cases v, simp-all)
      fix v1 v2
      assume v12: v = Tensor v1 v2
      have v1: v1 = ⟨un-Prim v1⟩ ∧ C.arr (un-Prim v1) ∧ Diag v1
        using v v12 by (cases v1) simp-all
      have v2: Diag v2 ∧ v2 ≠ 𝓘
        using v v12 by (cases v1) simp-all
```

```
      have 2: v = (⟨un-Prim v1⟩ ⊗ v2)
        using v1 v12 by simp
      show ((t1 ⊗ t2) ⌊⊗⌋ u) ⌊·⌋ ((v1 ⊗ v2) ⌊⊗⌋ w)
            = ((t1 ⌊·⌋ v1) ⊗ (t2 ⌊·⌋ v2)) ⌊⊗⌋ (u ⌊·⌋ w)
      proof −
        have 3: (t1 ⊗ t2) ⌊⊗⌋ u = t1 ⌊⊗⌋ (t2 ⌊⊗⌋ u)
          using u u′ by (cases u) simp-all
        have 4: v ⌊⊗⌋ w = v1 ⌊⊗⌋ (v2 ⌊⊗⌋ w)
          using v w v1 v2 2 TensorDiag-assoc TensorDiag-Diag by metis
        have ((t1 ⊗ t2) ⌊⊗⌋ u) ⌊·⌋ ((v1 ⊗ v2) ⌊⊗⌋ w)
              = (t1 ⌊⊗⌋ (t2 ⌊⊗⌋ u)) ⌊·⌋ (v1 ⌊⊗⌋ (v2 ⌊⊗⌋ w))
          using 3 4 v12 by simp
        also have ... = (t1 ⌊·⌋ v1) ⌊⊗⌋ ((t2 ⌊⊗⌋ u) ⌊·⌋ (v2 ⌊⊗⌋ w))
        proof −
          have is-Tensor (t2 ⌊⊗⌋ u)
            using t2 u u′ not-is-Tensor-TensorDiagE by auto
          moreover have is-Tensor (v2 ⌊⊗⌋ w)
            using v2 v12 w w′ not-is-Tensor-TensorDiagE by auto
          ultimately show ?thesis
            using u u′ v w t1 v1 t12 v12 TensorDiag-Prim not-is-Tensor-Unity
            by (metis (no-types, lifting) CompDiag.simps(2) CompDiag.simps(3)
                is-Tensor-def)
        qed
        also have ... = (t1 ⌊·⌋ v1) ⌊⊗⌋ (t2 ⌊·⌋ v2) ⌊⊗⌋ (u ⌊·⌋ w)
          using u w uw t2 v2 1 2 Diag-implies-Arr I2 by fastforce
        also have ... = ((t1 ⌊·⌋ v1) ⊗ (t2 ⌊·⌋ v2)) ⌊⊗⌋ (u ⌊·⌋ w)
        proof −
          have u ⌊·⌋ w ≠ Unity
          proof −
            have Arr v1 ∧ ⟨C.dom (un-Prim t1)⟩ = Cod v1
              using t1 v1 1 2 by (cases t1, auto)
            thus ?thesis
              using t1 t2 v1 v2 u w uw u′ CompDiag-preserves-Diag
                    TensorDiag-preserves-Diag TensorDiag-Prim
              by (metis (mono-tags, lifting) Cod.simps(2) Cod.simps(3)
                  TensorDiag.simps(2) term.distinct(3))
          qed
          hence ((t1 ⌊·⌋ v1) ⊗ (t2 ⌊·⌋ v2)) ⌊⊗⌋ (u ⌊·⌋ w)
                = (t1 ⌊·⌋ v1) ⌊⊗⌋ ((t2 ⌊·⌋ v2) ⌊⊗⌋ (u ⌊·⌋ w))
            by (cases u ⌊·⌋ w) simp-all
          thus ?thesis by argo
        qed
        finally show ?thesis by blast
      qed
    qed
  qed
  ultimately show ?thesis by blast
 qed
qed
```

thus *?thesis* **using** *assms* **by** *blast*
qed

The following function reduces an arrow to diagonal form. The precise relationship between a term and its diagonalization is developed below.

fun *Diagonalize* :: *'a term* ⇒ *'a term* (⌊-⌋)
where ⌊⟨*f*⟩⌋ = ⟨*f*⟩
| ⌊$\mathcal{I}$⌋ = $\mathcal{I}$
| ⌊*t* ⊗ *u*⌋ = ⌊*t*⌋ ⌊⊗⌋ ⌊*u*⌋
| ⌊*t* ⋆ *u*⌋ = ⌊*t*⌋ ⌊⋆⌋ ⌊*u*⌋
| ⌊**l**[*t*]⌋ = ⌊*t*⌋
| ⌊**l**$^{-1}$[*t*]⌋ = ⌊*t*⌋
| ⌊**r**[*t*]⌋ = ⌊*t*⌋
| ⌊**r**$^{-1}$[*t*]⌋ = ⌊*t*⌋
| ⌊**a**[*t*, *u*, *v*]⌋ = (⌊*t*⌋ ⌊⊗⌋ ⌊*u*⌋) ⌊⊗⌋ ⌊*v*⌋
| ⌊**a**$^{-1}$[*t*, *u*, *v*]⌋ = ⌊*t*⌋ ⌊⊗⌋ (⌊*u*⌋ ⌊⊗⌋ ⌊*v*⌋)

lemma *Diag-Diagonalize*:
assumes *Arr t*
shows *Diag* ⌊*t*⌋ **and** *Dom* ⌊*t*⌋ = ⌊*Dom t*⌋ **and** *Cod* ⌊*t*⌋ = ⌊*Cod t*⌋
proof −
have *0*: *Arr t* ⟹ *Diag* ⌊*t*⌋ ∧ *Dom* ⌊*t*⌋ = ⌊*Dom t*⌋ ∧ *Cod* ⌊*t*⌋ = ⌊*Cod t*⌋
using *TensorDiag-preserves-Diag CompDiag-preserves-Diag TensorDiag-assoc*
apply (*induct t*)
apply *auto*
apply (*metis* (*full-types*))
by (*metis* (*full-types*))
show *Diag* ⌊*t*⌋ **using** *assms 0* **by** *blast*
show *Dom* ⌊*t*⌋ = ⌊*Dom t*⌋ **using** *assms 0* **by** *blast*
show *Cod* ⌊*t*⌋ = ⌊*Cod t*⌋ **using** *assms 0* **by** *blast*
qed

lemma *Diagonalize-in-Hom*:
assumes *Arr t*
shows ⌊*t*⌋ ∈ *Hom* ⌊*Dom t*⌋ ⌊*Cod t*⌋
using *assms Diag-Diagonalize Diag-implies-Arr* **by** *blast*

lemma *Diagonalize-Dom*:
assumes *Arr t*
shows ⌊*Dom t*⌋ = *Dom* ⌊*t*⌋
using *assms Diagonalize-in-Hom* **by** *simp*

lemma *Diagonalize-Cod*:
assumes *Arr t*
shows ⌊*Cod t*⌋ = *Cod* ⌊*t*⌋
using *assms Diagonalize-in-Hom* **by** *simp*

lemma *Diagonalize-preserves-Ide*:
assumes *Ide a*

shows *Ide* $\lfloor a \rfloor$
proof −
  have *Ide a* $\Longrightarrow$ *Ide* $\lfloor a \rfloor$
    using *Ide-implies-Arr TensorDiag-preserves-Ide Diag-Diagonalize*
    by (*induct a*) *simp-all*
  thus *?thesis* **using** *assms* **by** *blast*
qed

The diagonalizations of canonical arrows are identities.

lemma *Ide-Diagonalize-Can*:
assumes *Can t*
shows *Ide* $\lfloor t \rfloor$
proof −
  have *Can t* $\Longrightarrow$ *Ide* $\lfloor t \rfloor$
  using *Can-implies-Arr TensorDiag-preserves-Ide Diag-Diagonalize CompDiag-preserves-Ide*
        TensorDiag-preserves-Diag
    by (*induct t*) *auto*
  thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *Diagonalize-preserves-Can*:
assumes *Can t*
shows *Can* $\lfloor t \rfloor$
  using *assms Ide-Diagonalize-Can Ide-implies-Can* **by** *auto*

lemma *Diagonalize-Diag* [*simp*]:
assumes *Diag t*
shows $\lfloor t \rfloor = t$
proof −
  have *Diag t* $\Longrightarrow$ $\lfloor t \rfloor = t$
    apply (*induct t*, *simp-all*)
    using *TensorDiag-Prim Diag-TensorE* **by** *metis*
  thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *Diagonalize-Diagonalize* [*simp*]:
assumes *Arr t*
shows $\lfloor \lfloor t \rfloor \rfloor = \lfloor t \rfloor$
  using *assms Diag-Diagonalize Diagonalize-Diag* **by** *blast*

lemma *Diagonalize-Tensor*:
assumes *Arr t* **and** *Arr u*
shows $\lfloor t \otimes u \rfloor = \lfloor \lfloor t \rfloor \otimes \lfloor u \rfloor \rfloor$
  using *assms Diagonalize-Diagonalize* **by** *simp*

lemma *Diagonalize-Tensor-Unity-Arr* [*simp*]:
assumes *Arr u*
shows $\lfloor \mathcal{I} \otimes u \rfloor = \lfloor u \rfloor$
  using *assms* **by** *simp*

lemma *Diagonalize-Tensor-Arr-Unity* [*simp*]:
assumes *Arr t*
shows $\lfloor t \otimes \mathcal{I} \rfloor = \lfloor t \rfloor$
using *assms* **by** *simp*

lemma *Diagonalize-Tensor-Prim-Arr* [*simp*]:
assumes *arr f* **and** *Arr u* **and** $\lfloor u \rfloor \neq Unity$
shows $\lfloor \langle f \rangle \otimes u \rfloor = \langle f \rangle \otimes \lfloor u \rfloor$
using *assms* **by** *simp*

lemma *Diagonalize-Tensor-Tensor*:
assumes *Arr t* **and** *Arr u* **and** *Arr v*
shows $\lfloor (t \otimes u) \otimes v \rfloor = \lfloor \lfloor t \rfloor \otimes (\lfloor u \rfloor \otimes \lfloor v \rfloor) \rfloor$
using *assms Diag-Diagonalize Diagonalize-Diagonalize* **by** (*simp add*: *TensorDiag-assoc*)

lemma *Diagonalize-Comp-Cod-Arr*:
assumes *Arr t*
shows $\lfloor Cod\ t \cdot t \rfloor = \lfloor t \rfloor$
proof −
have *Arr t* $\Longrightarrow$ $\lfloor Cod\ t \cdot t \rfloor = \lfloor t \rfloor$
using *C.comp-cod-arr*
apply (*induct t*, *simp-all*)
using *CompDiag-TensorDiag Arr-implies-Ide-Cod Ide-in-Hom Diag-Diagonalize*
Diagonalize-in-Hom
apply *simp*
using *CompDiag-preserves-Diag CompDiag-Cod-Diag Diag-Diagonalize*
apply *metis*
using *CompDiag-TensorDiag Arr-implies-Ide-Cod Ide-in-Hom TensorDiag-in-Hom*
TensorDiag-preserves-Diag Diag-Diagonalize Diagonalize-in-Hom TensorDiag-assoc
by *simp-all*
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *Diagonalize-Comp-Arr-Dom*:
assumes *Arr t*
shows $\lfloor t \cdot Dom\ t \rfloor = \lfloor t \rfloor$
proof −
have *Arr t* $\Longrightarrow$ $\lfloor t \cdot Dom\ t \rfloor = \lfloor t \rfloor$
by (*metis CompDiag-Diag-Dom Diag-Diagonalize*(*1*−*2*) *Diagonalize.simps*(*4*))
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *Diagonalize-Inv*:
assumes *Can t*
shows $\lfloor Inv\ t \rfloor = Inv\ \lfloor t \rfloor$
proof −
have *Can t* $\Longrightarrow$ $\lfloor Inv\ t \rfloor = Inv\ \lfloor t \rfloor$
proof (*induct t*, *simp-all*)

```
    fix u v
    assume I1: ⌊Inv u⌋ = Inv ⌊u⌋
    assume I2: ⌊Inv v⌋ = Inv ⌊v⌋
    show Can u ∧ Can v ⟹ Inv ⌊u⌋ ⌊⊗⌋ Inv ⌊v⌋ = Inv (⌊u⌋ ⌊⊗⌋ ⌊v⌋)
      using Inv-TensorDiag Diag-Diagonalize Can-implies-Arr Diagonalize-preserves-Can
            I1 I2
      by simp
    show Can u ∧ Can v ∧ Dom u = Cod v ⟹ Inv ⌊v⌋ ⌊⋅⌋ Inv ⌊u⌋ = Inv (⌊u⌋ ⌊⋅⌋ ⌊v⌋)
      using Inv-CompDiag Diag-Diagonalize Can-implies-Arr Diagonalize-in-Hom
            Diagonalize-preserves-Can I1 I2
      by simp
    fix w
    assume I3: ⌊Inv w⌋ = Inv ⌊w⌋
    assume uvw: Can u ∧ Can v ∧ Can w
    show Inv ⌊u⌋ ⌊⊗⌋ (Inv ⌊v⌋ ⌊⊗⌋ Inv ⌊w⌋) = Inv ((⌊u⌋ ⌊⊗⌋ ⌊v⌋) ⌊⊗⌋ ⌊w⌋)
      using uvw I1 I2 I3
            Inv-TensorDiag Diag-Diagonalize Can-implies-Arr Diagonalize-preserves-Can
            TensorDiag-preserves-Diag TensorDiag-preserves-Can TensorDiag-assoc
      by simp
    show (Inv ⌊u⌋ ⌊⊗⌋ Inv ⌊v⌋) ⌊⊗⌋ Inv ⌊w⌋ = Inv (⌊u⌋ ⌊⊗⌋ (⌊v⌋ ⌊⊗⌋ ⌊w⌋))
      by (simp add: Can-implies-Arr Ide-Diagonalize-Can TensorDiag-assoc
          TensorDiag-preserves-Diag(1) TensorDiag-preserves-Ide Diag-Diagonalize(1) uvw)
  qed
  thus ?thesis using assms by blast
qed
```

Our next objective is to begin making the connection, to be completed in a subsequent section, between arrows and their diagonalizations. To summarize, an arrow t and its diagonalization $\lfloor t \rfloor$ are opposite sides of a square whose other sides are certain canonical terms $Dom\ t\!\downarrow \in Hom\ (Dom\ t)\ \lfloor Dom\ t \rfloor$ and $Cod\ t\!\downarrow \in Hom\ (Cod\ t)\ \lfloor Cod\ t \rfloor$, where $Dom\ t\!\downarrow$ and $Cod\ t\!\downarrow$ are defined by the function *red* below. The coherence theorem amounts to the statement that every such square commutes when the formal terms involved are evaluated in the evident way in any monoidal category.

Function *red* defined below takes an identity term a to a canonical arrow $a\!\downarrow \in Hom\ a\ \lfloor a \rfloor$. The auxiliary function *red2* takes a pair $(a,\ b)$ of diagonal identity terms and produces a canonical arrow $a \Downarrow b \in Hom\ (a \otimes b)\ \lfloor a \otimes b \rfloor$. The canonical arrow $a\!\downarrow$ amounts to a "parallel innermost reduction" from a to $\lfloor a \rfloor$, where the reduction steps are canonical arrows that involve the unitors and associator only in their uninverted forms. In general, a parallel innermost reduction from a will not be unique: at some points there is a choice available between left and right unitors and at other points there are choices between unitors and associators. These choices are inessential, and the ordering of the clauses in the function definitions below resolves them in an arbitrary way. What is more important is having chosen an innermost reduction, which is what allows us to write these definitions in structurally recursive form.

The essence of coherence is that the axioms for a monoidal category allow us to prove that any reduction from a to $\lfloor a \rfloor$ is equivalent (under evaluation of terms) to a parallel innermost reduction. The problematic cases are terms of the form $((a \otimes b) \otimes$

$c) \otimes d$, which present a choice between an inner and outer reduction that lead to terms with different structures. It is of course the pentagon axiom that ensures the confluence (under evaluation) of the two resulting paths.

Although simple in appearance, the structurally recursive definitions below were difficult to get right even after I started to understand what I was doing. I wish I could have just written them down straightaway. If so, then I could have avoided laboriously constructing and then throwing away thousands of lines of proof text that used a non-structural, "operational" approach to defining a reduction from a to $\lfloor a \rfloor$.

```
fun red2                    (infixr ⇓ 53)
where ℐ ⇓ a = l[a]
    | ⟨f⟩ ⇓ ℐ = r[⟨f⟩]
    | ⟨f⟩ ⇓ a = ⟨f⟩ ⊗ a
    | (a ⊗ b) ⇓ ℐ = r[a ⊗ b]
    | (a ⊗ b) ⇓ c = (a ⇓ ⌊b ⊗ c⌋) ⋅ (a ⊗ (b ⇓ c)) ⋅ a[a, b, c]
    | a ⇓ b = undefined

fun red                     (-↓ [56] 56)
where ℐ↓ = ℐ
    | ⟨f⟩↓ = ⟨f⟩
    | (a ⊗ b)↓ = (if Diag (a ⊗ b) then a ⊗ b else (⌊a⌋ ⇓ ⌊b⌋) ⋅ (a↓ ⊗ b↓))
    | a↓ = undefined

lemma red-Diag [simp]:
assumes Diag a
shows a↓ = a
  using assms by (cases a) simp-all

lemma red2-Diag:
assumes Diag (a ⊗ b)
shows a ⇓ b = a ⊗ b
proof −
  have a: a = ⟨un-Prim a⟩
    using assms Diag-TensorE by metis
  have b: Diag b ∧ b ≠ ℐ
    using assms Diag-TensorE by metis
  show ?thesis using a b
    apply (cases b)
      apply simp-all
     apply (metis red2.simps(3))
    by (metis red2.simps(4))
qed

lemma Can-red2:
assumes Ide a and Diag a and Ide b and Diag b
shows Can (a ⇓ b)
and a ⇓ b ∈ Hom (a ⊗ b) ⌊a ⊗ b⌋
proof −
  have 0: ⋀b. ⟦ Ide a ∧ Diag a; Ide b ∧ Diag b ⟧ ⟹
```

```
            Can (a ⇓ b) ∧ a ⇓ b ∈ Hom (a ⊗ b) ⌊a ⊗ b⌋
proof (induct a, simp-all)
  fix b
  show Ide b ∧ Diag b ⟹ Can b ∧ Dom b = b ∧ Cod b = b
    using Ide-implies-Can Ide-in-Hom Diagonalize-Diag by auto
  fix f
  show ⟦ C.ide f ∧ C.arr f; Ide b ∧ Diag b ⟧ ⟹
         Can (⟨f⟩ ⇓ b) ∧ Arr (⟨f⟩ ⇓ b) ∧ Dom (⟨f⟩ ⇓ b) = ⟨f⟩ ⊗ b ∧
                                    Cod (⟨f⟩ ⇓ b) = ⟨f⟩ ⌊⊗⌋ b
    using Ide-implies-Can Ide-in-Hom by (cases b; auto)
  next
  fix a b c
  assume ab: Ide a ∧ Ide b ∧ Diag (Tensor a b)
  assume c: Ide c ∧ Diag c
  assume I1: ⋀c. ⟦ Diag a; Ide c ∧ Diag c ⟧ ⟹
                 Can (a ⇓ c) ∧ Arr (a ⇓ c) ∧ Dom (a ⇓ c) = a ⊗ c ∧
                                      Cod (a ⇓ c) = a ⌊⊗⌋ c
  assume I2: ⋀c. ⟦ Diag b; Ide c ∧ Diag c ⟧ ⟹
                 Can (b ⇓ c) ∧ Arr (b ⇓ c) ∧ Dom (b ⇓ c) = b ⊗ c ∧
                                      Cod (b ⇓ c) = b ⌊⊗⌋ c
  show Can ((a ⊗ b) ⇓ c) ∧ Arr ((a ⊗ b) ⇓ c) ∧
       Dom ((a ⊗ b) ⇓ c) = (a ⊗ b) ⊗ c ∧
       Cod ((a ⊗ b) ⇓ c) = (⌊a⌋ ⌊⊗⌋ ⌊b⌋) ⌊⊗⌋ c
  proof −
    have a: Diag a ∧ Ide a
      using ab Diag-TensorE by blast
    have b: Diag b ∧ Ide b
      using ab Diag-TensorE by blast
    have c = 𝓘 ⟹ ?thesis
    proof −
      assume 1: c = 𝓘
      have 2: (a ⊗ b) ⇓ c = r[a ⊗ b]
        using 1 by simp
      have 3: Can (a ⇓ b) ∧ Arr (a ⇓ b) ∧ Dom (a ⇓ b) = a ⊗ b ∧ Cod (a ⇓ b) = a ⊗ b
        using a b ab 1 2 I1 Diagonalize-Diag Diagonalize.simps(3) by metis
      hence 4: Seq (a ⇓ b) r[a ⊗ b]
        using ab
        by (metis (mono-tags, lifting) Arr.simps(7) Cod.simps(3) Cod.simps(7)
            Diag-implies-Arr Ide-in-Hom mem-Collect-eq)
      have Can ((a ⊗ b) ⇓ c)
        using 1 2 3 4 ab by (simp add: Ide-implies-Can)
      moreover have Dom ((a ⊗ b) ⇓ c) = (a ⊗ b) ⊗ c
        using 1 2 3 4 a b ab I1 Ide-in-Hom TensorDiag-preserves-Diag by simp
      moreover have Cod ((a ⊗ b) ⇓ c) = ⌊(a ⊗ b) ⊗ c⌋
       using 1 2 3 4 ab using Diagonalize-Diag by fastforce
      ultimately show ?thesis using Can-implies-Arr by (simp add: 1 ab)
    qed
    moreover have c ≠ 𝓘 ⟹ ?thesis
    proof −
```

assume *1*: $c \neq \boldsymbol{\mathcal{I}}$
have *2*: $(a \otimes b) \Downarrow c = (a \Downarrow \lfloor b \otimes c \rfloor) \cdot (a \otimes b \Downarrow c) \cdot \mathbf{a}[a, b, c]$
using *1 a b ab c* **by** (*cases c*; *simp*)
have *3*: $Can\ (a \Downarrow \lfloor b \otimes c \rfloor) \wedge Dom\ (a \Downarrow \lfloor b \otimes c \rfloor) = a \otimes \lfloor b \otimes c \rfloor \wedge$
$Cod\ (a \Downarrow \lfloor b \otimes c \rfloor) = \lfloor a \otimes (b \otimes c) \rfloor$
proof −
have $Can\ (a \Downarrow \lfloor b \otimes c \rfloor) \wedge Dom\ (a \Downarrow \lfloor b \otimes c \rfloor) = a \otimes \lfloor b \otimes c \rfloor \wedge$
$Cod\ (a \Downarrow \lfloor b \otimes c \rfloor) = \lfloor a \otimes \lfloor b \otimes c \rfloor \rfloor$
using *a c ab 1 2 I1 Diag-implies-Arr Diag-Diagonalize(1)*
Diagonalize-preserves-Ide TensorDiag-preserves-Ide
TensorDiag-preserves-Diag(1)
by *auto*
moreover have $\lfloor a \otimes \lfloor b \otimes c \rfloor \rfloor = \lfloor a \otimes (b \otimes c) \rfloor$
using *ab c Diagonalize-Tensor Diagonalize-Diagonalize Diag-implies-Arr*
by (*metis Arr.simps(3) Diagonalize.simps(3)*)
ultimately show *?thesis* **by** *metis*
qed
have *4*: $Can\ (b \Downarrow c) \wedge Dom\ (b \Downarrow c) = b \otimes c \wedge Cod\ (b \Downarrow c) = \lfloor b \otimes c \rfloor$
using *b c ab 1 2 I2* **by** *simp*
hence $Can\ (a \otimes (b \Downarrow c)) \wedge Dom\ (a \otimes (b \Downarrow c)) = a \otimes (b \otimes c) \wedge$
$Cod\ (a \otimes (b \Downarrow c)) = a \otimes \lfloor b \otimes c \rfloor$
using *ab Ide-implies-Can Ide-in-Hom* **by** *force*
moreover have $\lfloor a \otimes \lfloor b \otimes c \rfloor \rfloor = \lfloor a \otimes b \rfloor \lfloor \otimes \rfloor \lfloor c \rfloor$
proof −
have $\lfloor a \otimes \lfloor b \otimes c \rfloor \rfloor = a \lfloor \otimes \rfloor (b \lfloor \otimes \rfloor c)$
using *a b c 4*
by (*metis Arr-implies-Ide-Dom Can-implies-Arr Ide-implies-Arr*
Diag-Diagonalize(1) Diagonalize.simps(3) Diagonalize-Diag)
also have $... = (a \lfloor \otimes \rfloor b) \lfloor \otimes \rfloor c$
using *a b ab c TensorDiag-assoc* **by** *metis*
also have $... = \lfloor a \otimes b \rfloor \lfloor \otimes \rfloor \lfloor c \rfloor$
using *a b c* **by** (*metis Diagonalize.simps(3) Diagonalize-Diag*)
finally show *?thesis* **by** *blast*
qed
moreover have $Can\ \mathbf{a}[a, b, c] \wedge Dom\ \mathbf{a}[a, b, c] = (a \otimes b) \otimes c \wedge$
$Cod\ \mathbf{a}[a, b, c] = a \otimes (b \otimes c)$
using *ab c Ide-implies-Can Ide-in-Hom* **by** *auto*
ultimately show *?thesis*
using *c 2 3 4 Diagonalize-Diagonalize Ide-implies-Can*
Diagonalize-Diag Arr-implies-Ide-Dom Can-implies-Arr
by (*metis Can.simps(4) Cod.simps(4) Dom.simps(4) Diagonalize.simps(3)*)
qed
ultimately show *?thesis* **by** *blast*
qed
qed
show $Can\ (a \Downarrow b)$ **using** *assms 0* **by** *blast*
show $a \Downarrow b \in Hom\ (a \otimes b)\ \lfloor a \otimes b \rfloor$ **using** *0 assms* **by** *blast*
qed

lemma *red2-in-Hom*:
assumes *Ide a* **and** *Diag a* **and** *Ide b* **and** *Diag b*
shows $a \Downarrow b \in Hom\ (a \otimes b)\ \lfloor a \otimes b \rfloor$
  using *assms Can-red2 Can-implies-Arr* **by** *simp*

lemma *Can-red*:
assumes *Ide a*
shows *Can* $(a\downarrow)$ **and** $a\downarrow \in Hom\ a\ \lfloor a \rfloor$
proof −
  have *0*: *Ide a* $\Longrightarrow$ *Can* $(a\downarrow) \wedge a\downarrow \in Hom\ a\ \lfloor a \rfloor$
  proof (*induct a*, *simp-all*)
    fix *b c*
    assume *b*: *Can* $(b\downarrow) \wedge$ *Arr* $(b\downarrow) \wedge$ *Dom* $(b\downarrow) = b \wedge$ *Cod* $(b\downarrow) = \lfloor b \rfloor$
    assume *c*: *Can* $(c\downarrow) \wedge$ *Arr* $(c\downarrow) \wedge$ *Dom* $(c\downarrow) = c \wedge$ *Cod* $(c\downarrow) = \lfloor c \rfloor$
    assume *bc*: *Ide b* $\wedge$ *Ide c*
    show (*Diag* $(b \otimes c) \longrightarrow$
          Can b $\wedge$ *Can c* $\wedge$ *Dom b* $= b \wedge$ *Dom c* $= c \wedge$ *Cod b* $\otimes$ *Cod c* $= \lfloor b \rfloor\ \lfloor \otimes \rfloor\ \lfloor c \rfloor) \wedge$
        $(\neg$ *Diag* $(b \otimes c) \longrightarrow$
          Can $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) \wedge$ *Dom* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor \otimes \lfloor c \rfloor \wedge$ *Arr* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) \wedge$
          Dom $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor \otimes \lfloor c \rfloor \wedge$ *Cod* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor\ \lfloor \otimes \rfloor\ \lfloor c \rfloor)$
    proof
      show *Diag* $(b \otimes c) \longrightarrow$
            Can b $\wedge$ *Can c* $\wedge$ *Dom b* $= b \wedge$ *Dom c* $= c \wedge$ *Cod b* $\otimes$ *Cod c* $= \lfloor b \rfloor\ \lfloor \otimes \rfloor\ \lfloor c \rfloor$
        using *bc Diag-TensorE Ide-implies-Can Inv-preserves-Can*(*2*)
            CompDiag-Ide-Diag Inv-Ide Diagonalize.simps(*3*) *Diagonalize-Diag*
        by (*metis CompDiag-Inv-Can*)
      show $\neg$ *Diag* $(b \otimes c) \longrightarrow$
            Can $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) \wedge$ *Dom* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor \otimes \lfloor c \rfloor \wedge$ *Arr* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) \wedge$
                  Dom $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor \otimes \lfloor c \rfloor \wedge$ *Cod* $(\lfloor b \rfloor \Downarrow \lfloor c \rfloor) = \lfloor b \rfloor\ \lfloor \otimes \rfloor\ \lfloor c \rfloor$
        using *b c bc Ide-in-Hom Ide-implies-Can Can-red2 Diag-Diagonalize*
            Diagonalize-preserves-Ide TensorDiag-preserves-Diag TensorDiag-preserves-Ide
        by *force*
    qed
  qed
  show *Can* $(a\downarrow)$ **using** *assms 0* **by** *blast*
  show $a\downarrow \in Hom\ a\ \lfloor a \rfloor$ **using** *assms 0* **by** *blast*
qed

lemma *red-in-Hom*:
assumes *Ide a*
shows $a\downarrow \in Hom\ a\ \lfloor a \rfloor$
  using *assms Can-red Can-implies-Arr* **by** *simp*

lemma *Diagonalize-red* [*simp*]:
assumes *Ide a*
shows $\lfloor a\downarrow \rfloor = \lfloor a \rfloor$
  using *assms Can-red Ide-Diagonalize-Can Diagonalize-in-Hom Ide-in-Hom* **by** *fastforce*

lemma *Diagonalize-red2* [*simp*]:

assumes *Ide a* **and** *Ide b* **and** *Diag a* **and** *Diag b*
shows $\lfloor a \Downarrow b \rfloor = \lfloor a \otimes b \rfloor$
using *assms Can-red2 Ide-Diagonalize-Can Diagonalize-in-Hom* [*of a* $\Downarrow$ *b*]
red2-in-Hom Ide-in-Hom
by *simp*

end

2.6 Coherence

If D is a monoidal category, then a functor $V\colon C \to D$ extends in an evident way to an evaluation map that interprets each formal arrow of the monoidal language of C as an arrow of D.

locale *evaluation-map* =
monoidal-language C +
monoidal-category D T α ι +
V: *functor C D V*
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and *D* :: *'d comp* (**infixr** $\cdot$ *55*)
and *T* :: $'d * 'd \Rightarrow 'd$
and α :: $'d * 'd * 'd \Rightarrow 'd$
and ι :: *'d*
and *V* :: $'c \Rightarrow 'd$
begin

no-notation *C.in-hom* («- : - → -»)

notation *unity* ($\mathcal{I}$)
notation *runit* (r[-])
notation *lunit* (l[-])
notation *assoc'* (a^{-1}[-, -, -])
notation *runit'* (r^{-1}[-])
notation *lunit'* (l^{-1}[-])

primrec *eval* :: *'c term* $\Rightarrow$ *'d* ({{-}})
where $\{\!\{\langle f \rangle\}\!\} = V\,f$
| $\{\!\{\boldsymbol{\mathcal{I}}\}\!\} = \mathcal{I}$
| $\{\!\{t \otimes u\}\!\} = \{\!\{t\}\!\} \otimes \{\!\{u\}\!\}$
| $\{\!\{t \cdot u\}\!\} = \{\!\{t\}\!\} \cdot \{\!\{u\}\!\}$
| $\{\!\{\mathbf{l}[t]\}\!\} = \mathfrak{l}\ \{\!\{t\}\!\}$
| $\{\!\{\mathbf{l}^{-1}[t]\}\!\} = \mathfrak{l}'\ \{\!\{t\}\!\}$
| $\{\!\{\mathbf{r}[t]\}\!\} = \varrho\ \{\!\{t\}\!\}$
| $\{\!\{\mathbf{r}^{-1}[t]\}\!\} = \varrho'\ \{\!\{t\}\!\}$
| $\{\!\{\mathbf{a}[t, u, v]\}\!\} = \alpha\ (\{\!\{t\}\!\}, \{\!\{u\}\!\}, \{\!\{v\}\!\})$
| $\{\!\{\mathbf{a}^{-1}[t, u, v]\}\!\} = \alpha'\ (\{\!\{t\}\!\}, \{\!\{u\}\!\}, \{\!\{v\}\!\})$

Identity terms evaluate to identities of D and evaluation preserves domain and codomain.

lemma *ide-eval-Ide* [*simp*]:
shows *Ide t* $\Longrightarrow$ *ide* $\{\!|t|\!\}$
by (*induct t*, *auto*)

lemma *eval-in-hom*:
shows *Arr t* $\Longrightarrow$ «$\{\!|t|\!\}$: $\{\!|Dom\ t|\!\}$ $\rightarrow$ $\{\!|Cod\ t|\!\}$»
apply (*induct t*)
apply *auto*[4]
apply *fastforce*
proof −
fix *t u v*
assume *I*: *Arr t* $\Longrightarrow$ «$\{\!|t|\!\}$: $\{\!|Dom\ t|\!\}$ $\rightarrow$ $\{\!|Cod\ t|\!\}$»
show *Arr* $\mathbf{l}^{-1}[t]$ $\Longrightarrow$ «$\{\!|\mathbf{l}^{-1}[t]|\!\}$: $\{\!|Dom\ \mathbf{l}^{-1}[t]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{l}^{-1}[t]|\!\}$»
using *I arr-dom-iff-arr* [*of* $\{\!|t|\!\}$] **by** *force*
show *Arr* $\mathbf{r}[t]$ $\Longrightarrow$ «$\{\!|\mathbf{r}[t]|\!\}$: $\{\!|Dom\ \mathbf{r}[t]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{r}[t]|\!\}$»
using *I arr-cod-iff-arr* [*of* $\{\!|t|\!\}$] **by** *force*
show *Arr* $\mathbf{r}^{-1}[t]$ $\Longrightarrow$ «$\{\!|\mathbf{r}^{-1}[t]|\!\}$: $\{\!|Dom\ \mathbf{r}^{-1}[t]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{r}^{-1}[t]|\!\}$»
using *I arr-dom-iff-arr* [*of* $\{\!|t|\!\}$] **by** *force*
assume *I1*: *Arr t* $\Longrightarrow$ «$\{\!|t|\!\}$: $\{\!|Dom\ t|\!\}$ $\rightarrow$ $\{\!|Cod\ t|\!\}$»
assume *I2*: *Arr u* $\Longrightarrow$ «$\{\!|u|\!\}$: $\{\!|Dom\ u|\!\}$ $\rightarrow$ $\{\!|Cod\ u|\!\}$»
assume *I3*: *Arr v* $\Longrightarrow$ «$\{\!|v|\!\}$: $\{\!|Dom\ v|\!\}$ $\rightarrow$ $\{\!|Cod\ v|\!\}$»
show *Arr* $\mathbf{a}[t, u, v]$ $\Longrightarrow$ «$\{\!|\mathbf{a}[t, u, v]|\!\}$: $\{\!|Dom\ \mathbf{a}[t, u, v]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{a}[t, u, v]|\!\}$»
proof −
assume *1*: *Arr* $\mathbf{a}[t, u, v]$
have *t*: «$\{\!|t|\!\}$: *dom* $\{\!|t|\!\}$ $\rightarrow$ *cod* $\{\!|t|\!\}$» **using** *1 I1* **by** *auto*
have *u*: «$\{\!|u|\!\}$: *dom* $\{\!|u|\!\}$ $\rightarrow$ *cod* $\{\!|u|\!\}$» **using** *1 I2* **by** *auto*
have *v*: «$\{\!|v|\!\}$: *dom* $\{\!|v|\!\}$ $\rightarrow$ *cod* $\{\!|v|\!\}$» **using** *1 I3* **by** *auto*
have $\{\!|\mathbf{a}[t, u, v]|\!\}$ = ($\{\!|t|\!\}$ $\otimes$ $\{\!|u|\!\}$ $\otimes$ $\{\!|v|\!\}$) $\cdot$ a[*dom* $\{\!|t|\!\}$, *dom* $\{\!|u|\!\}$, *dom* $\{\!|v|\!\}$]
using *t u v α-simp* [*of* $\{\!|t|\!\}$ $\{\!|u|\!\}$ $\{\!|v|\!\}$] **by** *auto*
moreover have «($\{\!|t|\!\}$ $\otimes$ $\{\!|u|\!\}$ $\otimes$ $\{\!|v|\!\}$) $\cdot$ a[*dom* $\{\!|t|\!\}$, *dom* $\{\!|u|\!\}$, *dom* $\{\!|v|\!\}$] :
(*dom* $\{\!|t|\!\}$ $\otimes$ *dom* $\{\!|u|\!\}$) $\otimes$ *dom* $\{\!|v|\!\}$ $\rightarrow$ *cod* $\{\!|t|\!\}$ $\otimes$ *cod* $\{\!|u|\!\}$ $\otimes$ *cod* $\{\!|v|\!\}$»
using *t u v* **by** (*elim in-homE*, *auto*)
moreover have $\{\!|Dom\ t|\!\}$ = *dom* $\{\!|t|\!\}$ $\wedge$ $\{\!|Dom\ u|\!\}$ = *dom* $\{\!|u|\!\}$ $\wedge$ $\{\!|Dom\ v|\!\}$ = *dom* $\{\!|v|\!\}$ $\wedge$
$\{\!|Cod\ t|\!\}$ = *cod* $\{\!|t|\!\}$ $\wedge$ $\{\!|Cod\ u|\!\}$ = *cod* $\{\!|u|\!\}$ $\wedge$ $\{\!|Cod\ v|\!\}$ = *cod* $\{\!|v|\!\}$
using *1 I1 I2 I3* **by** *auto*
ultimately show «$\{\!|\mathbf{a}[t, u, v]|\!\}$: $\{\!|Dom\ \mathbf{a}[t, u, v]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{a}[t, u, v]|\!\}$»
by *simp*
qed
show *Arr* $\mathbf{a}^{-1}[t, u, v]$ $\Longrightarrow$ «$\{\!|\mathbf{a}^{-1}[t, u, v]|\!\}$: $\{\!|Dom\ \mathbf{a}^{-1}[t, u, v]|\!\}$ $\rightarrow$ $\{\!|Cod\ \mathbf{a}^{-1}[t, u, v]|\!\}$»
proof −
assume *1*: *Arr* $\mathbf{a}^{-1}[t, u, v]$
have *t*: «$\{\!|t|\!\}$: *dom* $\{\!|t|\!\}$ $\rightarrow$ *cod* $\{\!|t|\!\}$» **using** *1 I1* **by** *auto*
have *u*: «$\{\!|u|\!\}$: *dom* $\{\!|u|\!\}$ $\rightarrow$ *cod* $\{\!|u|\!\}$» **using** *1 I2* **by** *auto*
have *v*: «$\{\!|v|\!\}$: *dom* $\{\!|v|\!\}$ $\rightarrow$ *cod* $\{\!|v|\!\}$» **using** *1 I3* **by** *auto*
have $\{\!|\mathbf{a}^{-1}[t, u, v]|\!\}$ = (($\{\!|t|\!\}$ $\otimes$ $\{\!|u|\!\}$) $\otimes$ $\{\!|v|\!\}$) $\cdot$ a^{-1}[*dom* $\{\!|t|\!\}$, *dom* $\{\!|u|\!\}$, *dom* $\{\!|v|\!\}$]
using *1 I1 I2 I3* *α′-simp* [*of* $\{\!|t|\!\}$ $\{\!|u|\!\}$ $\{\!|v|\!\}$] **by** *auto*
moreover have «(($\{\!|t|\!\}$ $\otimes$ $\{\!|u|\!\}$) $\otimes$ $\{\!|v|\!\}$) $\cdot$ a^{-1}[*dom* $\{\!|t|\!\}$, *dom* $\{\!|u|\!\}$, *dom* $\{\!|v|\!\}$] :
dom $\{\!|t|\!\}$ $\otimes$ *dom* $\{\!|u|\!\}$ $\otimes$ *dom* $\{\!|v|\!\}$ $\rightarrow$ (*cod* $\{\!|t|\!\}$ $\otimes$ *cod* $\{\!|u|\!\}$) $\otimes$ *cod* $\{\!|v|\!\}$»
using *t u v assoc′-in-hom* [*of dom* $\{\!|t|\!\}$ *dom* $\{\!|u|\!\}$ *dom* $\{\!|v|\!\}$]

```
    by (elim in-homE, auto)
  moreover have ⦇Dom t⦈ = dom ⦇t⦈ ∧ ⦇Dom u⦈ = dom ⦇u⦈ ∧ ⦇Dom v⦈ = dom ⦇v⦈ ∧
                ⦇Cod t⦈ = cod ⦇t⦈ ∧ ⦇Cod u⦈ = cod ⦇u⦈ ∧ ⦇Cod v⦈ = cod ⦇v⦈
    using 1 I1 I2 I3 by auto
  ultimately show «⦇a⁻¹[t, u, v]⦈ : ⦇Dom a⁻¹[t, u, v]⦈ → ⦇Cod a⁻¹[t, u, v]⦈»
    by simp
 qed
qed

lemma arr-eval [simp]:
assumes Arr f
shows arr ⦇f⦈
  using assms eval-in-hom by auto

lemma dom-eval [simp]:
assumes Arr f
shows dom ⦇f⦈ = ⦇Dom f⦈
  using assms eval-in-hom by auto

lemma cod-eval [simp]:
assumes Arr f
shows cod ⦇f⦈ = ⦇Cod f⦈
  using assms eval-in-hom by auto

lemma eval-Prim [simp]:
assumes C.arr f
shows ⦇⟨f⟩⦈ = V f
  by simp

lemma eval-Tensor [simp]:
assumes Arr t and Arr u
shows ⦇t ⊗ u⦈ = ⦇t⦈ ⊗ ⦇u⦈
  using assms eval-in-hom by auto

lemma eval-Comp [simp]:
assumes Arr t and Arr u and Dom t = Cod u
shows  ⦇t ⋆ u⦈ = ⦇t⦈ · ⦇u⦈
  using assms by simp

lemma eval-Lunit [simp]:
assumes Arr t
shows ⦇l[t]⦈ = l[⦇Cod t⦈] · (I ⊗ ⦇t⦈)
  using assms lunit-naturality [of ⦇t⦈] by simp

lemma eval-Lunit' [simp]:
assumes Arr t
shows ⦇l⁻¹[t]⦈ = l⁻¹[⦇Cod t⦈] · ⦇t⦈
  using assms lunit'-naturality [of ⦇t⦈] l'.map-simp [of ⦇t⦈] l-ide-simp
        Arr-implies-Ide-Cod
```

by *simp*

lemma *eval-Runit* [*simp*]:
assumes *Arr t*
shows $\{\!|\mathbf{r}[t]|\!\} = \mathrm{r}[\{\!|Cod\ t|\!\}] \cdot (\{\!|t|\!\} \otimes \mathcal{I})$
using *assms runit-naturality* [*of* $\{\!|t|\!\}$] **by** *simp*

lemma *eval-Runit'* [*simp*]:
assumes *Arr t*
shows $\{\!|\mathbf{r}^{-1}[t]|\!\} = \mathrm{r}^{-1}[\{\!|Cod\ t|\!\}] \cdot \{\!|t|\!\}$
using *assms runit'-naturality* [*of* $\{\!|t|\!\}$] *ϱ'.map-simp* [*of* $\{\!|t|\!\}$] *ϱ-ide-simp*
Arr-implies-Ide-Cod
by *simp*

lemma *eval-Assoc* [*simp*]:
assumes *Arr t* **and** *Arr u* **and** *Arr v*
shows $\{\!|\mathbf{a}[t, u, v]|\!\} = \mathrm{a}[cod\ \{\!|t|\!\}, cod\ \{\!|u|\!\}, cod\ \{\!|v|\!\}] \cdot ((\{\!|t|\!\} \otimes \{\!|u|\!\}) \otimes \{\!|v|\!\})$
using *assms α.is-natural-2* [*of* ($\{\!|t|\!\}$, $\{\!|u|\!\}$, $\{\!|v|\!\}$)] **by** *auto*

lemma *eval-Assoc'* [*simp*]:
assumes *Arr t* **and** *Arr u* **and** *Arr v*
shows $\{\!|\mathbf{a}^{-1}[t, u, v]|\!\} = \mathrm{a}^{-1}[cod\ \{\!|t|\!\}, cod\ \{\!|u|\!\}, cod\ \{\!|v|\!\}] \cdot (\{\!|t|\!\} \otimes \{\!|u|\!\} \otimes \{\!|v|\!\})$
using *assms α'-simp* [*of* $\{\!|t|\!\}$ $\{\!|u|\!\}$ $\{\!|v|\!\}$] *assoc'-naturality* [*of* $\{\!|t|\!\}$ $\{\!|u|\!\}$ $\{\!|v|\!\}$]
by *simp*

The following are conveniences for the case of identity arguments to avoid having to get rid of the extra identities that are introduced by the general formulas above.

lemma *eval-Lunit-Ide* [*simp*]:
assumes *Ide a*
shows $\{\!|\mathbf{l}[a]|\!\} = \mathrm{l}[\{\!|a|\!\}]$
using *assms comp-cod-arr* **by** *simp*

lemma *eval-Lunit'-Ide* [*simp*]:
assumes *Ide a*
shows $\{\!|\mathbf{l}^{-1}[a]|\!\} = \mathrm{l}^{-1}[\{\!|a|\!\}]$
using *assms comp-cod-arr* **by** *simp*

lemma *eval-Runit-Ide* [*simp*]:
assumes *Ide a*
shows $\{\!|\mathbf{r}[a]|\!\} = \mathrm{r}[\{\!|a|\!\}]$
using *assms comp-cod-arr* **by** *simp*

lemma *eval-Runit'-Ide* [*simp*]:
assumes *Ide a*
shows $\{\!|\mathbf{r}^{-1}[a]|\!\} = \mathrm{r}^{-1}[\{\!|a|\!\}]$
using *assms comp-cod-arr* **by** *simp*

lemma *eval-Assoc-Ide* [*simp*]:
assumes *Ide a* **and** *Ide b* **and** *Ide c*

shows $\llbracket \mathbf{a}[a,\ b,\ c] \rrbracket = \mathrm{a}[\llbracket a \rrbracket,\ \llbracket b \rrbracket,\ \llbracket c \rrbracket]$
using *assms* **by** *simp*

lemma *eval-Assoc'-Ide* [*simp*]:
assumes *Ide a* **and** *Ide b* **and** *Ide c*
shows $\llbracket \mathbf{a}^{-1}[a,\ b,\ c] \rrbracket = \mathrm{a}^{-1}[\llbracket a \rrbracket,\ \llbracket b \rrbracket,\ \llbracket c \rrbracket]$
using *assms* α'*-ide-simp* **by** *simp*

Canonical arrows evaluate to isomorphisms in D, and formal inverses evaluate to inverses in D.

lemma *iso-eval-Can*:
shows *Can t* $\Longrightarrow$ *iso* $\llbracket t \rrbracket$
using *Can-implies-Arr* $\mathfrak{l}'$*.preserves-iso* ϱ'*.preserves-iso* α*.preserves-iso* α'*.preserves-iso*
Arr-implies-Ide-Dom
by (*induct t*) *auto*

lemma *eval-Inv-Can*:
shows *Can t* $\Longrightarrow$ $\llbracket$*Inv t*$\rrbracket$ = *inv* $\llbracket t \rrbracket$
apply (*induct t*)
using *iso-eval-Can inv-comp Can-implies-Arr*
apply *auto*[*4*]
proof −
fix *t*
assume *I*: *Can t* $\Longrightarrow$ $\llbracket$*Inv t*$\rrbracket$ = *inv* $\llbracket t \rrbracket$
show *Can* $\mathbf{l}[t]$ $\Longrightarrow$ $\llbracket$*Inv* $\mathbf{l}[t]\rrbracket$ = *inv* $\llbracket \mathbf{l}[t] \rrbracket$
using *I* $\mathfrak{l}'$*.is-natural-2* [*of inv* $\llbracket t \rrbracket$] *iso-eval-Can* $\mathfrak{l}$*-ide-simp iso-is-arr*
comp-cod-arr inv-comp
by *simp*
show *Can* $\mathbf{r}[t]$ $\Longrightarrow$ $\llbracket$*Inv* $\mathbf{r}[t]\rrbracket$ = *inv* $\llbracket \mathbf{r}[t] \rrbracket$
using *I* ϱ'*.is-natural-2* [*of inv* $\llbracket t \rrbracket$] *iso-eval-Can* ϱ*-ide-simp iso-is-arr*
comp-cod-arr inv-comp
by *simp*
show *Can* $\mathbf{l}^{-1}[t]$ $\Longrightarrow$ $\llbracket$*Inv* $\mathbf{l}^{-1}[t]\rrbracket$ = *inv* $\llbracket \mathbf{l}^{-1}[t] \rrbracket$
proof −
assume *t*: *Can* $\mathbf{l}^{-1}[t]$
hence *1*: *iso* $\llbracket t \rrbracket$ **using** *iso-eval-Can* **by** *simp*
have *inv* $\llbracket \mathbf{l}^{-1}[t] \rrbracket$ = *inv* ($\mathfrak{l}'$ $\llbracket t \rrbracket$)
using *t* **by** *simp*
also have ... = *inv* (l^{-1}[*cod* $\llbracket t \rrbracket$] · $\llbracket t \rrbracket$)
using *1* $\mathfrak{l}'$*.is-natural-2* [*of* $\llbracket t \rrbracket$] $\mathfrak{l}'$*-ide-simp iso-is-arr* **by** *auto*
also have ... = $\llbracket$*Inv* $\mathbf{l}^{-1}[t]\rrbracket$
using *t I 1 iso-is-arr inv-comp* **by** *auto*
finally show *?thesis* **by** *simp*
qed
show *Can* $\mathbf{r}^{-1}[t]$ $\Longrightarrow$ $\llbracket$*Inv* $\mathbf{r}^{-1}[t]\rrbracket$ = *inv* $\llbracket \mathbf{r}^{-1}[t] \rrbracket$
proof −
assume *t*: *Can* $\mathbf{r}^{-1}[t]$
hence *1*: *iso* $\llbracket t \rrbracket$ **using** *iso-eval-Can* **by** *simp*
have *inv* $\llbracket \mathbf{r}^{-1}[t] \rrbracket$ = *inv* (ϱ' $\llbracket t \rrbracket$)

```
    using t by simp
  also have ... = inv (r⁻¹[cod ⦇t⦈] · ⦇t⦈)
    using 1 ϱ'.is-natural-2 [of ⦇t⦈] ϱ'-ide-simp iso-is-arr by auto
  also have ... = ⦇Inv r⁻¹[t]⦈
    using t I 1 iso-is-arr inv-comp by auto
  finally show ?thesis by simp
 qed
 next
 fix t u v
 assume I1: Can t ⟹ ⦇Inv t⦈ = inv ⦇t⦈
 assume I2: Can u ⟹ ⦇Inv u⦈ = inv ⦇u⦈
 assume I3: Can v ⟹ ⦇Inv v⦈ = inv ⦇v⦈
 show Can a[t, u, v] ⟹ ⦇Inv a[t, u, v]⦈ = inv ⦇a[t, u, v]⦈
 proof −
  assume tuv: Can a[t, u, v]
  have t: iso ⦇t⦈ using tuv iso-eval-Can by auto
  have u: iso ⦇u⦈ using tuv iso-eval-Can by auto
  have v: iso ⦇v⦈ using tuv iso-eval-Can by auto
  have ⦇Inv a[t, u, v]⦈ = α' (inv ⦇t⦈, inv ⦇u⦈, inv ⦇v⦈)
    using tuv I1 I2 I3 by simp
  also have ... = inv (a[cod ⦇t⦈, cod ⦇u⦈, cod ⦇v⦈] · ((⦇t⦈ ⊗ ⦇u⦈) ⊗ ⦇v⦈))
    using t u v α'-simp iso-is-arr inv-comp by auto
  also have ... = inv ((⦇t⦈ ⊗ ⦇u⦈ ⊗ ⦇v⦈) · a[dom ⦇t⦈, dom ⦇u⦈, dom ⦇v⦈])
    using t u v iso-is-arr assoc-naturality by simp
  also have ... = inv ⦇a[t, u, v]⦈
    using t u v iso-is-arr α-simp [of ⦇t⦈ ⦇u⦈ ⦇v⦈] by simp
  finally show ?thesis by simp
 qed
 show Can a⁻¹[t, u, v] ⟹ ⦇Inv a⁻¹[t, u, v]⦈ = inv ⦇a⁻¹[t, u, v]⦈
 proof −
  assume tuv: Can a⁻¹[t, u, v]
  have t: iso ⦇t⦈ using tuv iso-eval-Can by auto
  have u: iso ⦇u⦈ using tuv iso-eval-Can by auto
  have v: iso ⦇v⦈ using tuv iso-eval-Can by auto
  have ⦇Inv a⁻¹[t, u, v]⦈ = α (inv ⦇t⦈, inv ⦇u⦈, inv ⦇v⦈)
    using tuv I1 I2 I3 by simp
  also have ... = (inv ⦇t⦈ ⊗ inv ⦇u⦈ ⊗ inv ⦇v⦈) · a[cod ⦇t⦈, cod ⦇u⦈, cod ⦇v⦈]
    using t u v iso-is-arr α-simp [of inv ⦇t⦈ inv ⦇u⦈ inv ⦇v⦈] by simp
  also have ... = inv (a⁻¹[cod ⦇t⦈, cod ⦇u⦈, cod ⦇v⦈] · (⦇t⦈ ⊗ ⦇u⦈ ⊗ ⦇v⦈))
    using t u v iso-is-arr inv-comp by auto
  also have ... = inv (((⦇t⦈ ⊗ ⦇u⦈) ⊗ ⦇v⦈) · a⁻¹[dom ⦇t⦈, dom ⦇u⦈, dom ⦇v⦈])
    using t u v iso-is-arr assoc'-naturality by simp
  also have ... = inv ⦇a⁻¹[t, u, v]⦈
    using t u v iso-is-arr α'-simp by auto
  finally show ?thesis by blast
 qed
qed
```

The operation $\lfloor \cdot \rfloor$ evaluates to composition in D.

lemma *eval-CompDiag*:
assumes *Diag t* **and** *Diag u* **and** *Seq t u*
shows $\{\!|t \lfloor\cdot\rfloor u|\!\} = \{\!|t|\!\} \cdot \{\!|u|\!\}$
proof −
  have $\bigwedge u.\ [\![\ \mathit{Diag}\ t;\ \mathit{Diag}\ u;\ \mathit{Seq}\ t\ u\ ]\!] \Longrightarrow \{\!|t \lfloor\cdot\rfloor u|\!\} = \{\!|t|\!\} \cdot \{\!|u|\!\}$
    using *eval-in-hom comp-cod-arr*
  proof (*induct t*, *simp-all*)
    fix *u f*
    assume *u*: *Diag u*
    assume *f*: *C.arr f*
    assume *1*: *Arr u* ∧ ⟨*C.dom f*⟩ = *Cod u*
    show $\{\!|\langle f\rangle \lfloor\cdot\rfloor u|\!\} = V\,f \cdot \{\!|u|\!\}$
      using *f u 1 as-nat-trans.preserves-comp-2* **by** (*cases u*; *simp*)
    next
    fix *u v w*
    assume *I1*: $\bigwedge u.\ [\![\ \mathit{Diag}\ v;\ \mathit{Diag}\ u;\ \mathit{Arr}\ u \wedge \mathit{Dom}\ v = \mathit{Cod}\ u\ ]\!] \Longrightarrow$
                  $\{\!|v \lfloor\cdot\rfloor u|\!\} = \{\!|v|\!\} \cdot \{\!|u|\!\}$
    assume *I2*: $\bigwedge u.\ [\![\ \mathit{Diag}\ w;\ \mathit{Diag}\ u;\ \mathit{Arr}\ u \wedge \mathit{Dom}\ w = \mathit{Cod}\ u\ ]\!] \Longrightarrow$
                  $\{\!|w \lfloor\cdot\rfloor u|\!\} = \{\!|w|\!\} \cdot \{\!|u|\!\}$
    assume *vw*: *Diag* (*Tensor v w*)
    have *v*: *Diag v* ∧ *v* = *Prim* (*un-Prim v*)
      using *vw* **by** (*simp add*: *Diag-TensorE*)
    have *w*: *Diag w*
      using *vw* **by** (*simp add*: *Diag-TensorE*)
    assume *u*: *Diag u*
    assume *1*: *Arr v* ∧ *Arr w* ∧ *Arr u* ∧ *Dom v* ⊗ *Dom w* = *Cod u*
    show $\{\!|(v \otimes w) \lfloor\cdot\rfloor u|\!\} = (\{\!|v|\!\} \otimes \{\!|w|\!\}) \cdot \{\!|u|\!\}$
      using *u 1 eval-in-hom CompDiag-in-Hom*
    proof (*cases u*, *simp-all*)
      fix *x y*
      assume *3*: *u* = *x* ⊗ *y*
      assume *4*: *Arr v* ∧ *Arr w* ∧ *Dom v* = *Cod x* ∧ *Dom w* = *Cod y*
      have *x*: *Diag x*
        using *u 1 3 Diag-TensorE* [*of x y*] **by** *simp*
      have *y*: *Diag y*
        using *u x 1 3 Diag-TensorE* [*of x y*] **by** *simp*
      show $\{\!|v \lfloor\cdot\rfloor x|\!\} \otimes \{\!|w \lfloor\cdot\rfloor y|\!\} = (\{\!|v|\!\} \otimes \{\!|w|\!\}) \cdot (\{\!|x|\!\} \otimes \{\!|y|\!\})$
        using *v w x y 4 I1 I2 CompDiag-in-Hom eval-in-hom Diag-implies-Arr interchange*
        by *auto*
    qed
  qed
  thus *?thesis* **using** *assms* **by** *blast*
qed

For identity terms a and b, the reduction $(a \otimes b)\!\downarrow$ factors (under evaluation in D) into the parallel reduction $a\!\downarrow \otimes b\!\downarrow$, followed by a reduction of its codomain $\lfloor a\rfloor \Downarrow \lfloor b\rfloor$.

lemma *eval-red-Tensor*:
assumes *Ide a* **and** *Ide b*
shows $\{\!|(a \otimes b)\!\downarrow|\!\} = \{\!|\lfloor a\rfloor \Downarrow \lfloor b\rfloor|\!\} \cdot (\{\!|a\!\downarrow|\!\} \otimes \{\!|b\!\downarrow|\!\})$

proof −
have *Diag* $(a \otimes b) \Longrightarrow$ *?thesis*
using *assms Can-red2 Ide-implies-Arr red-Diag*
Diagonalize-Diag red2-Diag Can-implies-Arr iso-eval-Can iso-is-arr
apply *simp*
using *Diag-TensorE eval-Tensor Diagonalize-Diag Diag-implies-Arr red-Diag*
tensor-preserves-ide ide-eval-Ide dom-eval comp-arr-dom
by *metis*
moreover have ¬ *Diag* $(a \otimes b) \Longrightarrow$ *?thesis*
using *assms Can-red2* **by** (*simp add*: *Can-red*(*1*) *iso-eval-Can*)
ultimately show *?thesis* **by** *blast*
qed

lemma *eval-red2-Diag-Unity*:
assumes *Ide a* **and** *Diag a*
shows $\lbag a \Downarrow \boldsymbol{\mathcal{I}} \rbag = \mathrm{r}[\lbag a \rbag]$
using *assms tensor-preserves-ide ϱ-ide-simp unitor-coincidence unit-in-hom comp-cod-arr*
by (*cases a*, *auto*)

Define a formal arrow t to be "coherent" if the square formed by t, $\lfloor t \rfloor$ and the reductions *Dom* $t\!\downarrow$ and *Cod* $t\!\downarrow$ commutes under evaluation in D. We will show that all formal arrows are coherent. Since the diagonalizations of canonical arrows are identities, a corollary is that parallel canonical arrows have equal evaluations.

abbreviation *coherent*
where *coherent* $t \equiv \lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag t \rbag = \lbag \lfloor t \rfloor \rbag \cdot \lbag \mathit{Dom}\ t\!\downarrow \rbag$

Diagonal arrows are coherent, since for such arrows t the reductions *Dom* $t\!\downarrow$ and *Cod* $t\!\downarrow$ are identities.

lemma *Diag-implies-coherent*:
assumes *Diag t*
shows *coherent t*
using *assms Diag-implies-Arr Arr-implies-Ide-Dom Arr-implies-Ide-Cod*
Dom-preserves-Diag Cod-preserves-Diag Diagonalize-Diag red-Diag
comp-arr-dom comp-cod-arr
by *simp*

The evaluation of a coherent arrow t has a canonical factorization in D into the evaluations of a reduction *Dom* $t\!\downarrow$, diagonalization $\lfloor t \rfloor$, and inverse reduction *Inv* (*Cod* $t\!\downarrow$). This will later allow us to use the term *Inv* (*Cod* $t\!\downarrow$) $\cdot$ $\lfloor t \rfloor$ $\cdot$ *Dom* $t\!\downarrow$ as a normal form for t.

lemma *canonical-factorization*:
assumes *Arr t*
shows *coherent* $t \longleftrightarrow \lbag t \rbag = \mathit{inv}\ \lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag \lfloor t \rfloor \rbag \cdot \lbag \mathit{Dom}\ t\!\downarrow \rbag$
proof
assume *1*: *coherent t*
have *inv* $\lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag \lfloor t \rfloor \rbag \cdot \lbag \mathit{Dom}\ t\!\downarrow \rbag = \mathit{inv}\ \lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag t \rbag$
using *1* **by** *simp*
also have ... = (*inv* $\lbag \mathit{Cod}\ t\!\downarrow \rbag \cdot \lbag \mathit{Cod}\ t\!\downarrow \rbag$) $\cdot$ $\lbag t \rbag$

```
    using comp-assoc by simp
  also have ... = ⦇t⦈
    using assms 1 red-in-Hom inv-in-hom Arr-implies-Ide-Cod Can-red iso-eval-Can
          comp-cod-arr Ide-in-Hom inv-is-inverse
    by (simp add: comp-inv-arr)
  finally show ⦇t⦈ = inv ⦇Cod t↓⦈ · ⦇⌊t⌋⦈ · ⦇Dom t↓⦈ by simp
  next
  assume 1: ⦇t⦈ = inv ⦇Cod t↓⦈ · ⦇⌊t⌋⦈ · ⦇Dom t↓⦈
  hence ⦇Cod t↓⦈ · ⦇t⦈ = ⦇Cod t↓⦈ · inv ⦇Cod t↓⦈ · ⦇⌊t⌋⦈ · ⦇Dom t↓⦈ by simp
  also have ... = (⦇Cod t↓⦈ · inv ⦇Cod t↓⦈) · ⦇⌊t⌋⦈ · ⦇Dom t↓⦈
    using comp-assoc by simp
  also have ... = ⦇⌊t⌋⦈ · ⦇Dom t↓⦈
    using assms 1 red-in-Hom Arr-implies-Ide-Cod Can-red iso-eval-Can inv-is-inverse
          Diagonalize-in-Hom comp-arr-inv comp-cod-arr Arr-implies-Ide-Dom Diagonal-
ize-in-Hom
    by auto
  finally show coherent t by blast
qed
```

A canonical arrow is coherent if and only if its formal inverse is.

```
lemma Can-implies-coherent-iff-coherent-Inv:
assumes Can t
shows coherent t ⟷ coherent (Inv t)
proof
  have 1: ⋀t. Can t ⟹ coherent t ⟹ coherent (Inv t)
  proof −
    fix t
    assume Can t
    hence t: Can t ∧ Arr t ∧ Ide (Dom t) ∧ Ide (Cod t) ∧
             arr ⦇t⦈ ∧ iso ⦇t⦈ ∧ inverse-arrows ⦇t⦈ (inv ⦇t⦈) ∧
             Can ⌊t⌋ ∧ Arr ⌊t⌋ ∧ arr ⦇⌊t⌋⦈ ∧ iso ⦇⌊t⌋⦈ ∧ ⌊t⌋ ∈ Hom ⌊Dom t⌋ ⌊Cod t⌋ ∧
             inverse-arrows ⦇⌊t⌋⦈ (inv ⦇⌊t⌋⦈) ∧ Inv t ∈ Hom (Cod t) (Dom t)
      using assms Can-implies-Arr Arr-implies-Ide-Dom Arr-implies-Ide-Cod iso-eval-Can
            inv-is-inverse Diagonalize-in-Hom Diagonalize-preserves-Can Inv-in-Hom
      by simp
    assume coh: coherent t
    have ⦇Cod (Inv t)↓⦈ · ⦇Inv t⦈ = (inv ⦇⌊t⌋⦈ · ⦇⌊t⌋⦈) · ⦇Cod (Inv t)↓⦈ · ⦇Inv t⦈
      using t red-in-Hom comp-cod-arr comp-inv-arr
      by (simp add: canonical-factorization coh Diagonalize-preserves-Can
          ‹Can t› inv-is-inverse)
    also have ... = inv ⦇⌊t⌋⦈ · (⦇Cod t↓⦈ · ⦇t⦈) · inv ⦇t⦈
      using t eval-Inv-Can coh comp-assoc by auto
    also have ... = ⦇⌊Inv t⌋⦈ · ⦇Dom (Inv t)↓⦈
     using t Diagonalize-Inv eval-Inv-Can comp-arr-inv red-in-Hom comp-arr-dom comp-assoc
      by auto
    finally show coherent (Inv t) by blast
  qed
  show coherent t ⟹ coherent (Inv t) using assms 1 by simp
  show coherent (Inv t) ⟹ coherent t
```

```
  proof −
    assume coherent (Inv t)
    hence coherent (Inv (Inv t))
      using assms 1 Inv-preserves-Can by blast
    thus ?thesis using assms by simp
  qed
qed
```

Some special cases of coherence are readily dispatched.

```
lemma coherent-Unity:
shows coherent 𝓘
  by simp

lemma coherent-Prim:
assumes Arr ⟨f⟩
shows coherent ⟨f⟩
  using assms by simp

lemma coherent-Lunit-Ide:
assumes Ide a
shows coherent 𝗹[a]
proof −
  have a: Ide a ∧ Arr a ∧ Dom a = a ∧ Cod a = a ∧
          ide ⦇a⦈ ∧ ide ⦇⌊a⌋⦈ ∧ ⦇a↓⦈ ∈ hom ⦇a⦈ ⦇⌊a⌋⦈
    using assms Ide-implies-Arr Ide-in-Hom Diagonalize-preserves-Ide red-in-Hom by auto
  thus ?thesis
    using a lunit-naturality [of ⦇a↓⦈] comp-cod-arr by auto
qed

lemma coherent-Runit-Ide:
assumes Ide a
shows coherent 𝗿[a]
proof −
  have a: Ide a ∧ Arr a ∧ Dom a = a ∧ Cod a = a ∧
          ide ⦇a⦈ ∧ ide ⦇⌊a⌋⦈ ∧ ⦇a↓⦈ ∈ hom ⦇a⦈ ⦇⌊a⌋⦈
    using assms Ide-implies-Arr Ide-in-Hom Diagonalize-preserves-Ide red-in-Hom
    by auto
  have ⦇Cod 𝗿[a]↓⦈ · ⦇𝗿[a]⦈ = ⦇a↓⦈ · r[⦇a⦈]
    using a runit-in-hom comp-cod-arr by simp
  also have ... = r[⦇⌊a⌋⦈] · (⦇a↓⦈ ⊗ 𝓘)
    using a eval-Runit runit-naturality [of ⦇red a⦈] by auto
  also have ... = ⦇⌊𝗿[a]⌋⦈ · ⦇Dom 𝗿[a]↓⦈
  proof −
    have ¬ Diag (a ⊗ 𝓘) by (cases a; simp)
    thus ?thesis
      using a comp-cod-arr red2-in-Hom eval-red2-Diag-Unity Diag-Diagonalize
            Diagonalize-preserves-Ide
      by auto
  qed
```

finally show *?thesis* **by** *blast*
qed

lemma *coherent-Lunit'-Ide*:
assumes *Ide a*
shows *coherent* $\mathbf{l}^{-1}[a]$
using *assms Ide-implies-Can coherent-Lunit-Ide*
Can-implies-coherent-iff-coherent-Inv [*of Lunit a*] **by** *simp*

lemma *coherent-Runit'-Ide*:
assumes *Ide a*
shows *coherent* $\mathbf{r}^{-1}[a]$
using *assms Ide-implies-Can coherent-Runit-Ide*
Can-implies-coherent-iff-coherent-Inv [*of Runit a*] **by** *simp*

To go further, we need the next result, which is in some sense the crux of coherence: For diagonal identities a, b, and c, the reduction $((a \lfloor\otimes\rfloor\ b) \Downarrow c) \cdot ((a \Downarrow b) \otimes c)$ from $(a \otimes b) \otimes c$ that first reduces the subterm $a \otimes b$ and then reduces the result, is equivalent under evaluation in D to the reduction that first applies the associator $\mathbf{a}[a,\ b,\ c]$ and then applies the reduction $(a \Downarrow b \lfloor\otimes\rfloor\ c) \cdot (a \otimes b \Downarrow c)$ from $a \otimes b \otimes c$. The triangle and pentagon axioms are used in the proof.

lemma *coherence-key-fact*:
assumes *Ide a* $\wedge$ *Diag a* **and** *Ide b* $\wedge$ *Diag b* **and** *Ide c* $\wedge$ *Diag c*
shows $\{\!|(a \lfloor\otimes\rfloor\ b) \Downarrow c|\!\} \cdot (\{\!|a \Downarrow b|\!\} \otimes \{\!|c|\!\})$
$= (\{\!|a \Downarrow (b \lfloor\otimes\rfloor\ c)|\!\} \cdot (\{\!|a|\!\} \otimes \{\!|b \Downarrow c|\!\})) \cdot \mathrm{a}[\{\!|a|\!\}, \{\!|b|\!\}, \{\!|c|\!\}]$
proof −
have $b = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis*
using *assms not-is-Tensor-TensorDiagE eval-red2-Diag-Unity triangle*
comp-cod-arr comp-assoc
by *simp*

The triangle is used!

moreover have $c = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis*
using *assms TensorDiag-preserves-Diag TensorDiag-preserves-Ide*
not-is-Tensor-TensorDiagE eval-red2-Diag-Unity
red2-in-Hom runit-tensor runit-naturality [*of* $\{\!|a \Downarrow b|\!\}$] *comp-assoc*
by *simp*
moreover have $[\![\ b \neq \boldsymbol{\mathcal{I}};\ c \neq \boldsymbol{\mathcal{I}}\ ]\!] \Longrightarrow$ *?thesis*
proof −
assume *b'*: $b \neq \boldsymbol{\mathcal{I}}$
hence *b*: *Ide b* $\wedge$ *Diag b* $\wedge$ *Arr b* $\wedge$ $b \neq \boldsymbol{\mathcal{I}} \wedge$
ide $\{\!|b|\!\} \wedge$ *arr* $\{\!|b|\!\} \wedge \lfloor b \rfloor = b \wedge b\downarrow = b \wedge$ *Dom b* $= b \wedge$ *Cod b* $= b$
using *assms Diagonalize-preserves-Ide Ide-in-Hom* **by** *simp*
assume *c'*: $c \neq \boldsymbol{\mathcal{I}}$
hence *c*: *Ide c* $\wedge$ *Diag c* $\wedge$ *Arr c* $\wedge$ $c \neq \boldsymbol{\mathcal{I}} \wedge$
ide $\{\!|c|\!\} \wedge$ *arr* $\{\!|c|\!\} \wedge \lfloor c \rfloor = c \wedge c\downarrow = c \wedge$ *Dom c* $= c \wedge$ *Cod c* $= c$
using *assms Diagonalize-preserves-Ide Ide-in-Hom* **by** *simp*
have $\bigwedge a.$ *Ide a* $\wedge$ *Diag a* $\Longrightarrow$
$\{\!|(a \lfloor\otimes\rfloor\ b) \Downarrow c|\!\} \cdot (\{\!|a \Downarrow b|\!\} \otimes \{\!|c|\!\})$

```
            = (⦇a ⇓ (b ⌊⊗⌋ c)⦈ · (⦇a⦈ ⊗ ⦇b ⇓ c⦈)) · a[⦇a⦈, ⦇b⦈, ⦇c⦈]
proof −
  fix a :: 'c term
  show Ide a ∧ Diag a ⟹
       ⦇(a ⌊⊗⌋ b) ⇓ c⦈ · (⦇a ⇓ b⦈ ⊗ ⦇c⦈)
         = (⦇a ⇓ (b ⌊⊗⌋ c)⦈ · (⦇a⦈ ⊗ ⦇b ⇓ c⦈)) · a[⦇a⦈, ⦇b⦈, ⦇c⦈]
    apply (induct a)
    using b c TensorDiag-in-Hom apply simp-all
  proof −
    show ⦇b ⇓ c⦈ · (⦇b⦈ · l[⦇b⦈] ⊗ ⦇c⦈)
          = ((⦇b ⌊⊗⌋ c⦈ · l[⦇b ⌊⊗⌋ c⦈]) · (ℐ ⊗ ⦇b ⇓ c⦈)) · a[ℐ, ⦇b⦈, ⦇c⦈]
    proof −
      have ⦇b ⌊⊗⌋ c⦈ · (l[⦇b ⌊⊗⌋ c⦈] · (ℐ ⊗ ⦇b ⇓ c⦈)) · a[ℐ, ⦇b⦈, ⦇c⦈] =
           ⦇b ⌊⊗⌋ c⦈ · (⦇b ⇓ c⦈ · l[⦇b⦈] ⊗ ⦇c⦈])  · a[ℐ, ⦇b⦈, ⦇c⦈]
        using b c red2-in-Hom lunit-naturality [of ⦇b ⇓ c⦈] by simp
      thus ?thesis
      using b c red2-in-Hom lunit-tensor comp-arr-dom comp-cod-arr comp-assoc by simp
    qed
    show ⋀f. C.ide f ∧ C.arr f ⟹
           ⦇(⟨f⟩ ⊗ b) ⇓ c⦈ · (⦇⟨f⟩ ⇓ b⦈ ⊗ ⦇c⦈)
             = (⦇⟨f⟩ ⇓ (b ⌊⊗⌋ c)⦈ · (V f ⊗ ⦇b ⇓ c⦈)) · a[V f, ⦇b⦈, ⦇c⦈]
    proof −
      fix f
      assume f: C.ide f ∧ C.arr f
      show ⦇(⟨f⟩ ⊗ b) ⇓ c⦈ · (⦇⟨f⟩ ⇓ b⦈ ⊗ ⦇c⦈)
            = (⦇⟨f⟩ ⇓ (b ⌊⊗⌋ c)⦈ · (V f ⊗ ⦇b ⇓ c⦈)) · a[V f, ⦇b⦈, ⦇c⦈]
      proof −
        have ⦇(⟨f⟩ ⊗ b) ⇓ c⦈ · (⦇⟨f⟩ ⇓ b⦈ ⊗ ⦇c⦈)
              = ((V f ⊗ ⦇b ⌊⊗⌋ c⦈) · (V f ⊗ ⦇b ⇓ c⦈) · a[V f, ⦇b⦈, ⦇c⦈]) ·
                ((V f ⊗ ⦇b⦈) ⊗ ⦇c⦈)
        proof −
          have ⦇⟨f⟩ ⇓ b⦈ = V f ⊗ ⦇b⦈
            using assms f b c red2-Diag by simp
          moreover have ⦇⟨f⟩ ⇓ b ⌊⊗⌋ c⦈ = V f ⊗ ⦇b ⌊⊗⌋ c⦈
          proof −
            have is-Tensor (b ⌊⊗⌋ c)
              using assms b c not-is-Tensor-TensorDiagE by blast
            thus ?thesis
              using assms f b c red2-Diag TensorDiag-preserves-Diag(1)
              by (cases b ⌊⊗⌋ c; simp)
          qed
          ultimately show ?thesis
            using assms b c by (cases c, simp-all)
        qed
        also have ... = ((V f ⊗ ⦇b ⌊⊗⌋ c⦈) · (V f ⊗ ⦇b ⇓ c⦈)) · a[V f, ⦇b⦈, ⦇c⦈]
          using b c f TensorDiag-in-Hom red2-in-Hom comp-arr-dom comp-cod-arr
          by simp
        also have ... = (⦇⟨f⟩ ⇓ (b ⌊⊗⌋ c)⦈ · (V f ⊗ ⦇b ⇓ c⦈)) · a[V f, ⦇b⦈, ⦇c⦈]
          using b c f Ide-implies-Arr TensorDiag-preserves-Ide not-is-Tensor-TensorDiagE
```

```
      by (cases b ⌊⊗⌋ c, simp-all; blast)
    finally show ?thesis by blast
  qed
qed
fix d e
assume I: Diag e ⟹ ⦇(e ⌊⊗⌋ b) ⇓ c⦈ · (⦇e ⇓ b⦈ ⊗ ⦇c⦈)
                     = (⦇e ⇓ b ⌊⊗⌋ c⦈ · (⦇e⦈ ⊗ ⦇b ⇓ c⦈)) · a[⦇e⦈, ⦇b⦈, ⦇c⦈]
assume de: Ide d ∧ Ide e ∧ Diag (d ⊗ e)
show ⦇((d ⊗ e) ⌊⊗⌋ b) ⇓ c⦈ · (⦇(d ⊗ e) ⇓ b⦈ ⊗ ⦇c⦈)
      = (⦇(d ⊗ e) ⇓ (b ⌊⊗⌋ c)⦈ · ((⦇d⦈ ⊗ ⦇e⦈) ⊗ ⦇b ⇓ c⦈)) · a[⦇d⦈ ⊗ ⦇e⦈, ⦇b⦈, ⦇c⦈]
proof −
  let ?f = un-Prim d
  have is-Prim d
    using de by (cases d, simp-all)
  hence d = ⟨?f⟩ ∧ C.ide ?f
    using de by (cases d, simp-all)
  hence d: Ide d ∧ Arr d ∧ Dom d = d ∧ Cod d = d ∧ Diag d ∧
          d = ⟨?f⟩ ∧ C.ide ?f ∧ ide ⦇d⦈ ∧ arr ⦇d⦈
    using de ide-eval-Ide Ide-implies-Arr Diag-Diagonalize(1) Ide-in-Hom
        Diag-TensorE [of d e]
    by simp
  have Diag e ∧ e ≠ 𝓘
    using de Diag-TensorE by metis
  hence e: Ide e ∧ Arr e ∧ Dom e = e ∧ Cod e = e ∧ Diag e ∧
          e ≠ 𝓘 ∧ ide ⦇e⦈ ∧ arr ⦇e⦈
    using de Ide-in-Hom by simp
  have 1: is-Tensor (e ⌊⊗⌋ b) ∧ is-Tensor (b ⌊⊗⌋ c) ∧ is-Tensor (e ⌊⊗⌋ (b ⌊⊗⌋ c))
    using b c e de not-is-Tensor-TensorDiagE TensorDiag-preserves-Diag
        not-is-Tensor-TensorDiagE [of e b ⌊⊗⌋ c]
    by auto
  have ⦇((d ⊗ e) ⌊⊗⌋ b) ⇓ c⦈ · (⦇(d ⊗ e) ⇓ b⦈ ⊗ ⦇c⦈)
        = ((⦇d⦈ ⊗ ⦇(e ⌊⊗⌋ b) ⌊⊗⌋ c⦈) · (⦇d⦈ ⊗ ⦇(e ⌊⊗⌋ b) ⇓ c⦈) ·
          a[⦇d⦈, ⦇e ⌊⊗⌋ b⦈, ⦇c⦈]) ·
          ((⦇d⦈ ⊗ ⦇e ⌊⊗⌋ b⦈) · (⦇d⦈ ⊗ ⦇e ⇓ b⦈) · a[⦇d⦈, ⦇e⦈, ⦇b⦈] ⊗ ⦇c⦈)
  proof −
    have ⦇((d ⊗ e) ⌊⊗⌋ b) ⇓ c⦈
          = (⦇d⦈ ⊗ ⦇(e ⌊⊗⌋ b) ⌊⊗⌋ c⦈) · (⦇d⦈ ⊗ ⦇(e ⌊⊗⌋ b) ⇓ c⦈) ·
            a[⦇d⦈, ⦇e ⌊⊗⌋ b⦈, ⦇c⦈]
    proof −
      have ((d ⊗ e) ⌊⊗⌋ b) ⇓ c = (d ⊗ (e ⌊⊗⌋ b)) ⇓ c
      using b c d e de 1 TensorDiag-Diag TensorDiag-preserves-Diag TensorDiag-assoc
          TensorDiag-Prim not-is-Tensor-Unity
        by metis
      also have ... = (d ⇓ (⌊e ⌊⊗⌋ b⌋ ⌊⊗⌋ c)) ⋆ (d ⊗ ((e ⌊⊗⌋ b) ⇓ c)) ⋆
                      a[d, e ⌊⊗⌋ b, c]
        using c d 1 by (cases c) simp-all
      also have ... = (d ⊗ ((e ⌊⊗⌋ b) ⌊⊗⌋ c)) ⋆ (d ⊗ ((e ⌊⊗⌋ b) ⇓ c)) ⋆
                      a[d, e ⌊⊗⌋ b, c]
        by (metis 1 Diagonalize-Diag TensorDiag-assoc TensorDiag-preserves-Diag(1)
```

b c d e is-Tensor-def red2.simps(4))
finally show *?thesis*
using *b c d e TensorDiag-in-Hom red2-in-Hom TensorDiag-preserves-Diag TensorDiag-preserves-Ide*
by *simp*
qed
moreover have $\{\!|(d \otimes e) \Downarrow b|\!\}$
$= (\{\!|d|\!\} \otimes \{\!|e \lfloor\otimes\rfloor\ b|\!\}) \cdot (\{\!|d|\!\} \otimes \{\!|e \Downarrow b|\!\}) \cdot \mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}]$
proof −
have $(d \otimes e) \Downarrow b = (d \Downarrow (e \lfloor\otimes\rfloor\ b)) \cdot (d \otimes (e \Downarrow b)) \cdot \mathbf{a}[d, e, b]$
using *b c d e de 1 TensorDiag-Prim Diagonalize-Diag*
by (*cases b*) *simp-all*
also have $... = (d \otimes (e \lfloor\otimes\rfloor\ b)) \cdot (d \otimes (e \Downarrow b)) \cdot \mathbf{a}[d, e, b]$
using *b d e 1 TensorDiag-preserves-Diag red2-Diag*
by (*metis Diag.simps(3) de term.disc(12)*)
finally have $(d \otimes e) \Downarrow b = (d \otimes (e \lfloor\otimes\rfloor\ b)) \cdot (d \otimes (e \Downarrow b)) \cdot \mathbf{a}[d, e, b]$
by *simp*
thus *?thesis* **using** *b d e eval-in-hom TensorDiag-in-Hom red2-in-Hom* **by** *simp*
qed
ultimately show *?thesis* **by** *argo*
qed
also have $... = (\{\!|d|\!\} \otimes \{\!|(e \lfloor\otimes\rfloor\ b) \Downarrow c|\!\}) \cdot \mathrm{a}[\{\!|d|\!\}, \{\!|e \lfloor\otimes\rfloor\ b|\!\}, \{\!|c|\!\}] \cdot$
$((\{\!|d|\!\} \otimes \{\!|e \Downarrow b|\!\}) \otimes \{\!|c|\!\}) \cdot (\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}] \otimes \{\!|c|\!\})$
using *b c d e red2-in-Hom TensorDiag-preserves-Ide*
TensorDiag-preserves-Diag interchange comp-cod-arr comp-assoc
by *simp*
also have $... = (\{\!|d|\!\} \otimes \{\!|(e \lfloor\otimes\rfloor\ b) \Downarrow c|\!\}) \cdot (\{\!|d|\!\} \otimes (\{\!|e \Downarrow b|\!\} \otimes \{\!|c|\!\})) \cdot$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\} \otimes \{\!|b|\!\}, \{\!|c|\!\}] \cdot (\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}] \otimes \{\!|c|\!\})$
using *b c d e TensorDiag-in-Hom red2-in-Hom TensorDiag-preserves-Ide*
TensorDiag-preserves-Diag assoc-naturality [*of* $\{\!|d|\!\}$ $\{\!|e \Downarrow b|\!\}$ $\{\!|c|\!\}$]
comp-permute [*of* $\mathrm{a}[\{\!|d|\!\}, \{\!|e \lfloor\otimes\rfloor\ b|\!\}, \{\!|c|\!\}]$ $(\{\!|d|\!\} \otimes \{\!|e \Downarrow b|\!\}) \otimes \{\!|c|\!\}$
$\{\!|d|\!\} \otimes (\{\!|e \Downarrow b|\!\} \otimes \{\!|c|\!\})$ $\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\} \otimes \{\!|b|\!\}, \{\!|c|\!\}]$]
by *simp*
also have $... = (\{\!|d|\!\} \otimes \{\!|(e \lfloor\otimes\rfloor\ b) \Downarrow c|\!\} \cdot (\{\!|e \Downarrow b|\!\} \otimes \{\!|c|\!\})) \cdot$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\} \otimes \{\!|b|\!\}, \{\!|c|\!\}] \cdot (\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}] \otimes \{\!|c|\!\})$
using *b c d e TensorDiag-in-Hom red2-in-Hom TensorDiag-preserves-Ide*
TensorDiag-preserves-Diag interchange
comp-reduce [*of* $\{\!|d|\!\} \otimes \{\!|(e \lfloor\otimes\rfloor\ b) \Downarrow c|\!\}$
$\{\!|d|\!\} \otimes (\{\!|e \Downarrow b|\!\} \otimes \{\!|c|\!\})$
$\{\!|d|\!\} \otimes \{\!|(e \lfloor\otimes\rfloor\ b) \Downarrow c|\!\} \cdot (\{\!|e \Downarrow b|\!\} \otimes \{\!|c|\!\})$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\} \otimes \{\!|b|\!\}, \{\!|c|\!\}] \cdot (\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}] \otimes \{\!|c|\!\})$]
by *simp*
also have $... = ((\{\!|d|\!\} \otimes \{\!|e \Downarrow (b \lfloor\otimes\rfloor\ c)|\!\}) \cdot (\{\!|d|\!\} \otimes \{\!|e|\!\} \otimes \{\!|b \Downarrow c|\!\})) \cdot$
$(\{\!|d|\!\} \otimes \mathrm{a}[\{\!|e|\!\}, \{\!|b|\!\}, \{\!|c|\!\}])) \cdot$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\} \otimes \{\!|b|\!\}, \{\!|c|\!\}] \cdot (\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\}] \otimes \{\!|c|\!\})$
using *b c d e I TensorDiag-in-Hom red2-in-Hom TensorDiag-preserves-Ide*
TensorDiag-preserves-Diag interchange
by *simp*
also have $... = ((\{\!|d|\!\} \otimes \{\!|e \Downarrow (b \lfloor\otimes\rfloor\ c)|\!\}) \cdot (\{\!|d|\!\} \otimes (\{\!|e|\!\} \otimes \{\!|b \Downarrow c|\!\}))) \cdot$

$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b|\!\} \otimes \{\!|c|\!\}] \cdot \mathrm{a}[\{\!|d|\!\} \otimes \{\!|e|\!\}, \{\!|b|\!\}, \{\!|c|\!\}]$
using *b c d e comp-assoc red2-in-Hom TensorDiag-in-Hom ide-eval-Ide TensorDiag-preserves-Diag tensor-preserves-ide TensorDiag-preserves-Ide pentagon*
by *simp*

The pentagon is used!

also have ... $= ((\{\!|d|\!\} \otimes \{\!|e \mathbin{\lfloor\otimes\rfloor} b \mathbin{\lfloor\otimes\rfloor} c|\!\}) \cdot (\{\!|d|\!\} \otimes \{\!|e \Downarrow b \mathbin{\lfloor\otimes\rfloor} c|\!\}) \cdot$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b \mathbin{\lfloor\otimes\rfloor} c|\!\}]) \cdot ((\{\!|d|\!\} \otimes \{\!|e|\!\}) \otimes \{\!|b \Downarrow c|\!\})) \cdot$
$\mathrm{a}[\{\!|d|\!\} \otimes \{\!|e|\!\}, \{\!|b|\!\}, \{\!|c|\!\}]$
using *b c d e red2-in-Hom TensorDiag-preserves-Ide TensorDiag-preserves-Diag assoc-naturality [of* $\{\!|d|\!\}$ $\{\!|e|\!\}$ $\{\!|b \Downarrow c|\!\}$*] comp-cod-arr comp-assoc*
by *simp*
also have ... $= (\{\!|(d \otimes e) \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c)|\!\}) \cdot ((\{\!|d|\!\} \otimes \{\!|e|\!\}) \otimes \{\!|b \Downarrow c|\!\})) \cdot$
$\mathrm{a}[\{\!|d|\!\} \otimes \{\!|e|\!\}, \{\!|b|\!\}, \{\!|c|\!\}]$
proof −
have $\{\!|(d \otimes e) \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c)|\!\}$
$= (\{\!|d|\!\} \otimes \{\!|e \mathbin{\lfloor\otimes\rfloor} (b \mathbin{\lfloor\otimes\rfloor} c)|\!\}) \cdot (\{\!|d|\!\} \otimes \{\!|e \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c)|\!\}) \cdot$
$\mathrm{a}[\{\!|d|\!\}, \{\!|e|\!\}, \{\!|b \mathbin{\lfloor\otimes\rfloor} c|\!\}]$
proof −
have $(d \otimes e) \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c)$
$= (d \Downarrow (e \mathbin{\lfloor\otimes\rfloor} \lfloor b \mathbin{\lfloor\otimes\rfloor} c\rfloor)) \cdot (d \otimes (e \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot \mathbf{a}[d, e, b \mathbin{\lfloor\otimes\rfloor} c]$
using *b c e not-is-Tensor-TensorDiagE*
by (*cases b* $\lfloor\otimes\rfloor$ *c*) *auto*
also have ... $= (d \Downarrow (e \mathbin{\lfloor\otimes\rfloor} (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot (d \otimes (e \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot$
$\mathbf{a}[d, e, b \mathbin{\lfloor\otimes\rfloor} c]$
using *b c d e 1 TensorDiag-preserves-Diag Diagonalize-Diag* **by** *simp*
also have ... $= (d \otimes (e \mathbin{\lfloor\otimes\rfloor} (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot (d \otimes (e \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot$
$\mathbf{a}[d, e, b \mathbin{\lfloor\otimes\rfloor} c]$
using *b c d e 1 TensorDiag-preserves-Diag(1) red2-Diag*
by (*metis Diag.simps(3) de not-is-Tensor-Unity*)
finally have $(d \otimes e) \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c)$
$= (d \otimes (e \mathbin{\lfloor\otimes\rfloor} (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot (d \otimes (e \Downarrow (b \mathbin{\lfloor\otimes\rfloor} c))) \cdot$
$\mathbf{a}[d, e, b \mathbin{\lfloor\otimes\rfloor} c]$
by *blast*
thus *?thesis*
using *b c d e red2-in-Hom TensorDiag-in-Hom TensorDiag-preserves-Diag TensorDiag-preserves-Ide*
by *simp*
qed
thus *?thesis* **using** *d e b c* **by** *simp*
qed
finally show *?thesis* **by** *simp*
qed
qed
qed
thus *?thesis* **using** *assms(1)* **by** *blast*
qed
ultimately show *?thesis* **by** *blast*

qed

lemma *coherent-Assoc-Ide*:
assumes *Ide a* **and** *Ide b* **and** *Ide c*
shows *coherent* $\mathbf{a}[a,\ b,\ c]$
proof −
  have *a*: *Ide a* ∧ *Arr a* ∧ *Dom a* = *a* ∧ *Cod a* = *a* ∧
        ide $\lbag a \rbag$ ∧ *ide* $\lbag \lfloor a \rfloor \rbag$ ∧ «$\lbag a\downarrow \rbag$: $\lbag a \rbag$ → $\lbag \lfloor a \rfloor \rbag$»
    using *assms Ide-implies-Arr Ide-in-Hom Diagonalize-preserves-Ide red-in-Hom* **by** *auto*
  have *b*: *Ide b* ∧ *Arr b* ∧ *Dom b* = *b* ∧ *Cod b* = *b* ∧
        ide $\lbag b \rbag$ ∧ *ide* $\lbag \lfloor b \rfloor \rbag$ ∧ «$\lbag b\downarrow \rbag$: $\lbag b \rbag$ → $\lbag \lfloor b \rfloor \rbag$»
    using *assms Ide-implies-Arr Ide-in-Hom Diagonalize-preserves-Ide red-in-Hom* **by** *auto*
  have *c*: *Ide c* ∧ *Arr c* ∧ *Dom c* = *c* ∧ *Cod c* = *c* ∧
        ide $\lbag c \rbag$ ∧ *ide* $\lbag \lfloor c \rfloor \rbag$ ∧ «$\lbag c\downarrow \rbag$: $\lbag c \rbag$ → $\lbag \lfloor c \rfloor \rbag$»
    using *assms Ide-implies-Arr Ide-in-Hom Diagonalize-preserves-Ide red-in-Hom* **by** *auto*
  have $\lbag$*Cod* $\mathbf{a}[a,\ b,\ c]\downarrow\rbag$ · $\lbag \mathbf{a}[a,\ b,\ c] \rbag$
        = ($\lbag \lfloor a \rfloor \Downarrow (\lfloor b \rfloor \ \lfloor\otimes\rfloor\ \lfloor c \rfloor) \rbag$ · ($\lbag \lfloor a \rfloor \rbag$ ⊗ ($\lbag \lfloor b \rfloor \Downarrow \lfloor c \rfloor \rbag$)) ·
          ($\lbag a\downarrow \rbag$ ⊗ $\lbag b\downarrow \rbag$ ⊗ $\lbag c\downarrow \rbag$)) · a[$\lbag a \rbag$, $\lbag b \rbag$, $\lbag c \rbag$]
    using *a b c red-in-Hom red2-in-Hom Diagonalize-in-Hom Diag-Diagonalize*
        Diagonalize-preserves-Ide interchange Ide-in-Hom eval-red-Tensor
        comp-cod-arr [*of* $\lbag a\downarrow \rbag$]
    by *simp*
  also have ... = (($\lbag \lfloor a \rfloor \Downarrow (\lfloor b \rfloor \ \lfloor\otimes\rfloor\ \lfloor c \rfloor) \rbag$ · ($\lbag \lfloor a \rfloor \rbag$ ⊗ $\lbag \lfloor b \rfloor \Downarrow \lfloor c \rfloor \rbag$)) ·
                a[$\lbag \lfloor a \rfloor \rbag$, $\lbag \lfloor b \rfloor \rbag$, $\lbag \lfloor c \rfloor \rbag$]) · (($\lbag a\downarrow \rbag$ ⊗ $\lbag b\downarrow \rbag$) ⊗ $\lbag c\downarrow \rbag$)
    using *a b c red-in-Hom Diag-Diagonalize TensorDiag-preserves-Diag*
        assoc-naturality [*of* $\lbag a\downarrow \rbag$ $\lbag b\downarrow \rbag$ $\lbag c\downarrow \rbag$] *comp-assoc*
    by *simp*
  also have ... = ($\lbag (\lfloor a \rfloor \ \lfloor\otimes\rfloor\ \lfloor b \rfloor) \Downarrow \lfloor c \rfloor \rbag$ · ($\lbag \lfloor a \rfloor \Downarrow \lfloor b \rfloor \rbag$ ⊗ $\lbag \lfloor c \rfloor \rbag$)) ·
              (($\lbag a\downarrow \rbag$ ⊗ $\lbag b\downarrow \rbag$) ⊗ $\lbag c\downarrow \rbag$)
    using *a b c Diag-Diagonalize Diagonalize-preserves-Ide coherence-key-fact* **by** *simp*
  also have ... = $\lbag \lfloor \mathbf{a}[a,\ b,\ c] \rfloor \rbag$ · $\lbag$*Dom* $\mathbf{a}[a,\ b,\ c]\downarrow\rbag$
    using *a b c red-in-Hom red2-in-Hom TensorDiag-preserves-Diag*
        Diagonalize-preserves-Ide TensorDiag-preserves-Ide Diag-Diagonalize interchange
        eval-red-Tensor TensorDiag-assoc comp-cod-arr [*of* $\lbag c\downarrow \rbag$]
        comp-cod-arr [*of* $\lbag (\lfloor a \rfloor \ \lfloor\otimes\rfloor\ \lfloor b \rfloor) \Downarrow \lfloor c \rfloor \rbag$ · ($\lbag \lfloor a \rfloor \Downarrow \lfloor b \rfloor \rbag$ · ($\lbag a\downarrow \rbag$ ⊗ $\lbag b\downarrow \rbag$) ⊗ $\lbag c\downarrow \rbag$)]
        comp-assoc
    by *simp*
  finally show *?thesis* **by** *blast*
qed

lemma *coherent-Assoc'-Ide*:
assumes *Ide a* **and** *Ide b* **and** *Ide c*
shows *coherent* $\mathbf{a}^{-1}[a,\ b,\ c]$
proof −
  have *Can* $\mathbf{a}[a,\ b,\ c]$ **using** *assms Ide-implies-Can* **by** *simp*
  moreover have $\mathbf{a}^{-1}[a,\ b,\ c]$ = *Inv* $\mathbf{a}[a,\ b,\ c]$
    using *assms Inv-Ide* **by** *simp*
  ultimately show *?thesis*
    using *assms Ide-implies-Can coherent-Assoc-Ide Inv-Ide*

Can-implies-coherent-iff-coherent-Inv
by *metis*
qed

The next lemma implies coherence for the special case of a term that is the tensor of two diagonal arrows.

lemma *eval-red2-naturality*:
assumes *Diag t* **and** *Diag u*
shows $\{\!|Cod\ t \Downarrow Cod\ u|\!\} \cdot (\{\!|t|\!\} \otimes \{\!|u|\!\}) = \{\!|t\ \lfloor\otimes\rfloor\ u|\!\} \cdot \{\!|Dom\ t \Downarrow Dom\ u|\!\}$
proof −
have ∗: $\bigwedge t\ u.\ Diag\ (t \otimes u) \Longrightarrow arr\ \{\!|t|\!\} \wedge arr\ \{\!|u|\!\}$
using *Diag-implies-Arr* **by** *force*
have $t = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis*
using *assms Diag-implies-Arr lunit-naturality* [*of* $\{\!|u|\!\}$]
Arr-implies-Ide-Dom Arr-implies-Ide-Cod comp-cod-arr
by *simp*
moreover have $t \neq \boldsymbol{\mathcal{I}} \wedge u = \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis*
using *assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod*
Diag-implies-Arr Dom-preserves-Diag Cod-preserves-Diag
eval-red2-Diag-Unity runit-naturality [*of* $\{\!|t|\!\}$]
by *simp*
moreover have $t \neq \boldsymbol{\mathcal{I}} \wedge u \neq \boldsymbol{\mathcal{I}} \Longrightarrow$ *?thesis*
using *assms* ∗ *Arr-implies-Ide-Dom Arr-implies-Ide-Cod*
Diag-implies-Arr Dom-preserves-Diag Cod-preserves-Diag
apply (*induct t*, *simp-all*)
proof −
fix *f*
assume *f*: *C.arr f*
assume $u \neq \boldsymbol{\mathcal{I}}$
hence *u*: $u \neq \boldsymbol{\mathcal{I}} \wedge$
$Diag\ u \wedge Diag\ (Dom\ u) \wedge Diag\ (Cod\ u) \wedge Ide\ (Dom\ u) \wedge Ide\ (Cod\ u) \wedge$
$arr\ \{\!|u|\!\} \wedge arr\ \{\!|Dom\ u|\!\} \wedge arr\ \{\!|Cod\ u|\!\} \wedge ide\ \{\!|Dom\ u|\!\} \wedge ide\ \{\!|Cod\ u|\!\}$
using *assms*(2) *Diag-implies-Arr Dom-preserves-Diag Cod-preserves-Diag*
Arr-implies-Ide-Dom Arr-implies-Ide-Cod
by *simp*
hence *1*: $Dom\ u \neq \boldsymbol{\mathcal{I}} \wedge Cod\ u \neq \boldsymbol{\mathcal{I}}$ **using** *u* **by** (*cases u*, *simp-all*)
show $\{\!|\langle C.cod\ f\rangle \Downarrow Cod\ u|\!\} \cdot (V\ f \otimes \{\!|u|\!\}) = (V\ f \otimes \{\!|u|\!\}) \cdot \{\!|\langle C.dom\ f\rangle \Downarrow Dom\ u|\!\}$
using *f u 1 Diag-implies-Arr red2-Diag comp-arr-dom comp-cod-arr* **by** *simp*
next
fix *v w*
assume *I2*: $[\![\ w \neq Unity;\ Diag\ w\ ]\!] \Longrightarrow$
$\{\!|Cod\ w \Downarrow Cod\ u|\!\} \cdot (\{\!|w|\!\} \otimes \{\!|u|\!\}) = \{\!|w\ \lfloor\otimes\rfloor\ u|\!\} \cdot \{\!|Dom\ w \Downarrow Dom\ u|\!\}$
assume $u \neq \boldsymbol{\mathcal{I}}$
hence *u*: $u \neq \boldsymbol{\mathcal{I}} \wedge Arr\ u \wedge Arr\ (Dom\ u) \wedge Arr\ (Cod\ u) \wedge$
$Diag\ u \wedge Diag\ (Dom\ u) \wedge Diag\ (Cod\ u) \wedge Ide\ (Dom\ u) \wedge Ide\ (Cod\ u) \wedge$
$arr\ \{\!|u|\!\} \wedge arr\ \{\!|Dom\ u|\!\} \wedge arr\ \{\!|Cod\ u|\!\} \wedge ide\ \{\!|Dom\ u|\!\} \wedge ide\ \{\!|Cod\ u|\!\}$
using *assms*(2) *Diag-implies-Arr Dom-preserves-Diag Cod-preserves-Diag*
Arr-implies-Ide-Dom Arr-implies-Ide-Cod
by *simp*

assume *vw*: *Diag* $(v \otimes w)$
let *?f* = *un-Prim v*
have $v = \langle ?f \rangle \wedge C.arr\ ?f$
  using *vw* **by** (*metis Diag-TensorE*(*1*) *Diag-TensorE*(*2*))
hence *Arr v* $\wedge$ $v = \langle$*un-Prim v*$\rangle \wedge C.arr\ ?f \wedge$ *Diag v* **by** (*cases v*; *simp*)
hence *v*: $v = \langle ?f \rangle \wedge C.arr\ ?f \wedge$ *Arr v* $\wedge$ *Ide* (*Dom v*) $\wedge$ *Ide* (*Cod v*) $\wedge$ *Diag v* $\wedge$
        Diag (*Dom v*) $\wedge$ *arr* $\lVert v \rVert \wedge$ *arr* $\lVert$*Dom v*$\rVert \wedge$ *arr* $\lVert$*Cod v*$\rVert \wedge$
        ide $\lVert$*Dom v*$\rVert \wedge$ *ide* $\lVert$*Cod v*$\rVert$
  by (*cases v*, *simp-all*)
have *Diag w* $\wedge$ $w \neq \boldsymbol{\mathcal{I}}$
  using *vw v* **by** (*metis Diag.simps*(*3*))
hence *w*: $w \neq \boldsymbol{\mathcal{I}} \wedge$ *Arr w* $\wedge$ *Arr* (*Dom w*) $\wedge$ *Arr* (*Cod w*) $\wedge$
        Diag w $\wedge$ *Diag* (*Dom w*) $\wedge$ *Diag* (*Cod w*) $\wedge$
        Ide (*Dom w*) $\wedge$ *Ide* (*Cod w*) $\wedge$
        arr $\lVert w \rVert \wedge$ *arr* $\lVert$*Dom w*$\rVert \wedge$ *arr* $\lVert$*Cod w*$\rVert \wedge$ *ide* $\lVert$*Dom w*$\rVert \wedge$ *ide* $\lVert$*Cod w*$\rVert$
using *vw* $*$ *Diag-implies-Arr Dom-preserves-Diag Cod-preserves-Diag Arr-implies-Ide-Dom*
      Arr-implies-Ide-Cod ide-eval-Ide Ide-implies-Arr Ide-in-Hom
  by *simp*
show $\lVert$(*Cod v* $\otimes$ *Cod w*) $\Downarrow$ *Cod u*$\rVert \cdot$ (($\lVert v \rVert \otimes \lVert w \rVert$) $\otimes \lVert u \rVert$)
      = $\lVert$(*v* $\otimes$ *w*) $\lfloor\otimes\rfloor$ *u*$\rVert \cdot \lVert$(*Dom v* $\otimes$ *Dom w*) $\Downarrow$ *Dom u*$\rVert$
proof −
  have *u′*: *Dom u* $\neq \boldsymbol{\mathcal{I}} \wedge$ *Cod u* $\neq \boldsymbol{\mathcal{I}}$ **using** *u* **by** (*cases u*) *simp-all*
  have *w′*: *Dom w* $\neq \boldsymbol{\mathcal{I}} \wedge$ *Cod w* $\neq \boldsymbol{\mathcal{I}}$ **using** *w* **by** (*cases w*) *simp-all*
  have *D*: *Diag* (*Dom v* $\otimes$ (*Dom w* $\lfloor\otimes\rfloor$ *Dom u*))
  proof −
    have *Dom w* $\lfloor\otimes\rfloor$ *Dom u* $\neq \boldsymbol{\mathcal{I}}$
      using *u u′ w w′ not-is-Tensor-TensorDiagE* **by** *blast*
    moreover have *Diag* (*Dom w* $\lfloor\otimes\rfloor$ *Dom u*)
      using *u w TensorDiag-preserves-Diag* **by** *simp*
    moreover have *Dom v* = $\langle$*C.dom ?f*$\rangle$
      using *v* **by** (*cases v*, *simp-all*)
    ultimately show *?thesis*
      using *u v w TensorDiag-preserves-Diag* **by** *auto*
  qed
  have *C*: *Diag* (*Cod v* $\otimes$ (*Cod w* $\lfloor\otimes\rfloor$ *Cod u*))
  proof −
    have *Cod w* $\lfloor\otimes\rfloor$ *Cod u* $\neq \boldsymbol{\mathcal{I}}$
      using *u u′ w w′ not-is-Tensor-TensorDiagE* **by** *blast*
    moreover have *Diag* (*Cod w* $\lfloor\otimes\rfloor$ *Cod u*)
      using *u w TensorDiag-preserves-Diag* **by** *simp*
    moreover have *Cod v* = $\langle$*C.cod ?f*$\rangle$
      using *v* **by** (*cases v*, *simp-all*)
    ultimately show *?thesis*
      using *u v w* **by** (*cases Cod w* $\lfloor\otimes\rfloor$ *Cod u*) *simp-all*
  qed
  have $\lVert$(*Cod v* $\otimes$ *Cod w*) $\Downarrow$ *Cod u*$\rVert \cdot$ (($\lVert v \rVert \otimes \lVert w \rVert$) $\otimes \lVert u \rVert$)
      = ($\lVert$*Cod v* $\Downarrow$ (*Cod w* $\lfloor\otimes\rfloor$ *Cod u*)$\rVert \cdot$ ($\lVert$*Cod v*$\rVert \otimes \lVert$*Cod w* $\Downarrow$ *Cod u*$\rVert$) $\cdot$
        a[$\lVert$*Cod v*$\rVert$, $\lVert$*Cod w*$\rVert$, $\lVert$*Cod u*$\rVert$]) $\cdot$ (($\lVert v \rVert \otimes \lVert w \rVert$) $\otimes \lVert u \rVert$)
  proof −

```
  have (Cod v ⊗ Cod w) ⇓ Cod u
         = (Cod v ⇓ (Cod w ⌊⊗⌋ ⌊Cod u⌋)) ⋆ (Cod v ⊗ Cod w ⇓ Cod u) ⋆
           a[Cod v, Cod w, Cod u]
    using u v w by (cases u, simp-all)
  hence ⦇(Cod v ⊗ Cod w) ⇓ Cod u⦈
         = ⦇Cod v ⇓ (Cod w ⌊⊗⌋ Cod u)⦈ ⋆ (⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈) ⋆
           a[⦇Cod v⦈, ⦇Cod w⦈, ⦇Cod u⦈]
    using u v w by simp
  thus ?thesis by argo
qed
also have ... = ((⦇Cod v⦈ ⊗ ⦇Cod w ⌊⊗⌋ Cod u⦈) ⋆ (⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈) ⋆
                  a[⦇Cod v⦈, ⦇Cod w⦈, ⦇Cod u⦈]) ⋆ ((⦇v⦈ ⊗ ⦇w⦈) ⊗ ⦇u⦈)
  using u v w C red2-Diag by simp
also have ... = ((⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈) ⋆ a[⦇Cod v⦈, ⦇Cod w⦈, ⦇Cod u⦈]) ⋆
                  ((⦇v⦈ ⊗ ⦇w⦈) ⊗ ⦇u⦈)
proof −
  have (⦇Cod v⦈ ⊗ ⦇Cod w ⌊⊗⌋ Cod u⦈) ⋆ (⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈)
         = ⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈
    using u v w comp-cod-arr red2-in-Hom by simp
  moreover have
      seq (⦇Cod v⦈ ⊗ ⦇Cod w ⌊⊗⌋ Cod u⦈) (⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈)
    using u v w red2-in-Hom TensorDiag-in-Hom Ide-in-Hom by simp
  moreover have seq (⦇Cod v⦈ ⊗ ⦇Cod w ⇓ Cod u⦈) a[⦇Cod v⦈, ⦇Cod w⦈, ⦇Cod u⦈]
    using u v w red2-in-Hom by simp
  ultimately show ?thesis
    using u v w comp-reduce by presburger
qed
also have
  ... = (⦇v⦈ ⊗ ⦇w ⌊⊗⌋ u⦈) ⋆ ⦇Dom w ⇓ Dom u⦈) ⋆ a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
  using u v w I2 red2-in-Hom TensorDiag-in-Hom interchange comp-reduce
        assoc-naturality [of ⦇v⦈ ⦇w⦈ ⦇u⦈] comp-cod-arr comp-assoc
  by simp
also have ... = (⦇v⦈ ⊗ ⦇w ⌊⊗⌋ u⦈) ⋆ (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ⋆
                  a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
  using u v w red2-in-Hom TensorDiag-in-Hom interchange comp-reduce comp-arr-dom
  by simp
also have ... = ⦇v ⌊⊗⌋ w ⌊⊗⌋ u⦈ ⋆ (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ⋆
                  a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
  using u u′ v w not-is-Tensor-TensorDiagE TensorDiag-Prim [of w ⌊⊗⌋ u ?f]
  by force
also have ... = ⦇v ⌊⊗⌋ w ⌊⊗⌋ u⦈ ⋆ ⦇Dom v ⌊⊗⌋ Dom w ⌊⊗⌋ Dom u⦈ ⋆
                (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ⋆ a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
proof −
  have
    ⦇v ⌊⊗⌋ w ⌊⊗⌋ u⦈ ⋆ (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ⋆
      a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈] =
    (⦇v ⌊⊗⌋ w ⌊⊗⌋ u⦈ ⋆ ⦇Dom v ⌊⊗⌋ Dom w ⌊⊗⌋ Dom u⦈) ⋆
    (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ⋆ a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
    using u v w comp-arr-dom TensorDiag-in-Hom TensorDiag-preserves-Diag by simp
```

```
      also have ... = ⦇v ⌊⊗⌋ w ⌊⊗⌋ u⦈ · ⦇Dom v ⌊⊗⌋ Dom w ⌊⊗⌋ Dom u⦈ ·
                      (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) · a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
        using comp-assoc by simp
      finally show ?thesis by blast
    qed
    also have ... = ⦇(v ⌊⊗⌋ w) ⌊⊗⌋ u⦈ · ⦇(Dom v ⊗ Dom w) ⇓ Dom u⦈
    proof −
      have
        ⦇(Dom v ⊗ Dom w) ⇓ Dom u⦈
            = ⦇Dom v ⇓ (Dom w ⌊⊗⌋ Dom u)⦈ · (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ·
              a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
      proof −
        have (Dom v ⊗ Dom w) ⇓ Dom u
              = (Dom v ⇓ (Dom w ⌊⊗⌋ ⌊Dom u⌋)) ⋆ (Dom v ⊗ (Dom w ⇓ Dom u)) ⋆
                a[Dom v, Dom w, Dom u]
          using u u′ v w red2-in-Hom TensorDiag-in-Hom Ide-in-Hom
          by (cases u) auto
        thus ?thesis
          using u v w red2-in-Hom by simp
      qed
      also have
        ... = ⦇Dom v ⌊⊗⌋ Dom w ⌊⊗⌋ Dom u⦈ · (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ·
                    a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
        using D TensorDiag-Diag red2-Diag by simp
      finally have
        ⦇(Dom v ⊗ Dom w) ⇓ Dom u⦈
            = ⦇Dom v ⌊⊗⌋ Dom w ⌊⊗⌋ Dom u⦈ · (⦇Dom v⦈ ⊗ ⦇Dom w ⇓ Dom u⦈) ·
              a[⦇Dom v⦈, ⦇Dom w⦈, ⦇Dom u⦈]
        by blast
      thus ?thesis
        using assms v w TensorDiag-assoc by auto
    qed
    finally show ?thesis
      using vw TensorDiag-Diag by simp
  qed
 qed
 ultimately show ?thesis by blast
qed

lemma Tensor-preserves-coherent:
assumes Arr t and Arr u and coherent t and coherent u
shows coherent (t ⊗ u)
proof −
 have t: Arr t ∧ Ide (Dom t) ∧ Ide (Cod t) ∧ Ide ⌊Dom t⌋ ∧ Ide ⌊Cod t⌋ ∧
         arr ⦇t⦈ ∧ arr ⦇Dom t⦈ ∧ ide ⦇Dom t⦈ ∧ arr ⦇Cod t⦈ ∧ ide ⦇Cod t⦈
   using assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diagonalize-preserves-Ide
   by auto
 have u: Arr u ∧ Ide (Dom u) ∧ Ide (Cod u) ∧ Ide ⌊Dom u⌋ ∧ Ide ⌊Cod u⌋ ∧
         arr ⦇u⦈ ∧ arr ⦇Dom u⦈ ∧ ide ⦇Dom u⦈ ∧ arr ⦇Cod u⦈ ∧ ide ⦇Cod u⦈
```

```
    using assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diagonalize-preserves-Ide
    by auto
  have ⦇Cod (t ⊗ u)↓⦈ · (⦇t⦈ ⊗ ⦇u⦈)
        = (⦇⌊Cod t⌋ ⇓ ⌊Cod u⌋⦈ · (⦇Cod t↓⦈ ⊗ ⦇Cod u↓⦈)) · (⦇t⦈ ⊗ ⦇u⦈)
    using t u eval-red-Tensor by simp
  also have ... = ⦇⌊Cod t⌋ ⇓ ⌊Cod u⌋⦈ · (⦇Cod t↓⦈ ⊗ ⦇Cod u↓⦈) · (⦇t⦈ ⊗ ⦇u⦈)
    using comp-assoc by simp
  also have ... = ⦇⌊Cod t⌋ ⇓ ⌊Cod u⌋⦈ · (⦇⌊t⌋⦈ ⊗ ⦇⌊u⌋⦈) · (⦇Dom t↓⦈ ⊗ ⦇Dom u↓⦈)
    using assms t u Diagonalize-in-Hom red-in-Hom interchange by simp
  also have ... = (⦇⌊Cod t⌋ ⇓ ⌊Cod u⌋⦈ · (⦇⌊t⌋⦈ ⊗ ⦇⌊u⌋⦈)) · (⦇Dom t↓⦈ ⊗ ⦇Dom u↓⦈)
    using comp-assoc by simp
  also have ... = (⦇⌊t⌋ ⌊⊗⌋ ⌊u⌋⦈ · ⦇⌊Dom t⌋ ⇓ ⌊Dom u⌋⦈) · (⦇Dom t↓⦈ ⊗ ⦇Dom u↓⦈)
    using assms t u Diag-Diagonalize Diagonalize-in-Hom
          eval-red2-naturality [of Diagonalize t Diagonalize u]
    by simp
  also have ... = ⦇⌊t⌋ ⌊⊗⌋ ⌊u⌋⦈ · ⦇⌊Dom t⌋ ⇓ ⌊Dom u⌋⦈ · (⦇Dom t↓⦈ ⊗ ⦇Dom u↓⦈)
    using comp-assoc by simp
  also have ... = ⦇⌊t⌋ ⌊⊗⌋ ⌊u⌋⦈ · ⦇(Dom t ⊗ Dom u)↓⦈
    using t u eval-red-Tensor by simp
  finally have ⦇Cod (t ⊗ u)↓⦈ · (⦇t⦈ ⊗ ⦇u⦈) = ⦇⌊t⌋ ⌊⊗⌋ ⌊u⌋⦈ · ⦇(Dom t ⊗ Dom u)↓⦈
    by blast
  thus ?thesis using t u by simp
qed

lemma Comp-preserves-coherent:
assumes Arr t and Arr u and Dom t = Cod u
and coherent t and coherent u
shows coherent (t ⋆ u)
proof −
  have t: Arr t ∧ Ide (Dom t) ∧ Ide (Cod t) ∧ Ide ⌊Dom t⌋ ∧ Ide ⌊Cod t⌋ ∧
          arr ⦇t⦈ ∧ arr ⦇Dom t⦈ ∧ ide ⦇Dom t⦈ ∧ arr ⦇Cod t⦈ ∧ ide ⦇Cod t⦈
    using assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diagonalize-preserves-Ide
    by auto
  have u: Arr u ∧ Ide (Dom u) ∧ Ide (Cod u) ∧ Ide ⌊Dom u⌋ ∧ Ide ⌊Cod u⌋ ∧
          arr ⦇u⦈ ∧ arr ⦇Dom u⦈ ∧ ide ⦇Dom u⦈ ∧ arr ⦇Cod u⦈ ∧ ide ⦇Cod u⦈
    using assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diagonalize-preserves-Ide
    by auto
  have ⦇Cod (t ⋆ u)↓⦈ · ⦇t ⋆ u⦈ = ⦇Cod t↓⦈ · ⦇t⦈ · ⦇u⦈
    using t u by simp
  also have ... = (⦇Cod t↓⦈ · ⦇t⦈) · ⦇u⦈
  proof −

    have seq ⦇Cod t↓⦈ ⦇t⦈
      using assms t red-in-Hom by (intro seqI, auto)
    moreover have seq ⦇t⦈ ⦇u⦈
      using assms t u by auto
    ultimately show ?thesis using comp-assoc by auto
  qed
  also have ... = ⦇⌊t ⋆ u⌋⦈ · ⦇Dom (t ⋆ u)↓⦈
```

```
    using t u assms red-in-Hom Diag-Diagonalize comp-assoc
    by (simp add: Diag-implies-Arr eval-CompDiag)
  finally show coherent (t ⋆ u) by blast
qed
```

The main result: "Every formal arrow is coherent."

```
theorem coherence:
assumes Arr t
shows coherent t
proof −
  have Arr t ⟹ coherent t
  proof (induct t)
    fix u v
    show ⟦ Arr u ⟹ coherent u; Arr v ⟹ coherent v ⟧ ⟹ Arr (u ⊗ v)
           ⟹ coherent (u ⊗ v)
      using Tensor-preserves-coherent by simp
    show ⟦ Arr u ⟹ coherent u; Arr v ⟹ coherent v ⟧ ⟹ Arr (u ⋆ v)
           ⟹ coherent (u ⋆ v)
      using Comp-preserves-coherent by simp
    next
    show coherent 𝓘 by simp
    fix f
    show Arr ⟨f⟩ ⟹ coherent ⟨f⟩ by simp
    next
    fix t
    assume I: Arr t ⟹ coherent t
    show Lunit: Arr 𝗹[t] ⟹ coherent 𝗹[t]
      using I Arr-implies-Ide-Dom coherent-Lunit-Ide Ide-in-Hom Ide-implies-Arr
            Comp-preserves-coherent [of t 𝗹[Dom t]] Diagonalize-Comp-Arr-Dom ι-ide-simp
      by auto
    show Runit: Arr 𝗿[t] ⟹ coherent 𝗿[t]
      using I Arr-implies-Ide-Dom coherent-Runit-Ide Ide-in-Hom Ide-implies-Arr
            Comp-preserves-coherent [of t 𝗿[Dom t]] Diagonalize-Comp-Arr-Dom ϱ-ide-simp
      by auto
    show Arr 𝗹⁻¹[t] ⟹ coherent 𝗹⁻¹[t]
    proof −
      assume Arr 𝗹⁻¹[t]
      hence t: Arr t by simp
      have coherent (𝗹⁻¹[Cod t] ⋆ t)
        using t I Arr-implies-Ide-Cod coherent-Lunit'-Ide Ide-in-Hom
              Comp-preserves-coherent [of 𝗹⁻¹[Cod t] t]
        by fastforce
      thus ?thesis
        using t Arr-implies-Ide-Cod Ide-implies-Arr Ide-in-Hom Diagonalize-Comp-Cod-Arr
              eval-in-hom ι'.is-natural-2 [of ⦇t⦈]
        by force
    qed
    show Arr 𝗿⁻¹[t] ⟹ coherent 𝗿⁻¹[t]
    proof −
```

```
  assume Arr r⁻¹[t]
  hence t: Arr t by simp
  have coherent (r⁻¹[Cod t] · t)
    using t I Arr-implies-Ide-Cod coherent-Runit'-Ide Ide-in-Hom
          Comp-preserves-coherent [of r⁻¹[Cod t] t]
    by fastforce
  thus ?thesis
    using t Arr-implies-Ide-Cod Ide-implies-Arr Ide-in-Hom Diagonalize-Comp-Cod-Arr
          eval-in-hom ϱ'.is-natural-2 [of ⦇t⦈]
    by force
qed
next
fix t u v
assume I1: Arr t ⟹ coherent t
assume I2: Arr u ⟹ coherent u
assume I3: Arr v ⟹ coherent v
show Arr a[t, u, v] ⟹ coherent a[t, u, v]
proof −
  assume tuv: Arr a[t, u, v]
  have t: Arr t using tuv by simp
  have u: Arr u using tuv by simp
  have v: Arr v using tuv by simp
  have coherent ((t ⊗ u ⊗ v) · a[Dom t, Dom u, Dom v])
  proof −
    have Arr (t ⊗ u ⊗ v) ∧ coherent (t ⊗ u ⊗ v)
    proof
      have 1: Arr t ∧ coherent t using t I1 by simp
      have 2: Arr (u ⊗ v) ∧ coherent (u ⊗ v)
        using u v I2 I3 Tensor-preserves-coherent by force
      show Arr (t ⊗ u ⊗ v)  using 1 2 by simp
      show coherent (t ⊗ u ⊗ v)
        using 1 2 Tensor-preserves-coherent by blast
    qed
    moreover have Arr a[Dom t, Dom u, Dom v]
      using t u v Arr-implies-Ide-Dom by simp
    moreover have coherent a[Dom t, Dom u, Dom v]
      using t u v Arr-implies-Ide-Dom coherent-Assoc-Ide by blast
    moreover have Dom (t ⊗ u ⊗ v) = Cod a[Dom t, Dom u, Dom v]
      using t u v Arr-implies-Ide-Dom Ide-in-Hom by simp
    ultimately show ?thesis
      using t u v Arr-implies-Ide-Dom Ide-implies-Arr
            Comp-preserves-coherent [of t ⊗ u ⊗ v a[Dom t, Dom u, Dom v]]
      by blast
  qed
  moreover have Par a[t, u, v] ((t ⊗ u ⊗ v) · a[Dom t, Dom u, Dom v])
    using t u v Arr-implies-Ide-Dom Ide-implies-Arr Ide-in-Hom by simp
  moreover have ⌊a[t, u, v]⌋ = ⌊(t ⊗ u ⊗ v) · a[Dom t, Dom u, Dom v]⌋
  proof −
    have (⌊t⌋ ⌊⊗⌋ ⌊u⌋) ⌊⊗⌋ ⌊v⌋
```

$= (\lfloor t \rfloor \ \lfloor \otimes \rfloor \ \lfloor u \rfloor \ \lfloor \otimes \rfloor \ \lfloor v \rfloor) \ \lfloor \cdot \rfloor \ ((\lfloor Dom\ t \rfloor \ \lfloor \otimes \rfloor \ \lfloor Dom\ u \rfloor) \ \lfloor \otimes \rfloor \ \lfloor Dom\ v \rfloor)$
proof −
have *1*: $Diag\ \lfloor t \rfloor \wedge Diag\ \lfloor u \rfloor \wedge Diag\ \lfloor v \rfloor \wedge$
$Dom\ \lfloor t \rfloor = \lfloor Dom\ t \rfloor \wedge Dom\ \lfloor u \rfloor = \lfloor Dom\ u \rfloor \wedge Dom\ \lfloor v \rfloor = \lfloor Dom\ v \rfloor$
using *t u v Diag-Diagonalize* **by** *blast*
moreover have $Diag\ (\lfloor t \rfloor \ \lfloor \otimes \rfloor \ \lfloor u \rfloor)$
using *1 TensorDiag-preserves-Diag(1)* **by** *blast*
moreover have $\bigwedge t.\ Arr\ t \Longrightarrow \lfloor t \rfloor \ \lfloor \cdot \rfloor \ \lfloor Dom\ t \rfloor = \lfloor t \rfloor$
using *t Diagonalize-Comp-Arr-Dom* **by** *simp*
moreover have $Dom\ \lfloor \mathbf{a}[t,\ u,\ v] \rfloor = \lfloor Dom\ \mathbf{a}[t,\ u,\ v] \rfloor$
using *Diag-Diagonalize tuv* **by** *blast*
ultimately show *?thesis*
using *t u v tuv 1 TensorDiag-assoc TensorDiag-preserves-Diag(2)*
by (*metis* (*no-types*) *Diagonalize.simps*(*9*))
qed
thus *?thesis*
using *t u v Diagonalize-Comp-Arr-Dom CompDiag-TensorDiag Diag-Diagonalize*
by *simp*
qed
moreover have $\{\!|\mathbf{a}[t,\ u,\ v]|\!\} = \{\!|(t \otimes u \otimes v) \cdot \mathbf{a}[Dom\ t,\ Dom\ u,\ Dom\ v]|\!\}$
using *t u v Arr-implies-Ide-Dom Ide-implies-Arr α-simp* [*of* $\{\!|t|\!\}$ $\{\!|u|\!\}$ $\{\!|v|\!\}$]
by *simp*
ultimately show $coherent\ \mathbf{a}[t,\ u,\ v]$ **by** *argo*
qed
show $Arr\ \mathbf{a}^{-1}[t,\ u,\ v] \Longrightarrow coherent\ \mathbf{a}^{-1}[t,\ u,\ v]$
proof −
assume *tuv*: $Arr\ \mathbf{a}^{-1}[t,\ u,\ v]$
have *t*: $Arr\ t$ **using** *tuv* **by** *simp*
have *u*: $Arr\ u$ **using** *tuv* **by** *simp*
have *v*: $Arr\ v$ **using** *tuv* **by** *simp*
have $coherent\ (((t \otimes u) \otimes v) \cdot \mathbf{a}^{-1}[Dom\ t,\ Dom\ u,\ Dom\ v])$
proof −
have $Arr\ ((t \otimes u) \otimes v) \wedge coherent\ ((t \otimes u) \otimes v)$
proof
have *1*: $Arr\ v \wedge coherent\ v$ **using** *v I3* **by** *simp*
have *2*: $Arr\ (t \otimes u) \wedge coherent\ (t \otimes u)$
using *t u I1 I2 Tensor-preserves-coherent* **by** *force*
show $Arr\ ((t \otimes u) \otimes v)$ **using** *1 2* **by** *simp*
show $coherent\ ((t \otimes u) \otimes v)$
using *1 2 Tensor-preserves-coherent* **by** *blast*
qed
moreover have $Arr\ \mathbf{a}^{-1}[Dom\ t,\ Dom\ u,\ Dom\ v]$
using *t u v Arr-implies-Ide-Dom* **by** *simp*
moreover have $coherent\ \mathbf{a}^{-1}[Dom\ t,\ Dom\ u,\ Dom\ v]$
using *t u v Arr-implies-Ide-Dom coherent-Assoc'-Ide* **by** *blast*
moreover have $Dom\ ((t \otimes u) \otimes v) = Cod\ \mathbf{a}^{-1}[Dom\ t,\ Dom\ u,\ Dom\ v]$
using *t u v Arr-implies-Ide-Dom Ide-in-Hom* **by** *simp*
ultimately show *?thesis*
using *t u v Arr-implies-Ide-Dom Ide-implies-Arr*

```
            Comp-preserves-coherent [of ((t ⊗ u) ⊗ v) a⁻¹[Dom t, Dom u, Dom v]]
        by metis
      qed
      moreover have Par a⁻¹[t, u, v] (((t ⊗ u) ⊗ v) · a⁻¹[Dom t, Dom u, Dom v])
        using t u v Arr-implies-Ide-Dom Ide-implies-Arr Ide-in-Hom by simp
      moreover have ⌊a⁻¹[t, u, v]⌋ = ⌊((t ⊗ u) ⊗ v) · a⁻¹[Dom t, Dom u, Dom v]⌋
        using t u v Diagonalize-Comp-Arr-Dom CompDiag-TensorDiag Diag-Diagonalize
            TensorDiag-assoc TensorDiag-preserves-Diag TensorDiag-in-Hom
            CompDiag-Diag-Dom [of (⌊t⌋ ⌊⊗⌋ ⌊u⌋) ⌊⊗⌋ ⌊v⌋]
        by simp
      moreover have {|a⁻¹[t, u, v]|} = {|((t ⊗ u) ⊗ v) · a⁻¹[Dom t, Dom u, Dom v]|}
        using t u v Arr-implies-Ide-Dom Ide-implies-Arr eval-in-hom comp-cod-arr
            α'.is-natural-1 α'-simp
        by simp
      ultimately show coherent a⁻¹[t, u, v] by argo
    qed
  qed
  thus ?thesis using assms by blast
qed
```

MacLane [5] says: "A coherence theorem asserts 'Every diagram commutes'," but that is somewhat misleading. A coherence theorem provides some kind of hopefully useful way of distinguishing diagrams that definitely commute from diagrams that might not. The next result expresses coherence for monoidal categories in this way. As the hypotheses can be verified algorithmically (using the functions *Dom*, *Cod*, *Arr*, and *Diagonalize*) if we are given an oracle for equality of arrows in C, the result provides a decision procedure, relative to C, for the word problem for the free monoidal category generated by C.

```
corollary eval-eqI:
assumes Par t u and ⌊t⌋ = ⌊u⌋
shows {|t|} = {|u|}
  using assms coherence canonical-factorization by simp
```

Our final corollary expresses coherence in a more "MacLane-like" fashion: parallel canonical arrows are equivalent under evaluation.

```
corollary maclane-coherence:
assumes Par t u and Can t and Can u
shows {|t|} = {|u|}
proof (intro eval-eqI)
  show Par t u by fact
  show ⌊t⌋ = ⌊u⌋
  proof −
    have Ide ⌊t⌋ ∧ Ide ⌊u⌋ ∧ Par ⌊t⌋ ⌊u⌋
      using assms eval-eqI Ide-Diagonalize-Can Diagonalize-in-Hom by simp
    thus ?thesis using Ide-in-Hom by auto
  qed
qed

end
```

end

Chapter 3

Monoidal Functor

theory *MonoidalFunctor*
imports *MonoidalCategory*
begin

A monoidal functor is a functor F between monoidal categories C and D that preserves the monoidal structure up to isomorphism. The traditional definition assumes a monoidal functor to be equipped with two natural isomorphisms, a natural isomorphism φ that expresses the preservation of tensor product and a natural isomorphism ψ that expresses the preservation of the unit object. These natural isomorphisms are subject to coherence conditions; the condition for φ involving the associator and the conditions for ψ involving the unitors. However, as pointed out in [2] (Section 2.4), it is not necessary to take the natural isomorphism ψ as given, since the mere assumption that $F\ \mathcal{I}_C$ is isomorphic to $\mathcal{I}_D$ is sufficient for there to be a canonical definition of ψ from which the coherence conditions can be derived. This leads to a more economical definition of monoidal functor, which is the one we adopt here.

locale *monoidal-functor* =
  C: *monoidal-category* $C\ T_C\ \alpha_C\ \iota_C$ +
  D: *monoidal-category* $D\ T_D\ \alpha_D\ \iota_D$ +
  functor $C\ D\ F$ +
  CC: *product-category* $C\ C$ +
  DD: *product-category* $D\ D$ +
  FF: *product-functor* $C\ C\ D\ D\ F\ F$ +
  FoT_C: *composite-functor* $C.CC.comp\ C\ D\ T_C\ F$ +
  $T_D oFF$: *composite-functor* $C.CC.comp\ D.CC.comp\ D\ FF.map\ T_D$ +
  φ: *natural-isomorphism* $C.CC.comp\ D\ T_D oFF.map\ FoT_C.map\ \varphi$
for $C :: 'c\ comp$ (**infixr** $\cdot_C$ *55*)
and $T_C :: 'c * 'c \Rightarrow 'c$
and $\alpha_C :: 'c * 'c * 'c \Rightarrow 'c$
and $\iota_C :: 'c$
and $D :: 'd\ comp$ (**infixr** $\cdot_D$ *55*)
and $T_D :: 'd * 'd \Rightarrow 'd$
and $\alpha_D :: 'd * 'd * 'd \Rightarrow 'd$
and $\iota_D :: 'd$

and $F :: 'c \Rightarrow 'd$
and $\varphi :: 'c * 'c \Rightarrow 'd$ +
assumes *preserves-unity*: *D.isomorphic D.unity* (*F C.unity*)
and *assoc-coherence*:
$[\![C.ide\ a;\ C.ide\ b;\ C.ide\ c]\!] \Longrightarrow$
$F\ (\alpha_C\ (a,\ b,\ c))\ \cdot_D\ \varphi\ (T_C\ (a,\ b),\ c)\ \cdot_D\ T_D\ (\varphi\ (a,\ b),\ F\ c)$
$= \varphi\ (a,\ T_C\ (b,\ c))\ \cdot_D\ T_D\ (F\ a,\ \varphi\ (b,\ c))\ \cdot_D\ \alpha_D\ (F\ a,\ F\ b,\ F\ c)$
begin

notation *C.tensor* (**infixr** $\otimes_C$ *53*)
and *C.unity* ($\mathcal{I}_C$)
and *C.lunit* ($\mathrm{l}_C[\text{-}]$)
and *C.runit* ($\mathrm{r}_C[\text{-}]$)
and *C.assoc* ($\mathrm{a}_C[\text{-}, \text{-}, \text{-}]$)
and *D.tensor* (**infixr** $\otimes_D$ *53*)
and *D.unity* ($\mathcal{I}_D$)
and *D.lunit* ($\mathrm{l}_D[\text{-}]$)
and *D.runit* ($\mathrm{r}_D[\text{-}]$)
and *D.assoc* ($\mathrm{a}_D[\text{-}, \text{-}, \text{-}]$)

lemma *φ-in-hom*:
assumes *C.ide a* **and** *C.ide b*
shows «$\varphi\ (a,\ b) : F\ a \otimes_D F\ b \rightarrow_D F\ (a \otimes_C b)$»
using *assms* **by** *auto*

We wish to exhibit a canonical definition of an isomorphism $\psi \in D.hom\ \mathcal{I}_D\ (F\ \mathcal{I}_C)$ that satisfies certain coherence conditions that involve the left and right unitors. In [2], the isomorphism ψ is defined by the equation $\mathrm{l}_D[F\ \mathcal{I}_C] = F\ \mathrm{l}_C[\mathcal{I}_C]\ \cdot_D\ \varphi\ (\mathcal{I}_C,\ \mathcal{I}_C)\ \cdot_D\ (\psi \otimes_D F\ \mathcal{I}_C)$, which suffices for the definition because the functor $- \otimes_D F\ \mathcal{I}_C$ is fully faithful. It is then asserted (Proposition 2.4.3) that the coherence condition $\mathrm{l}_D[F\ a] = F\ \mathrm{l}_C[a]\ \cdot_D\ \varphi\ (\mathcal{I}_C,\ a)\ \cdot_D\ (\psi \otimes_D F\ a)$ is satisfied for any object a of C, as well as the corresponding condition for the right unitor. However, the proof is left as an exercise (Exercise 2.4.4). The organization of the presentation suggests that that one should derive the general coherence condition from the special case $\mathrm{l}_D[F\ \mathcal{I}_C] = F\ \mathrm{l}_C[\mathcal{I}_C]\ \cdot_D\ \varphi\ (\mathcal{I}_C,\ \mathcal{I}_C)\ \cdot_D\ (\psi \otimes_D F\ \mathcal{I}_C)$ used as the definition of ψ. However, I did not see how to do it that way, so I used a different approach. The isomorphism $\iota_D' \equiv F\ \iota_C\ \cdot_D\ \varphi\ (\mathcal{I}_C,\ \mathcal{I}_C)$ serves as an alternative unit for the monoidal category D. There is consequently a unique isomorphism that maps ι_D to ι_D'. We define ψ to be this isomorphism and then use the definition to establish the desired coherence conditions.

abbreviation ι_1
where $\iota_1 \equiv F\ \iota_C\ \cdot_D\ \varphi\ (\mathcal{I}_C,\ \mathcal{I}_C)$

lemma *ι₁-in-hom*:
shows «$\iota_1 : F\ \mathcal{I}_C \otimes_D F\ \mathcal{I}_C \rightarrow_D F\ \mathcal{I}_C$»
using *C.unit-in-hom* **by** (*intro D.in-homI*, *auto*)

lemma *ι₁-is-iso*:
shows *D.iso* ι_1

using *C.unit-is-iso C.unit-in-hom* φ*-in-hom D.isos-compose* **by** *auto*

interpretation *D*: *monoidal-category-with-alternate-unit D* T_D α_D ι_D ι_1
proof −
have *1*: $\exists \psi.$ «$\psi : F\ \mathcal{I}_C \to_D \mathcal{I}_D$» $\wedge$ *D.iso* ψ
proof −
obtain ψ' **where** ψ': «$\psi' : \mathcal{I}_D \to_D F\ \mathcal{I}_C$» $\wedge$ *D.iso* ψ'
using *preserves-unity* **by** *auto*
have «*D.inv* $\psi' : F\ \mathcal{I}_C \to_D \mathcal{I}_D$» $\wedge$ *D.iso* (*D.inv* ψ')
using ψ' **by** *simp*
thus *?thesis* **by** *auto*
qed
obtain ψ **where** ψ: «$\psi : F\ \mathcal{I}_C \to_D \mathcal{I}_D$» $\wedge$ *D.iso* ψ
using *1* **by** *blast*
interpret *L*: *equivalence-functor D D* ‹$\lambda f.\ (D.cod\ \iota_1) \otimes_D f$›
proof −
interpret *L*: *functor D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$›
using *D.T.fixing-ide-gives-functor-1* **by** *simp*
interpret *L*: *endofunctor D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$› **..**
interpret ψx: *natural-transformation D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$› ‹$\lambda f.\ \mathcal{I}_D \otimes_D f$›
‹$\lambda f.\ \psi \otimes_D f$›
using ψ *D.T.fixing-arr-gives-natural-transformation-1* [*of* ψ] **by** *auto*
interpret ψx: *natural-isomorphism D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$› ‹$\lambda f.\ \mathcal{I}_D \otimes_D f$› ‹$\lambda f.\ \psi \otimes_D f$›
apply *unfold-locales* **using** ψ *D.tensor-preserves-iso* **by** *simp*
interpret $\iota_D o \psi x$: *vertical-composite D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$› ‹$\lambda f.\ \mathcal{I}_D \otimes_D f$› *D.map*
‹$\lambda f.\ \psi \otimes_D f$› *D.*$\iota$ **..**
interpret $\iota_D o \psi x$: *natural-isomorphism D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$› *D.map* $\iota_D o \psi x$*.map*
using ψx*.natural-isomorphism-axioms D.*ι*.natural-isomorphism-axioms*
natural-isomorphisms-compose **by** *blast*
interpret *L*: *equivalence-functor D D* ‹$\lambda f.\ (F\ \mathcal{I}_C) \otimes_D f$›
using *L.isomorphic-to-identity-is-equivalence* $\iota_D o \psi x$*.natural-isomorphism-axioms*
by *simp*
show *equivalence-functor D D* ($\lambda f.\ (D.cod\ \iota_1) \otimes_D f$)
using *L.equivalence-functor-axioms C.unit-in-hom* **by** *auto*
qed
interpret *R*: *equivalence-functor D D* ‹$\lambda f.\ T_D\ (f,\ D.cod\ \iota_1)$›
proof −
interpret *R*: *functor D D* ‹$\lambda f.\ T_D\ (f,\ F\ \mathcal{I}_C)$›
using *D.T.fixing-ide-gives-functor-2* **by** *simp*
interpret *R*: *endofunctor D* ‹$\lambda f.\ T_D\ (f,\ F\ \mathcal{I}_C)$› **..**
interpret $x\psi$: *natural-transformation D D* ‹$\lambda f.\ f \otimes_D (F\ \mathcal{I}_C)$› ‹$\lambda f.\ f \otimes_D \mathcal{I}_D$›
‹$\lambda f.\ f \otimes_D \psi$›
using ψ *D.T.fixing-arr-gives-natural-transformation-2* [*of* ψ] **by** *auto*
interpret $x\psi$: *natural-isomorphism D D* ‹$\lambda f.\ f \otimes_D (F\ \mathcal{I}_C)$› ‹$\lambda f.\ f \otimes_D \mathcal{I}_D$› ‹$\lambda f.\ f \otimes_D \psi$›
using ψ *D.tensor-preserves-iso* **by** (*unfold-locales*, *simp*)
interpret $\varrho_D o x\psi$: *vertical-composite D D* ‹$\lambda f.\ f \otimes_D (F\ \mathcal{I}_C)$› ‹$\lambda f.\ f \otimes_D \mathcal{I}_D$› *D.map*
‹$\lambda f.\ f \otimes_D \psi$› *D.*$\varrho$ **..**
interpret $\varrho_D o x\psi$: *natural-isomorphism D D* ‹$\lambda f.\ f \otimes_D (F\ \mathcal{I}_C)$› *D.map* $\varrho_D o x\psi$*.map*
using $x\psi$*.natural-isomorphism-axioms D.*ϱ*.natural-isomorphism-axioms*

natural-isomorphisms-compose **by** *blast*
interpret *R*: *equivalence-functor D D* ‹$\lambda f.\ f \otimes_D (F\ \mathcal{I}_C)$›
using *R.isomorphic-to-identity-is-equivalence* $\varrho_D ox \psi$*.natural-isomorphism-axioms*
by *simp*
show *equivalence-functor D D* ($\lambda f.\ f \otimes_D (D.cod\ \iota_1)$)
using *R.equivalence-functor-axioms C.unit-in-hom* **by** *auto*
qed
show *monoidal-category-with-alternate-unit D* $T_D\ \alpha_D\ \iota_D\ \iota_1$
using *D.pentagon C.unit-is-iso C.unit-in-hom preserves-hom* ι_1*-is-iso* ι_1*-in-hom*
by (*unfold-locales*, *auto*)
qed

no-notation *D.tensor* (**infixr** $\otimes_D$ *53*)
notation $D.C_1.tensor$ (**infixr** $\otimes_D$ *53*)
no-notation *D.assoc* (a_D[-, -, -])
notation $D.C_1.assoc$ (a_D[-, -, -])
no-notation *D.assoc′* ($\mathrm{a}_D{}^{-1}$[-, -, -])
notation $D.C_1.assoc'$ ($\mathrm{a}_D{}^{-1}$[-, -, -])
notation $D.C_1.unity$ ($\mathcal{I}_1$)
notation $D.C_1.lunit$ (l_1[-])
notation $D.C_1.runit$ (r_1[-])

lemma $\mathcal{I}_1$*-char* [*simp*]:
shows $\mathcal{I}_1 = F\ \mathcal{I}_C$
using ι_1*-in-hom* **by** *auto*

definition ψ
where $\psi \equiv THE\ \psi.\ «\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C» \wedge D.iso\ \psi \wedge \psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$

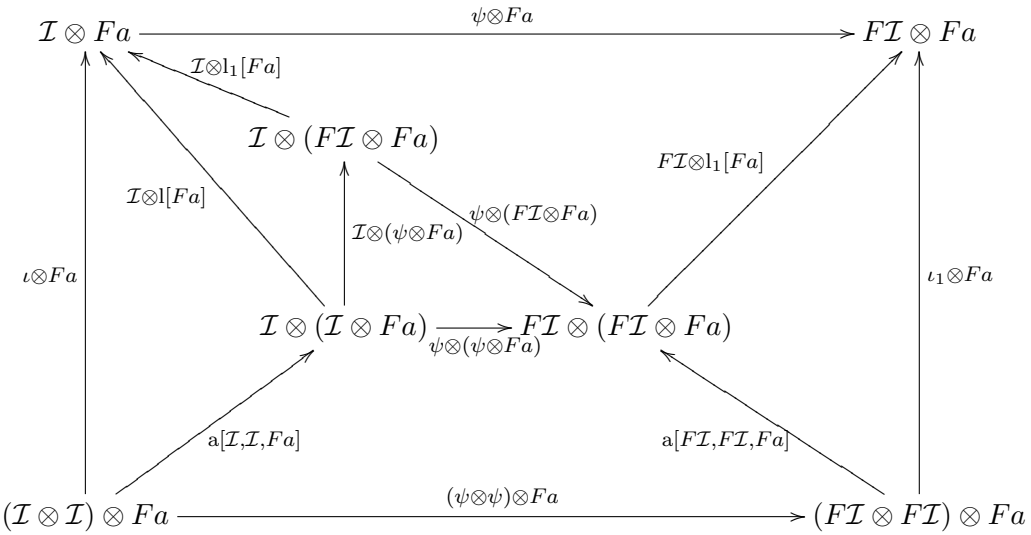
lemma ψ*-char*:
shows «$\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C$» **and** *D.iso* ψ **and** $\psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$
and $\exists!\psi.\ «\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C» \wedge D.iso\ \psi \wedge \psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$
proof −
show $\exists!\psi.\ «\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C» \wedge D.iso\ \psi \wedge \psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$
using *D.unit-unique-upto-unique-iso* ι_1*-in-hom*
by (*elim D.in-homE*, *auto*)
hence *1*: «$\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C$» $\wedge$ *D.iso* $\psi \wedge \psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$
unfolding ψ*-def*
using *theI′* [*of* $\lambda\psi.$ «$\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C$» $\wedge$ *D.iso* $\psi \wedge \psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$]
by *fast*
show «$\psi : \mathcal{I}_D \to_D F\ \mathcal{I}_C$» **using** *1* **by** *simp*
show *D.iso* ψ **using** *1* **by** *simp*
show $\psi \cdot_D \iota_D = \iota_1 \cdot_D (\psi \otimes_D \psi)$ **using** *1* **by** *simp*
qed

lemma ψ*-eqI*:
assumes «f: $\mathcal{I}_D \to_D F\ \mathcal{I}_C$» **and** *D.iso f* **and** $f \cdot_D \iota_D = \iota_1 \cdot_D (f \otimes_D f)$
shows $f = \psi$
using *assms* ψ*-def* ψ*-char*

the1-equality [*of* $\lambda f.$ «f: $\mathcal{I}_D \to_D F\ \mathcal{I}_C$» $\wedge$ *D.iso* $f \wedge f \cdot_D \iota_D = \iota_1 \cdot_D (f \otimes_D f)\ f$]
by *simp*

lemma *lunit-coherence1*:
assumes *C.ide a*
shows $\mathrm{l}_1[F\ a] \cdot_D (\psi \otimes_D F\ a) = \mathrm{l}_D[F\ a]$
proof −
have *D.par* $(\mathrm{l}_1[F\ a] \cdot_D (\psi \otimes_D F\ a))\ \mathrm{l}_D[F\ a]$
using *assms D.C*$_1$*.lunit-in-hom D.tensor-in-hom D.lunit-in-hom* ψ*-char*(*1*)
by *auto*

The upper left triangle in the following diagram commutes.

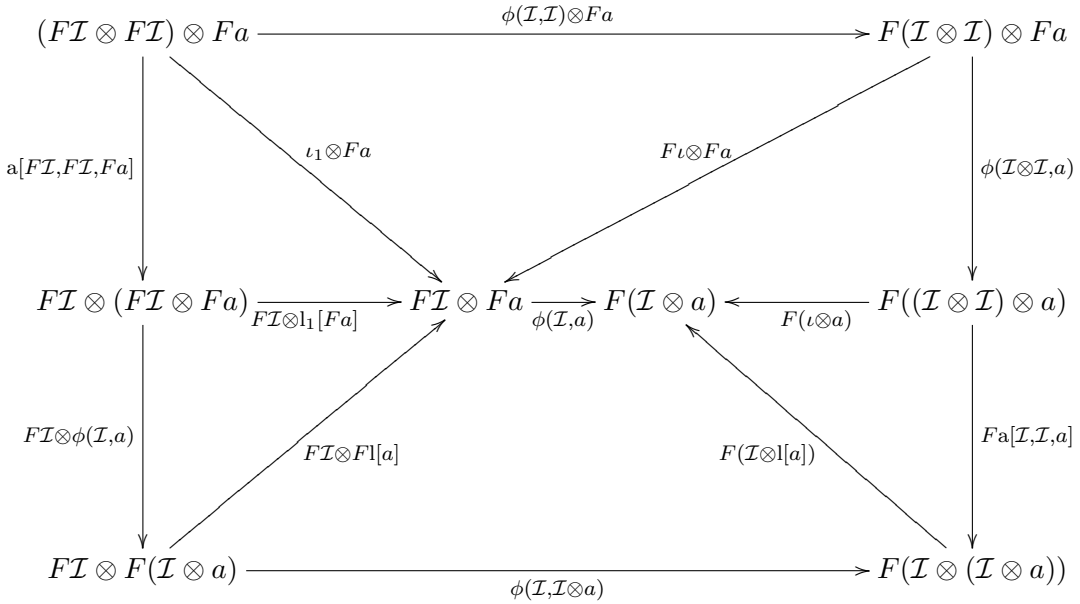


moreover have $(\mathcal{I}_D \otimes_D \mathrm{l}_1[F\ a]) \cdot_D (\mathcal{I}_D \otimes_D \psi \otimes_D F\ a) = \mathcal{I}_D \otimes_D \mathrm{l}_D[F\ a]$
proof −
have $(\mathcal{I}_D \otimes_D \mathrm{l}_1[F\ a]) \cdot_D (\mathcal{I}_D \otimes_D \psi \otimes_D F\ a)$
$= (\mathcal{I}_D \otimes_D \mathrm{l}_1[F\ a]) \cdot_D (D.inv\ \psi \otimes_D F\ \mathcal{I}_C \otimes_D F\ a) \cdot_D (\psi \otimes_D \psi \otimes_D F\ a)$
using *assms* ψ*-char*(*1*−*2*) *D.interchange* [*of D.inv* ψ] *D.comp-cod-arr*
D.inv-is-inverse D.comp-inv-arr
by (*elim D.in-homE*, *simp*)
also have ... $= (D.inv\ \psi \otimes_D F\ a) \cdot_D (F\ \mathcal{I}_C \otimes_D \mathrm{l}_1[F\ a]) \cdot_D (\psi \otimes_D \psi \otimes_D F\ a)$
proof −
have $(\mathcal{I}_D \otimes_D \mathrm{l}_1[F\ a]) \cdot_D (D.inv\ \psi \otimes_D F\ \mathcal{I}_C \otimes_D F\ a) =$
$(D.inv\ \psi \otimes_D F\ a) \cdot_D (F\ \mathcal{I}_C \otimes_D \mathrm{l}_1[F\ a])$
using *assms* ψ*-char*(*1*−*2*) *D.interchange* [*of* $\mathcal{I}_D$] *D.interchange* [*of D.inv* ψ]
D.comp-arr-dom D.comp-cod-arr
by (*elim D.in-homE*, *auto*)
thus *?thesis*
using *assms* ψ*-char*(*1*−*2*) *D.inv-in-hom*
D.comp-permute [*of* $\mathcal{I}_D \otimes_D \mathrm{l}_1[F\ a]$ *D.inv* $\psi \otimes_D F\ \mathcal{I}_C \otimes_D F\ a$
D.inv $\psi \otimes_D F\ a\ F\ \mathcal{I}_C \otimes_D \mathrm{l}_1[F\ a]$]

```
      by (elim D.in-homE, auto)
    qed
    also have ... = (D.inv ψ ⊗D F a) ·D (ι1 ⊗D F a) ·D D.inv aD[F ℐC, F ℐC, F a] ·D
                    (ψ ⊗D ψ ⊗D F a)
      using assms ψ-char(1−2) D.C1.lunit-char(2) D.comp-assoc by auto
    also have ... = ((D.inv ψ ⊗D F a) ·D (ι1 ⊗D F a) ·D ((ψ ⊗D ψ) ⊗D F a)) ·D
                    D.inv aD[ℐD, ℐD, F a]
      using assms ψ-char(1−2) D.assoc'-naturality [of ψ ψ F a] D.comp-assoc by auto
    also have ... = (ιD ⊗D F a) ·D D.inv aD[ℐD, ℐD, F a]
    proof −
      have (D.inv ψ ⊗D F a) ·D (ι1 ⊗D F a) ·D ((ψ ⊗D ψ) ⊗D F a) = ιD ⊗D F a
      proof −
        have (D.inv ψ ⊗D F a) ·D (ι1 ⊗D F a) ·D ((ψ ⊗D ψ) ⊗D F a) =
              D.inv ψ ·D ψ ·D ιD ⊗D F a
          using assms ψ-char(1−3) ι1-in-hom D.interchange
          by (elim D.in-homE, auto)
        also have ... = ιD ⊗D F a
          using assms ψ-char(1−2) D.inv-is-inverse D.comp-inv-arr D.comp-cod-arr
                D.comp-reduce D.unit-in-hom
          by (elim D.in-homE, auto)
        finally show ?thesis by blast
      qed
      thus ?thesis by simp
    qed
    also have ... = ℐD ⊗D l_D[F a]
      using assms D.lunit-char by simp
    finally show ?thesis by blast
  qed
  ultimately show ?thesis
    using D.L.is-faithful [of l1[F a] ·D (ψ ⊗D F a) l_D[F a]] by force
qed

lemma lunit-coherence2:
assumes C.ide a
shows F l_C[a] ·D φ (ℐC, a) = l1[F a]
proof −
```

We show that the lower left triangle in the following diagram commutes.



have $(F\ \mathcal{I}_C \otimes_D F\ \mathrm{l}_C[a]) \cdot_D (F\ \mathcal{I}_C \otimes_D \varphi\ (\mathcal{I}_C,\ a)) = F\ \mathcal{I}_C \otimes_D \mathrm{l}_1[F\ a]$
proof −
have $(F\ \mathcal{I}_C \otimes_D F\ \mathrm{l}_C[a]) \cdot_D (F\ \mathcal{I}_C \otimes_D \varphi\ (\mathcal{I}_C,\ a))$
$= (F\ \mathcal{I}_C \otimes_D F\ \mathrm{l}_C[a]) \cdot_D D.inv\ (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C \otimes_C a)) \cdot_D F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a] \cdot_D$
$\varphi\ (\mathcal{I}_C \otimes_C \mathcal{I}_C,\ a) \cdot_D (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C) \otimes_D F\ a) \cdot_D D.inv\ \mathrm{a}_D[F\ \mathcal{I}_C,\ F\ \mathcal{I}_C,\ F\ a]$
proof −
have $D.inv\ (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C \otimes_C a)) \cdot_D F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a] \cdot_D \varphi\ (\mathcal{I}_C \otimes_C \mathcal{I}_C,\ a) \cdot_D$
$(\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C) \otimes_D F\ a)$
$= (F\ \mathcal{I}_C \otimes_D \varphi\ (\mathcal{I}_C,\ a)) \cdot_D \mathrm{a}_D[F\ \mathcal{I}_C,\ F\ \mathcal{I}_C,\ F\ a]$
using *assms* φ*-in-hom assoc-coherence D.invert-side-of-triangle*(*1*) **by** *simp*
hence $F\ \mathcal{I}_C \otimes_D \varphi\ (\mathcal{I}_C,\ a)$
$= (D.inv\ (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C \otimes_C a)) \cdot_D F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a] \cdot_D \varphi\ (\mathcal{I}_C \otimes_C \mathcal{I}_C,\ a) \cdot_D$
$(\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C) \otimes_D F\ a)) \cdot_D D.inv\ \mathrm{a}_D[F\ \mathcal{I}_C,\ F\ \mathcal{I}_C,\ F\ a]$
using *assms* φ*-in-hom D.invert-side-of-triangle*(*2*) **by** *simp*
thus *?thesis*
using *D.comp-assoc* **by** *simp*
qed
also have ... $= (F\ \mathcal{I}_C \otimes_D F\ \mathrm{l}_C[a]) \cdot_D D.inv\ (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C \otimes_C a)) \cdot_D$
$(D.inv\ (F\ (\mathcal{I}_C \otimes_C \mathrm{l}_C[a])) \cdot_D F\ (\iota_C \otimes_C a)) \cdot_D$
$\varphi\ (\mathcal{I}_C \otimes_C \mathcal{I}_C,\ a) \cdot_D (\varphi\ (\mathcal{I}_C,\ \mathcal{I}_C) \otimes_D F\ a) \cdot_D$
$D.inv\ \mathrm{a}_D[F\ \mathcal{I}_C,\ F\ \mathcal{I}_C,\ F\ a]$
proof −
have *1*: $F\ (\mathcal{I}_C \otimes_C \mathrm{l}_C[a]) = F\ (\iota_C \otimes_C a) \cdot_D D.inv\ (F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a])$
using *assms C.lunit-char*(*1−2*) *C.unit-in-hom preserves-inv* **by** *auto*
hence $F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a] = D.inv\ (F\ (\mathcal{I}_C \otimes_C \mathrm{l}_C[a])) \cdot_D F\ (\iota_C \otimes_C a)$
proof −
have $F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C,\ a] \cdot_D D.inv\ (F\ (\iota_C \otimes_C a))$
$= D.inv\ (F\ (\iota_C \otimes_C a) \cdot_D D.inv\ (F\ \mathrm{a}_C[\mathcal{I}_C,\ \mathcal{I}_C\ ,a]))$

```
    using assms 1 preserves-iso C.ide-is-iso C.unit-is-iso C.ide-unity C.iso-assoc
          C.iso-lunit C.tensor-preserves-iso D.inv-comp D.inv-inv
          D.iso-inv-iso D.iso-is-arr
    by metis
  thus ?thesis
    using assms 1 preserves-iso C.ide-is-iso C.unit-is-iso C.ide-unity C.iso-assoc
          C.iso-lunit C.tensor-preserves-iso D.inv-comp D.inv-inv
          D.iso-inv-iso D.iso-is-arr D.invert-side-of-triangle(2)
    by metis
 qed
 thus ?thesis by argo
qed
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D]
                D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] (F (ι[C] ⊗[C] a) ⋅[D] φ (ℐ[C] ⊗[C] ℐ[C], a)) ⋅[D]
                (φ (ℐ[C], ℐ[C]) ⊗[D] F a) ⋅[D] D.inv a[D][F ℐ[C], F ℐ[C], F a]
  using D.comp-assoc by auto
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D]
                D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] (φ (ℐ[C], a) ⋅[D] (F ι[C] ⊗[D] F a)) ⋅[D]
                (φ (ℐ[C], ℐ[C]) ⊗[D] F a) ⋅[D] D.inv a[D][F ℐ[C], F ℐ[C], F a]
  using assms φ.naturality [of (ι[C], a)] C.unit-in-hom by auto
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D]
                D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] φ (ℐ[C], a) ⋅[D]
                ((F ι[C] ⊗[D] F a) ⋅[D] (φ (ℐ[C], ℐ[C]) ⊗[D] F a)) ⋅[D]
                D.inv a[D][F ℐ[C], F ℐ[C], F a]
  using D.comp-assoc by auto
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D]
                D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] φ (ℐ[C], a) ⋅[D] (ι[1] ⊗[D] F a) ⋅[D]
                D.inv a[D][F ℐ[C], F ℐ[C], F a]
  using assms D.interchange C.unit-in-hom by auto
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D]
                D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] φ (ℐ[C], a) ⋅[D]
                ((F ℐ[C] ⊗[D] l[1][F a]) ⋅[D] a[D][F ℐ[C], F ℐ[C], F a]) ⋅[D]
                D.inv a[D][F ℐ[C], F ℐ[C], F a]
proof −
  have (ι[1] ⊗[D] F a) ⋅[D] a[D]⁻¹[F ℐ[C], F ℐ[C], F a] = F ℐ[C] ⊗[D] l[1][F a]
    using assms D.C[1].lunit-char [of F a] by auto
  thus ?thesis
    using assms D.inv-is-inverse ι[1]-in-hom φ-in-hom D.invert-side-of-triangle(2)
    by simp
qed
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D]
                (D.inv (φ (ℐ[C], ℐ[C] ⊗[C] a)) ⋅[D] D.inv (F (ℐ[C] ⊗[C] l[C][a])) ⋅[D] φ (ℐ[C], a)) ⋅[D]
                (F ℐ[C] ⊗[D] l[1][F a])
  using assms D.comp-arr-dom [of F ℐ[C] ⊗[D] l[1][F a]] D.comp-assoc by auto
also have ... = (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] D.inv (F ℐ[C] ⊗[D] F l[C][a]) ⋅[D] (F ℐ[C] ⊗[D] l[1][F a])
proof −
  have D.inv (F ℐ[C] ⊗[D] F l[C][a])
         = D.inv (D.inv (φ (ℐ[C], a)) ⋅[D] F (ℐ[C] ⊗[C] l[C][a]) ⋅[D] φ (ℐ[C], ℐ[C] ⊗[C] a))
    using assms φ.naturality [of (ℐ[C], l[C][a])] D.invert-side-of-triangle(1) by simp
```

```
    also have ... = D.inv (φ (ℐ_C, ℐ_C ⊗_C a)) ⋅_D D.inv (F (ℐ_C ⊗_C l_C[a])) ⋅_D φ (ℐ_C, a)
      using assms D.inv-comp D.inv-is-inverse D.isos-compose D.comp-assoc
      by simp
    finally have D.inv (F ℐ_C ⊗_D F l_C[a])
                   = D.inv (φ (ℐ_C, ℐ_C ⊗_C a)) ⋅_D D.inv (F (ℐ_C ⊗_C l_C[a])) ⋅_D φ (ℐ_C, a)
      by blast
    thus ?thesis by argo
  qed
  also have ... = ((F ℐ_C ⊗_D F l_C[a]) ⋅_D D.inv (F ℐ_C ⊗_D F l_C[a])) ⋅_D (F ℐ_C ⊗_D l_1[F a])
    using assms D.tensor-preserves-iso D.comp-assoc by simp
  also have ... = F ℐ_C ⊗_D l_1[F a]
    using assms D.tensor-preserves-iso D.comp-arr-inv D.inv-is-inverse D.comp-cod-arr
          D.interchange
    by simp
  finally show ?thesis by blast
qed
hence F ℐ_C ⊗_D F l_C[a] ⋅_D φ (ℐ_C, a) = F ℐ_C ⊗_D l_1[F a]
  using assms φ-in-hom D.interchange by simp
moreover have D.par (F l_C[a] ⋅_D φ (ℐ_C, a)) l_1[F a]
  using assms φ-in-hom by simp
ultimately show ?thesis
  using D.C_1.L.is-faithful [of F l_C[a] ⋅_D φ (ℐ_C, a) l_1[F a]] by simp
qed
```

Combining the two previous lemmas yields the coherence result we seek. This is the condition that is traditionally taken as part of the definition of monoidal functor.

```
lemma lunit-coherence:
assumes C.ide a
shows l_D[F a] = F l_C[a] ⋅_D φ (ℐ_C, a) ⋅_D (ψ ⊗_D F a)
proof −
  have l_D[F a] ⋅_D D.inv (ψ ⊗_D F a) = l_1[F a]
    using assms lunit-coherence1 ψ-char(2)
          D.invert-side-of-triangle(2) [of l_D[F a] l_1[F a] ψ ⊗_D F a]
    by auto
  also have ... = F l_C[a] ⋅_D φ (ℐ_C, a)
    using assms lunit-coherence2 by simp
  finally have l_D[F a] ⋅_D D.inv (ψ ⊗_D F a) = F l_C[a] ⋅_D φ (ℐ_C, a)
    by blast
  hence l_D[F a] = (F l_C[a] ⋅_D φ (ℐ_C, a)) ⋅_D (ψ ⊗_D F a)
    using assms ψ-char(2) φ-in-hom
          D.invert-side-of-triangle(2) [of F l_C[a] ⋅_D φ (ℐ_C, a) l_D[F a] D.inv (ψ ⊗_D F a)]
    by simp
  thus ?thesis
    using assms ψ-char(1) D.comp-assoc by auto
qed
```

We now want to obtain the corresponding result for the right unitor. To avoid a repetition of what would amount to essentially the same tedious diagram chases that were carried out above, we instead show here that F becomes a monoidal functor from

the opposite of C to the opposite of D, with $\lambda f.\ \varphi\ (snd\ f,\ fst\ f)$ as the structure map. The fact that in the opposite monoidal categories the left and right unitors are exchanged then permits us to obtain the result for the right unitor from the result already proved for the left unitor.

interpretation C': *opposite-monoidal-category* $C\ T_C\ \alpha_C\ \iota_C$ **..**
interpretation D': *opposite-monoidal-category* $D\ T_D\ \alpha_D\ \iota_D$ **..**
interpretation $T_D'oFF$: *composite-functor* $C.CC.comp\ D.CC.comp\ D\ FF.map\ D'.T$ **..**
interpretation FoT_C': *composite-functor* $C.CC.comp\ C\ D\ C'.T\ F$ **..**
interpretation φ': *natural-transformation* $C.CC.comp\ D\ T_D'oFF.map\ FoT_C'.map$
$\langle\lambda f.\ \varphi\ (snd\ f,\ fst\ f)\rangle$
using *φ.is-natural-1 φ.is-natural-2 φ.is-extensional* **by** (*unfold-locales*, *auto*)
interpretation φ': *natural-isomorphism* $C.CC.comp\ D\ T_D'oFF.map\ FoT_C'.map$
$\langle\lambda f.\ \varphi\ (snd\ f,\ fst\ f)\rangle$
by (*unfold-locales*, *simp*)
interpretation F': *monoidal-functor* $C\ C'.T\ C'.\alpha\ \iota_C\ D\ D'.T\ D'.\alpha\ \iota_D\ F\ \langle\lambda f.\ \varphi\ (snd\ f,\ fst\ f)\rangle$
using *preserves-unity* **apply** (*unfold-locales*; *simp*)
proof −
fix *a b c*
assume *a*: $C.ide\ a$ **and** *b*: $C.ide\ b$ **and** *c*: $C.ide\ c$
have $(\varphi\ (c \otimes_C b,\ a) \cdot_D (\varphi\ (c,\ b) \otimes_D F\ a)) \cdot_D \mathrm{a}_D^{-1}[F\ c,\ F\ b,\ F\ a] =$
$F\ (C.assoc'\ c\ b\ a) \cdot_D \varphi\ (c,\ b \otimes_C a) \cdot_D (F\ c \otimes_D \varphi\ (b,\ a))$
proof −
have $D.seq\ (F\ \mathrm{a}_C[c,\ b,\ a])\ (\varphi\ (c \otimes_C b,\ a) \cdot_D (\varphi\ (c,\ b) \otimes_D F\ a))$
using *a b c φ-in-hom* **by** *simp*
moreover have $D.seq\ (\varphi\ (c,\ b \otimes_C a) \cdot_D (F\ c \otimes_D \varphi\ (b,\ a)))\ \mathrm{a}_D[F\ c,\ F\ b,\ F\ a]$
using *a b c φ-in-hom* **by** *simp*
moreover have
$F\ \mathrm{a}_C[c,\ b,\ a] \cdot_D \varphi\ (c \otimes_C b,\ a) \cdot_D (\varphi\ (c,\ b) \otimes_D F\ a) =$
$(\varphi\ (c,\ b \otimes_C a) \cdot_D (F\ c \otimes_D \varphi\ (b,\ a))) \cdot_D \mathrm{a}_D[F\ c,\ F\ b,\ F\ a]$
using *a b c assoc-coherence D.comp-assoc* **by** *simp*
moreover have $D.iso\ (F\ \mathrm{a}_C[c,b,a])$
using *a b c* **by** *simp*
moreover have $D.iso\ \mathrm{a}_D[F\ c,\ F\ b,\ F\ a]$
using *a b c* **by** *simp*
moreover have $D.inv\ (F\ \mathrm{a}_C[c,b,a]) = F\ (C.assoc'\ c\ b\ a)$
using *a b c preserves-inv* **by** *simp*
ultimately show *?thesis*
using *D.invert-opposite-sides-of-square* **by** *simp*
qed
thus $F\ (C.assoc'\ c\ b\ a) \cdot_D \varphi\ (c,\ b \otimes_C a) \cdot_D (F\ c \otimes_D \varphi\ (b,\ a)) =$
$\varphi\ (c \otimes_C b,\ a) \cdot_D (\varphi\ (c,\ b) \otimes_D F\ a) \cdot_D \mathrm{a}_D^{-1}[F\ c,\ F\ b,\ F\ a]$
using *D.comp-assoc* **by** *simp*
qed

lemma *induces-monoidal-functor-between-opposites*:
shows $monoidal\text{-}functor\ C\ C'.T\ C'.\alpha\ \iota_C\ D\ D'.T\ D'.\alpha\ \iota_D\ F\ (\lambda f.\ \varphi\ (snd\ f,\ fst\ f))$
..

```
  lemma runit-coherence:
  assumes C.ide a
  shows r_D[F a] = F r_C[a] ·_D φ (a, I_C) ·_D (F a ⊗_D ψ)
  proof −
    have C'.lunit a = r_C[a]
      using assms C'.lunit-simp by simp
    moreover have D'.lunit (F a) = r_D[F a]
      using assms D'.lunit-simp by simp
    moreover have F'.ψ = ψ
    proof (intro ψ-eqI)
      show «F'.ψ : D'.unity →_D F C'.unity» using F'.ψ-char(1) by simp
      show D.iso F'.ψ using F'.ψ-char(2) by simp
      show F'.ψ ·_D ι_D = ι_1 ·_D (F'.ψ ⊗_D F'.ψ) using F'.ψ-char(3) by simp
    qed
    moreover have D'.lunit (F a) = F (C'.lunit a) ·_D φ (a, C'.unity) ·_D (F a ⊗_D F'.ψ)
      using assms F'.lunit-coherence by simp
    ultimately show ?thesis by simp
  qed

end
```

3.1 Strict Monoidal Functor

A strict monoidal functor preserves the monoidal structure "on the nose".

```
  locale strict-monoidal-functor =
    C: monoidal-category C T_C α_C ι_C +
    D: monoidal-category D T_D α_D ι_D +
    functor C D F
  for C :: 'c comp                    (infixr ·_C 55)
  and T_C :: 'c * 'c ⇒ 'c
  and α_C :: 'c * 'c * 'c ⇒ 'c
  and ι_C :: 'c
  and D :: 'd comp                    (infixr ·_D 55)
  and T_D :: 'd * 'd ⇒ 'd
  and α_D :: 'd * 'd * 'd ⇒ 'd
  and ι_D :: 'd
  and F :: 'c ⇒ 'd +
  assumes strictly-preserves-ι: F ι_C = ι_D
  and strictly-preserves-T: ⟦ C.arr f; C.arr g ⟧ ⟹ F (T_C (f, g)) = T_D (F f, F g)
  and strictly-preserves-α-ide: ⟦ C.ide a; C.ide b; C.ide c ⟧ ⟹
                                F (α_C (a, b, c)) = α_D (F a, F b, F c)
  begin

    notation C.tensor                 (infixr ⊗_C 53)
    and C.unity                       (I_C)
    and C.lunit                       (l_C[-])
    and C.runit                       (r_C[-])
    and C.assoc                       (a_C[-, -, -])
```

and *D.tensor* (**infixr** $\otimes_D$ *53*)
and *D.unity* ($\mathcal{I}_D$)
and *D.lunit* (l_D[-])
and *D.runit* (r_D[-])
and *D.assoc* (a_D[-, -, -])

lemma *strictly-preserves-tensor*:
assumes *C.arr f* **and** *C.arr g*
shows *F* (*f* $\otimes_C$ *g*) = *F f* $\otimes_D$ *F g*
using *assms strictly-preserves-T* **by** *blast*

lemma *strictly-preserves-α*:
assumes *C.arr f* **and** *C.arr g* **and** *C.arr h*
shows *F* (α_C (*f*, *g*, *h*)) = α_D (*F f*, *F g*, *F h*)
proof −
have *F* (α_C (*f*, *g*, *h*)) = *F* ((*f* $\otimes_C$ *g* $\otimes_C$ *h*) $\cdot_C$ α_C (*C.dom f*, *C.dom g*, *C.dom h*))
using *assms C.α.is-natural-1* [*of* (*f*, *g*, *h*)] *C.T.ToCT-simp* **by** *force*
also have ... = (*F f* $\otimes_D$ *F g* $\otimes_D$ *F h*) $\cdot_D$ α_D (*D.dom* (*F f*), *D.dom* (*F g*), *D.dom* (*F h*))
using *assms strictly-preserves-α-ide strictly-preserves-tensor* **by** *simp*
also have ... = α_D (*F f*, *F g*, *F h*)
using *assms D.α.is-natural-1* [*of* (*F f*, *F g*, *F h*)] **by** *simp*
finally show *?thesis* **by** *blast*
qed

lemma *strictly-preserves-unity*:
shows *F* $\mathcal{I}_C$ = $\mathcal{I}_D$
using *C.unit-in-hom strictly-preserves-ι* **by** *auto*

lemma *strictly-preserves-assoc*:
assumes *C.arr a* **and** *C.arr b* **and** *C.arr c*
shows *F* a_C[*a*, *b*, *c*] = a_D[*F a*, *F b*, *F c*]
using *assms strictly-preserves-α* **by** *simp*

lemma *strictly-preserves-lunit*:
assumes *C.ide a*
shows *F* l_C[*a*] = l_D[*F a*]
proof −
let *?P* = λ*f*. *f* ∈ *C.hom* ($\mathcal{I}_C$ $\otimes_C$ *a*) *a* ∧ $\mathcal{I}_C$ $\otimes_C$ *f* = (ι_C $\otimes_C$ *a*) $\cdot_C$ *C.assoc′* $\mathcal{I}_C$ $\mathcal{I}_C$ *a*
let *?Q* = λ*f*. *f* ∈ *D.hom* ($\mathcal{I}_D$ $\otimes_D$ *F a*) (*F a*) ∧
$\mathcal{I}_D$ $\otimes_D$ *f* = (ι_D $\otimes_D$ *F a*) $\cdot_D$ *D.assoc′* $\mathcal{I}_D$ $\mathcal{I}_D$ (*F a*)
have *1*: *?P* l_C[*a*] **using** *assms C.lunit-char* **by** *simp*
hence *?Q* (*F* l_C[*a*])
proof −
have *F* l_C[*a*] ∈ *D.hom* ($\mathcal{I}_D$ $\otimes_D$ *F a*) (*F a*)
using *assms 1 strictly-preserves-unity strictly-preserves-tensor* **by** *auto*
moreover have
F ((ι_C $\otimes_C$ *a*) $\cdot_C$ *C.assoc′* $\mathcal{I}_C$ $\mathcal{I}_C$ *a*) = (ι_D $\otimes_D$ *F a*) $\cdot_D$ *D.assoc′* $\mathcal{I}_D$ $\mathcal{I}_D$ (*F a*)
using *assms 1 strictly-preserves-ι strictly-preserves-assoc strictly-preserves-unity*
strictly-preserves-tensor preserves-inv C.unit-in-hom

```
    by auto
  moreover have 𝓘_D ⊗_D F l_C[a] = F (𝓘_C ⊗_C l_C[a])
    using assms strictly-preserves-unity strictly-preserves-tensor by simp
  ultimately show ?thesis
    using assms C.lunit-char(2) by simp
 qed
 thus ?thesis using assms D.lunit-eqI by simp
qed

lemma strictly-preserves-runit:
assumes C.ide a
shows F r_C[a] = r_D[F a]
proof −
 let ?P = λf. f ∈ C.hom (a ⊗_C 𝓘_C) a ∧ f ⊗_C 𝓘_C = (a ⊗_C ι_C) ·_C C.assoc a 𝓘_C 𝓘_C
 let ?Q = λf. f ∈ D.hom (F a ⊗_D 𝓘_D) (F a) ∧
              f ⊗_D 𝓘_D = (F a ⊗_D ι_D) ·_D D.assoc (F a) 𝓘_D 𝓘_D
 have 1: ?P r_C[a] using assms C.runit-char by simp
 hence ?Q (F r_C[a])
 proof −
  have F r_C[a] ∈ D.hom (F a ⊗_D 𝓘_D) (F a)
    using assms 1 strictly-preserves-unity strictly-preserves-tensor by auto
  moreover have F ((a ⊗_C ι_C) ·_C C.assoc a 𝓘_C 𝓘_C)
                  = (F a ⊗_D ι_D) ·_D D.assoc (F a) 𝓘_D 𝓘_D
    using assms 1 strictly-preserves-ι strictly-preserves-assoc strictly-preserves-unity
          strictly-preserves-tensor preserves-inv C.unit-in-hom
    by auto
  moreover have F r_C[a] ⊗_D 𝓘_D = F (r_C[a] ⊗_C 𝓘_C)
    using assms strictly-preserves-unity strictly-preserves-tensor by simp
  ultimately show ?thesis
    using assms C.runit-char(2) by simp
 qed
 thus ?thesis using assms D.runit-eqI by simp
qed
```

The following are used to simplify the expression of the sublocale relationship between *strict-monoidal-functor* and *monoidal-functor*, as the definition of the latter mentions the structure map φ. For a strict monoidal functor, this is an identity transformation.

```
interpretation FF: product-functor C C D D F F ..
interpretation FoT_C: composite-functor C.CC.comp C D T_C F ..
interpretation T_DoFF: composite-functor C.CC.comp D.CC.comp D FF.map T_D ..

lemma structure-is-trivial:
shows T_DoFF.map = FoT_C.map
proof
 fix x
 have C.CC.arr x ⟹ T_DoFF.map x = FoT_C.map x
 proof −
  assume x: C.CC.arr x
  have T_DoFF.map x = F (fst x) ⊗_D F (snd x)
```

```
      using x by simp
    also have ... = FoT_C.map x
      using x strictly-preserves-tensor [of fst x snd x] by simp
    finally show T_DoFF.map x = FoT_C.map x by simp
  qed
  moreover have ¬ C.CC.arr x ⟹ T_DoFF.map x = FoT_C.map x
    using T_DoFF.is-extensional FoT_C.is-extensional by simp
  ultimately show T_DoFF.map x = FoT_C.map x by blast
 qed

 abbreviation φ where φ ≡ T_DoFF.map

 lemma structure-is-natural-isomorphism:
 shows natural-isomorphism C.CC.comp D T_DoFF.map FoT_C.map φ
   using T_DoFF.as-nat-iso.natural-isomorphism-axioms structure-is-trivial by force

end
```

A strict monoidal functor is a monoidal functor.

```
sublocale strict-monoidal-functor ⊆ monoidal-functor C T_C α_C ι_C D T_D α_D ι_D F φ
proof −
 interpret FF: product-functor C C D D F F ..
 interpret FoT_C: composite-functor C.CC.comp C D T_C F ..
 interpret T_DoFF: composite-functor C.CC.comp D.CC.comp D FF.map T_D ..
 interpret φ: natural-isomorphism C.CC.comp D T_DoFF.map FoT_C.map φ
   using structure-is-natural-isomorphism by simp
 show monoidal-functor C T_C α_C ι_C D T_D α_D ι_D F φ
 proof
   show D.isomorphic I_D (F I_C)
   proof (unfold D.isomorphic-def)
     have «I_D : I_D →_D F I_C» ∧ D.iso I_D
       using strictly-preserves-unity by auto
     thus ∃f. «f : I_D →_D F I_C» ∧ D.iso f by blast
   qed
   fix a b c
   assume a: C.ide a
   assume b: C.ide b
   assume c: C.ide c
   show F a_C[a, b, c] ·_D φ (a ⊗_C b, c) ·_D (φ (a, b) ⊗_D F c) =
        φ (a, b ⊗_C c) ·_D (F a ⊗_D φ (b, c)) ·_D a_D[F a, F b, F c]
     using a b c strictly-preserves-tensor strictly-preserves-assoc
           D.comp-arr-dom D.comp-cod-arr
     by simp
 qed
qed

lemma strict-monoidal-functors-compose:
assumes strict-monoidal-functor B T_B α_B ι_B C T_C α_C ι_C F
and strict-monoidal-functor C T_C α_C ι_C D T_D α_D ι_D G
```

shows *strict-monoidal-functor B* T_B α_B ι_B *D* T_D α_D ι_D (*G o F*)
proof −
interpret *F*: *strict-monoidal-functor B* T_B α_B ι_B *C* T_C α_C ι_C *F*
using *assms*(*1*) **by** *auto*
interpret *G*: *strict-monoidal-functor C* T_C α_C ι_C *D* T_D α_D ι_D *G*
using *assms*(*2*) **by** *auto*
interpret *GoF*: *composite-functor B C D F G* **..**
show *?thesis*
using *F.strictly-preserves-T F.strictly-preserves-ι F.strictly-preserves-α*
G.strictly-preserves-T G.strictly-preserves-ι G.strictly-preserves-α
by (*unfold-locales*, *simp-all*)
qed

An equivalence of monoidal categories is a monoidal functor whose underlying ordinary functor is also part of an ordinary equivalence of categories.

locale *equivalence-of-monoidal-categories* =
C: *monoidal-category C* T_C α_C ι_C +
D: *monoidal-category D* T_D α_D ι_D +
equivalence-of-categories C D F G η ε +
monoidal-functor D T_D α_D ι_D *C* T_C α_C ι_C *F* φ
for *C* :: *'c comp* (**infixr** $\cdot_C$ *55*)
and T_C :: $'c * 'c \Rightarrow 'c$
and α_C :: $'c * 'c * 'c \Rightarrow 'c$
and ι_C :: *'c*
and *D* :: *'d comp* (**infixr** $\cdot_D$ *55*)
and T_D :: $'d * 'd \Rightarrow 'd$
and α_D :: $'d * 'd * 'd \Rightarrow 'd$
and ι_D :: *'d*
and *F* :: $'d \Rightarrow 'c$
and φ :: $'d * 'd \Rightarrow 'c$
and ι :: *'c*
and *G* :: $'c \Rightarrow 'd$
and η :: $'d \Rightarrow 'd$
and ε :: $'c \Rightarrow 'c$

end

Chapter 4

The Free Monoidal Category

theory *FreeMonoidalCategory*
imports *Category3.Subcategory MonoidalFunctor*
begin

In this theory, we use the monoidal language of a category C defined in *MonoidalCategory.MonoidalCategory* to give a construction of the free monoidal category $\mathcal{F}C$ generated by C. The arrows of $\mathcal{F}C$ are the equivalence classes of formal arrows obtained by declaring two formal arrows to be equivalent if they are parallel and have the same diagonalization. Composition, tensor, and the components of the associator and unitors are all defined in terms of the corresponding syntactic constructs. After defining $\mathcal{F}C$ and showing that it does indeed have the structure of a monoidal category, we prove the freeness: every functor from C to a monoidal category D extends uniquely to a strict monoidal functor from $\mathcal{F}C$ to D.

We then consider the full subcategory $\mathcal{F}_S C$ of $\mathcal{F}C$ whose objects are the equivalence classes of diagonal identity terms (*i.e.* equivalence classes of lists of identity arrows of C), and we show that this category is monoidally equivalent to $\mathcal{F}C$. In addition, we show that $\mathcal{F}_S C$ is the free strict monoidal category, as any functor from C to a strict monoidal category D extends uniquely to a strict monoidal functor from $\mathcal{F}_S C$ to D.

4.1 Syntactic Construction

locale *free-monoidal-category* =
  monoidal-language C
  for $C :: 'c$ *comp*
begin

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

Two terms of the monoidal language of C are defined to be equivalent if they are parallel formal arrows with the same diagonalization.

  abbreviation *equiv*

where *equiv t u* ≡ *Par t u* ∧ ⌊*t*⌋ = ⌊*u*⌋

Arrows of $\mathcal{F}C$ will be the equivalence classes of formal arrows determined by the relation *equiv*. We define here the property of being an equivalence class of the relation *equiv*. Later we show that this property coincides with that of being an arrow of the category that we will construct.

type-synonym *'a arr* = *'a term set*
definition *ARR* **where** *ARR f* ≡ *f* ≠ {} ∧ (∀ *t*. *t* ∈ *f* ⟶ *f* = *Collect* (*equiv t*))

lemma *not-ARR-empty*:
shows ¬*ARR* {}
using *ARR-def* **by** *simp*

lemma *ARR-eqI*:
assumes *ARR f* **and** *ARR g* **and** *f* ∩ *g* ≠ {}
shows *f* = *g*
using *assms ARR-def* **by** *fastforce*

We will need to choose a representative of each equivalence class as a normal form. The requirements we have of these representatives are: (1) the normal form of an arrow t is equivalent to t; (2) equivalent arrows have identical normal forms; (3) a normal form is a canonical term if and only if its diagonalization is an identity. It follows from these properties and coherence that a term and its normal form have the same evaluation in any monoidal category. We choose here as a normal form for an arrow t the particular term *Inv* (*Cod t*↓) ⋅ ⌊*t*⌋ ⋅ *Dom t*↓. However, the only specific properties of this definition we actually use are the three we have just stated.

definition *norm* (‖-‖)
where ‖*t*‖ = *Inv* (*Cod t*↓) ⋅ ⌊*t*⌋ ⋅ *Dom t*↓

If t is a formal arrow, then t is equivalent to its normal form.

lemma *equiv-norm-Arr*:
assumes *Arr t*
shows *equiv* ‖*t*‖ *t*
proof −
have *Par t* (*Inv* (*Cod t*↓) ⋅ ⌊*t*⌋ ⋅ *Dom t*↓)
using *assms Diagonalize-in-Hom red-in-Hom Inv-in-Hom Arr-implies-Ide-Dom*
Arr-implies-Ide-Cod Ide-implies-Arr Can-red
by *auto*
moreover have ⌊(*Inv* (*Cod t*↓) ⋅ ⌊*t*⌋ ⋅ *Dom t*↓)⌋ = ⌊*t*⌋
using *assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diagonalize-preserves-Ide*
Diagonalize-in-Hom Diagonalize-Inv [*of Cod t*↓] *Diag-Diagonalize*
CompDiag-Diag-Dom [*of* ⌊*t*⌋] *CompDiag-Cod-Diag* [*of* ⌊*t*⌋]
by (*simp add*: *Diagonalize-red* [*of Cod t*] *Can-red*(*1*))
ultimately show *?thesis* **using** *norm-def* **by** *simp*
qed

Equivalent arrows have identical normal forms.

lemma *norm-respects-equiv*:

assumes *equiv t u*
shows $\|t\| = \|u\|$
using *assms norm-def* **by** *simp*

The normal form of an arrow is canonical if and only if its diagonalization is an identity term.

lemma *Can-norm-iff-Ide-Diagonalize*:
assumes *Arr t*
shows *Can* $\|t\| \longleftrightarrow$ *Ide* $\lfloor t \rfloor$
using *assms norm-def Can-implies-Arr Arr-implies-Ide-Dom Arr-implies-Ide-Cod Can-red Inv-preserves-Can Diagonalize-preserves-Can red-in-Hom Diagonalize-in-Hom Ide-Diagonalize-Can*
by *fastforce*

We now establish various additional properties of normal forms that are consequences of the three already proved. The definition *norm-def* is not used subsequently.

lemma *norm-preserves-Can*:
assumes *Can t*
shows *Can* $\|t\|$
using *assms Can-implies-Arr Can-norm-iff-Ide-Diagonalize Ide-Diagonalize-Can* **by** *simp*

lemma *Par-Arr-norm*:
assumes *Arr t*
shows *Par* $\|t\|$ *t*
using *assms equiv-norm-Arr* **by** *auto*

lemma *Diagonalize-norm* [*simp*]:
assumes *Arr t*
shows $\lfloor \|t\| \rfloor = \lfloor t \rfloor$
using *assms equiv-norm-Arr* **by** *auto*

lemma *unique-norm*:
assumes *ARR f*
shows $\exists !t.\ \forall u.\ u \in f \longrightarrow \|u\| = t$
proof
have *1*: $(SOME\ t.\ t \in f) \in f$
using *assms ARR-def someI-ex* [*of* $\lambda t.\ t \in f$] **by** *auto*
show $\bigwedge t.\ \forall u.\ u \in f \longrightarrow \|u\| = t \Longrightarrow t = \|SOME\ t.\ t \in f\|$
using *assms ARR-def 1* **by** *auto*
show $\forall u.\ u \in f \longrightarrow \|u\| = \|SOME\ t.\ t \in f\|$
using *assms ARR-def 1 norm-respects-equiv* **by** *blast*
qed

lemma *Dom-norm*:
assumes *Arr t*
shows *Dom* $\|t\|$ = *Dom t*
using *assms Par-Arr-norm* **by** *metis*

lemma *Cod-norm*:

assumes *Arr t*
shows *Cod* $\|t\|$ = *Cod t*
using *assms Par-Arr-norm* **by** *metis*

lemma *norm-in-Hom*:
assumes *Arr t*
shows $\|t\| \in$ *Hom* (*Dom t*) (*Cod t*)
using *assms Par-Arr-norm* [*of t*] **by** *simp*

As all the elements of an equivalence class have the same normal form, we can use the normal form of an arbitrarily chosen element as a canonical representative.

definition *rep* **where** *rep f* ≡ $\|$*SOME t*. *t* ∈ *f*$\|$

lemma *rep-in-ARR*:
assumes *ARR f*
shows *rep f* ∈ *f*
using *assms ARR-def someI-ex* [*of* λ*t*. *t* ∈ *f*] *equiv-norm-Arr rep-def ARR-def*
by *fastforce*

lemma *Arr-rep-ARR*:
assumes *ARR f*
shows *Arr* (*rep f*)
using *assms ARR-def rep-in-ARR* **by** *auto*

We next define a function *mkarr* that maps formal arrows to their equivalence classes. For terms that are not formal arrows, the function yields the empty set.

definition *mkarr* **where** *mkarr t* = *Collect* (*equiv t*)

lemma *mkarr-extensionality*:
assumes ¬*Arr t*
shows *mkarr t* = {}
using *assms mkarr-def* **by** *simp*

lemma *ARR-mkarr*:
assumes *Arr t*
shows *ARR* (*mkarr t*)
using *assms ARR-def mkarr-def* **by** *auto*

lemma *mkarr-memb-ARR*:
assumes *ARR f* **and** *t* ∈ *f*
shows *mkarr t* = *f*
using *assms ARR-def mkarr-def* **by** *simp*

lemma *mkarr-rep-ARR* [*simp*]:
assumes *ARR f*
shows *mkarr* (*rep f*) = *f*
using *assms rep-in-ARR mkarr-memb-ARR* **by** *auto*

lemma *Arr-in-mkarr*:

assumes *Arr t*
shows $t \in$ *mkarr t*
using *assms mkarr-def* **by** *simp*

Two terms are related by *equiv* iff they are both formal arrows and have identical normal forms.

lemma *equiv-iff-eq-norm*:
shows *equiv t u* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ $\|t\| = \|u\|$
proof
show *equiv t u* $\Longrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ $\|t\| = \|u\|$
using *mkarr-def Arr-in-mkarr ARR-mkarr unique-norm* **by** *blast*
show *Arr t* $\wedge$ *Arr u* $\wedge$ $\|t\| = \|u\| \Longrightarrow$ *equiv t u*
using *Par-Arr-norm Diagonalize-norm* **by** *metis*
qed

lemma *norm-norm* [*simp*]:
assumes *Arr t*
shows $\|\|t\|\| = \|t\|$
proof −
have $t \in$ *mkarr t*
using *assms Arr-in-mkarr* **by** *blast*
moreover have $\|t\| \in$ *mkarr t*
using *assms equiv-norm-Arr mkarr-def* **by** *simp*
ultimately show *?thesis* **using** *assms ARR-mkarr unique-norm* **by** *auto*
qed

lemma *norm-in-ARR*:
assumes *ARR f* **and** $t \in f$
shows $\|t\| \in f$
using *assms ARR-def equiv-iff-eq-norm norm-norm Par-Arr-norm* **by** *fastforce*

lemma *norm-rep-ARR* [*simp*]:
assumes *ARR f*
shows $\|rep\ f\| = rep\ f$
using *assms ARR-def someI-ex* [*of* $\lambda t.\ t \in f$] *rep-def norm-norm* **by** *fastforce*

lemma *norm-memb-eq-rep-ARR*:
assumes *ARR f* **and** $t \in f$
shows *norm t* = *rep f*
using *assms ARR-def someI-ex* [*of* $\lambda t.\ t \in f$] *unique-norm rep-def* **by** *metis*

lemma *rep-mkarr*:
assumes *Arr f*
shows *rep* (*mkarr f*) = $\|f\|$
using *assms ARR-mkarr Arr-in-mkarr norm-memb-eq-rep-ARR* **by** *fastforce*

To prove that two terms determine the same equivalence class, it suffices to show that they are parallel formal arrows with identical diagonalizations.

lemma *mkarr-eqI* [*intro*]:

assumes *Par f g* **and** $\lfloor f \rfloor = \lfloor g \rfloor$
shows *mkarr f* = *mkarr g*
using *assms* **by** (*metis ARR-mkarr equiv-iff-eq-norm rep-mkarr mkarr-rep-ARR*)

We use canonical representatives to lift the formal domain and codomain functions from terms to equivalence classes.

abbreviation *DOM* **where** *DOM f* ≡ *Dom* (*rep f*)
abbreviation *COD* **where** *COD f* ≡ *Cod* (*rep f*)

lemma *DOM-mkarr*:
assumes *Arr t*
shows *DOM* (*mkarr t*) = *Dom t*
using *assms rep-mkarr* **by** (*metis Par-Arr-norm*)

lemma *COD-mkarr*:
assumes *Arr t*
shows *COD* (*mkarr t*) = *Cod t*
using *assms rep-mkarr* **by** (*metis Par-Arr-norm*)

A composition operation can now be defined on equivalence classes using the syntactic constructor *Comp*.

definition *comp* (**infixr** · *55*)
where *comp f g* ≡ (*if ARR f* ∧ *ARR g* ∧ *DOM f* = *COD g*
then mkarr ((*rep f*) · (*rep g*)) *else* {})

We commence the task of showing that the composition *comp* so defined determines a category.

interpretation *partial-composition comp*
apply *unfold-locales*
using *comp-def not-ARR-empty* **by** *metis*

notation *in-hom* («- : - → -»)

The empty set serves as the null for the composition.

lemma *null-char*:
shows *null* = {}
proof −
let *?P* = λ*n*. ∀ *f*. *f* · *n* = *n* ∧ *n* · *f* = *n*
have *?P* {} **using** *comp-def not-ARR-empty* **by** *simp*
moreover have ∃!*n*. *?P n* **using** *ex-un-null* **by** *metis*
ultimately show *?thesis* **using** *null-def theI-unique* [*of ?P* {}]
by (*metis null-is-zero*(*2*))
qed

lemma *ARR-comp*:
assumes *ARR f* **and** *ARR g* **and** *DOM f* = *COD g*
shows *ARR* (*f* · *g*)
using *assms comp-def Arr-rep-ARR ARR-mkarr*(*1*) **by** *simp*

lemma *DOM-comp* [*simp*]:
assumes *ARR f* **and** *ARR g* **and** *DOM f* = *COD g*
shows *DOM* (*f* · *g*) = *DOM g*
using *assms comp-def ARR-comp Arr-rep-ARR DOM-mkarr* **by** *simp*

lemma *COD-comp* [*simp*]:
assumes *ARR f* **and** *ARR g* **and** *DOM f* = *COD g*
shows *COD* (*f* · *g*) = *COD f*
using *assms comp-def ARR-comp Arr-rep-ARR COD-mkarr* **by** *simp*

lemma *comp-assoc*:
assumes *g* · *f* ≠ *null* **and** *h* · *g* ≠ *null*
shows *h* · (*g* · *f*) = (*h* · *g*) · *f*
proof −
have *1*: *ARR f* ∧ *ARR g* ∧ *ARR h* ∧ *DOM h* = *COD g* ∧ *DOM g* = *COD f*
using *assms comp-def not-ARR-empty null-char* **by** *metis*
hence *2*: *Arr* (*rep f*) ∧ *Arr* (*rep g*) ∧ *Arr* (*rep h*) ∧
Dom (*rep h*) = *Cod* (*rep g*) ∧ *Dom* (*rep g*) = *Cod* (*rep f*)
using *Arr-rep-ARR* **by** *simp*
have *3*: *h* · *g* · *f* = *mkarr* (*rep h* · *rep* (*mkarr* (*rep g* · *rep f*)))
using *1 comp-def ARR-comp COD-comp* **by** *simp*
also have ... = *mkarr* (*rep h* · *rep g* · *rep f*)
proof −
have *equiv* (*rep h* · *rep* (*mkarr* (*rep g* · *rep f*))) (*rep h* · *rep g* · *rep f*)
proof −
have *Par* (*rep h* · *rep* (*mkarr* (*rep g* · *rep f*))) (*rep h* · *rep g* · *rep f*)
using *1 2 3 DOM-mkarr ARR-comp COD-comp mkarr-extensionality not-ARR-empty*
by (*metis Arr.simps*(*4*) *Cod.simps*(*4*) *Dom.simps*(*4*) *snd-map-prod*)

moreover have ⌊*rep h* · *rep* (*mkarr* (*rep g* · *rep f*))⌋ = ⌊*rep h* · *rep g* · *rep f*⌋
using *1 2 Arr-rep-ARR rep-mkarr rep-in-ARR assms*(*1*) *ARR-comp mkarr-extensionality*
comp-def equiv-iff-eq-norm norm-memb-eq-rep-ARR null-char
by *auto*
ultimately show *?thesis* **using** *equiv-iff-eq-norm* **by** *blast*
qed
thus *?thesis*
using *mkarr-def* **by** *force*
qed
also have ... = *mkarr* ((*rep h* · *rep g*) · *rep f*)
proof −
have *Par* (*rep h* · *rep g* · *rep f*) ((*rep h* · *rep g*) · *rep f*)
using *2* **by** *simp*
moreover have ⌊*rep h* · *rep g* · *rep f*⌋ = ⌊(*rep h* · *rep g*) · *rep f*⌋
using *2 Diag-Diagonalize* **by** (*simp add*: *CompDiag-assoc*)
ultimately show *?thesis*
using *equiv-iff-eq-norm* **by** (*simp add*: *mkarr-def*)
qed
also have ... = *mkarr* (*rep* (*mkarr* (*rep h* · *rep g*)) · *rep f*)
proof −

```
    have equiv (rep (mkarr (rep h · rep g)) · rep f) ((rep h · rep g) · rep f)
    proof −
      have Par (rep (mkarr (rep h · rep g)) · rep f) ((rep h · rep g) · rep f)
        using 1 2 Arr-rep-ARR DOM-comp ARR-comp COD-comp comp-def by auto
      moreover have ⌊rep (mkarr (rep h · rep g)) · rep f⌋ = ⌊(rep h · rep g) · rep f⌋
     using assms(2) 1 2 ARR-comp Arr-rep-ARR mkarr-extensionality rep-mkarr rep-in-ARR
           equiv-iff-eq-norm norm-memb-eq-rep-ARR comp-def null-char
        by simp
      ultimately show ?thesis using equiv-iff-eq-norm by blast
    qed
    thus ?thesis
      using mkarr-def by auto
  qed
  also have ... = (h · g) · f
    using 1 comp-def ARR-comp DOM-comp by simp
  finally show ?thesis by blast
qed

lemma Comp-in-comp-ARR:
assumes ARR f and ARR g and DOM f = COD g
and t ∈ f and u ∈ g
shows t · u ∈ f · g
proof −
  have equiv (t · u) (rep f · rep g)
  proof −
    have 1: Par (t · u) (rep f · rep g)
      using assms ARR-def Arr-rep-ARR COD-mkarr DOM-mkarr mkarr-memb-ARR
            mkarr-extensionality
      by (metis (no-types, lifting) Arr.simps(4) Cod.simps(4) Dom.simps(4) snd-map-prod)

    moreover have ⌊t · u⌋ = ⌊rep f · rep g⌋
      using assms 1 rep-in-ARR equiv-iff-eq-norm norm-memb-eq-rep-ARR
      by (metis (no-types, lifting) Arr.simps(4) Diagonalize.simps(4))
    ultimately show ?thesis by simp
  qed
  thus ?thesis
    using assms comp-def mkarr-def by simp
qed
```

Ultimately, we will show that that the identities of the category are those equivalence classes, all of whose members diagonalize to formal identity arrows, having the further property that their canonical representative is a formal endo-arrow.

```
definition IDE where IDE f ≡ ARR f ∧ (∀ t. t ∈ f ⟶ Ide ⌊t⌋) ∧ DOM f = COD f

lemma IDE-implies-ARR:
assumes IDE f
shows ARR f
  using assms IDE-def ARR-def by auto
```

lemma *IDE-mkarr-Ide*:
assumes *Ide a*
shows *IDE* (*mkarr a*)
proof −
  have *DOM* (*mkarr a*) = *COD* (*mkarr a*)
  using *assms mkarr-def equiv-iff-eq-norm Par-Arr-norm COD-mkarr DOM-mkarr Ide-in-Hom*
    by (*metis Ide-implies-Can Inv-Ide Ide-implies-Arr Inv-preserves-Can*(*2*))
  moreover have *ARR* (*mkarr a*) ∧ (∀ *t*. *t* ∈ *mkarr a* ⟶ *Ide* ⌊*t*⌋)
  proof −
    have *ARR* (*mkarr a*) **using** *assms ARR-mkarr Ide-implies-Arr* **by** *simp*
    moreover have ∀ *t*. *t* ∈ *mkarr a* ⟶ *Ide* ⌊*t*⌋
      using *assms mkarr-def Diagonalize-preserves-Ide* **by** *fastforce*
    ultimately show *?thesis* **by** *blast*
  qed
  ultimately show *?thesis* **using** *IDE-def* **by** *blast*
qed

lemma *IDE-implies-ide*:
assumes *IDE a*
shows *ide a*
proof (*unfold ide-def*)
  have *a* · *a* ≠ *null*
  proof −
    have *rep a* • *rep a* ∈ *a* · *a*
      using *assms IDE-def comp-def Arr-rep-ARR Arr-in-mkarr* **by** *simp*
    thus *?thesis*
      using *null-char* **by** *auto*
  qed
  moreover have ⋀*f*. (*f* · *a* ≠ *null* ⟶ *f* · *a* = *f*) ∧ (*a* · *f* ≠ *null* ⟶ *a* · *f* = *f*)
  proof
    fix *f* :: *'c arr*
    show *a* · *f* ≠ *null* ⟶ *a* · *f* = *f*
    proof
      assume *f*: *a* · *f* ≠ *null*
      hence *ARR f*
        using *comp-def null-char* **by** *auto*
      have *rep a* • *rep f* ∈ *a* · *f*
        using *assms f Comp-in-comp-ARR comp-def rep-in-ARR null-char* **by** *metis*
      moreover have *rep a* • *rep f* ∈ *f*
      proof −
        have *rep f* ∈ *f*
          using ‹*ARR f*› *rep-in-ARR* **by** *auto*
        moreover have *equiv* (*rep a* • *rep f*) (*rep f*)
        proof −
          have *1*: *Par* (*rep a* • *rep f*) (*rep f*)
            using *assms f comp-def mkarr-extensionality Arr-rep-ARR IDE-def null-char*
            by (*metis Cod.simps*(*4*) *Dom.simps*(*4*))
          moreover have ⌊*rep a* • *rep f*⌋ = ⌊*rep f*⌋
            using *assms f 1 comp-def IDE-def CompDiag-Ide-Diag Diag-Diagonalize*(*1*)

```
            Diag-Diagonalize(2) Diag-Diagonalize(3) rep-in-ARR
        by auto
       ultimately show ?thesis by auto
     qed
     ultimately show ?thesis
      using ‹ARR f› ARR-def by auto
    qed
    ultimately show a · f = f
     using mkarr-memb-ARR comp-def by auto
   qed
   show f · a ≠ null ⟶ f · a = f
   proof
    assume f: f · a ≠ null
    hence ARR f
     using comp-def null-char by auto
    have rep f • rep a ∈ f · a
     using assms f Comp-in-comp-ARR comp-def rep-in-ARR null-char by metis
    moreover have rep f • rep a ∈ f
    proof −
     have rep f ∈ f
      using ‹ARR f› rep-in-ARR by auto
     moreover have equiv (rep f • rep a) (rep f)
     proof −
      have 1: Par (rep f • rep a) (rep f)
       using assms f comp-def mkarr-extensionality Arr-rep-ARR IDE-def null-char
       by (metis Cod.simps(4) Dom.simps(4))
      moreover have ⌊rep f • rep a⌋ = ⌊rep f⌋
       using assms f 1 comp-def IDE-def CompDiag-Diag-Ide
            Diag-Diagonalize(1) Diag-Diagonalize(2) Diag-Diagonalize(3)
            rep-in-ARR
       by force
      ultimately show ?thesis by auto
     qed
     ultimately show ?thesis
      using ‹ARR f› ARR-def by auto
    qed
    ultimately show f · a = f
     using mkarr-memb-ARR comp-def by auto
   qed
  qed
  ultimately show a · a ≠ null ∧
                  (∀f. (f · a ≠ null ⟶ f · a = f) ∧ (a · f ≠ null ⟶ a · f = f))
   by blast
qed

lemma ARR-iff-has-domain:
shows ARR f ⟷ domains f ≠ {}
proof
  assume f: domains f ≠ {}
```

```
  show ARR f using f domains-def comp-def null-char by auto
  next
  assume f: ARR f
  have Ide (DOM f)
    using f ARR-def by (simp add: Arr-implies-Ide-Dom Arr-rep-ARR)
  hence IDE (mkarr (DOM f)) using IDE-mkarr-Ide by metis
  hence ide (mkarr (DOM f)) using IDE-implies-ide by simp
  moreover have f · mkarr (DOM f) = f
  proof −
    have 1: rep f ⋆ DOM f ∈ f · mkarr (DOM f)
      using f Comp-in-comp-ARR
      using IDE-implies-ARR Ide-in-Hom rep-in-ARR ‹IDE (mkarr (DOM f))›
          ‹Ide (DOM f)› Arr-in-mkarr COD-mkarr
      by fastforce
    moreover have rep f ⋆ DOM f ∈ f
    proof −
      have 2: rep f ∈ f using f rep-in-ARR by simp
      moreover have equiv (rep f ⋆ DOM f) (rep f)
        by (metis 1 Arr.simps(4) Arr-rep-ARR COD-mkarr Cod.simps(4)
            Diagonalize-Comp-Arr-Dom Dom.simps(4) IDE-def Ide-implies-Arr
            ‹IDE (mkarr (DOM f))› ‹Ide (DOM f)› all-not-in-conv DOM-mkarr comp-def)
      ultimately show ?thesis
        using f ARR-eqI 1 ‹ide (mkarr (DOM f))› null-char ide-def by auto
    qed
    ultimately show ?thesis
      using f ARR-eqI ‹ide (mkarr (DOM f))› null-char ide-def by auto
  qed
  ultimately show domains f ≠ {}
    using f domains-def not-ARR-empty null-char by auto
qed

lemma ARR-iff-has-codomain:
shows ARR f ⟷ codomains f ≠ {}
proof
  assume f: codomains f ≠ {}
  show ARR f using f codomains-def comp-def null-char by auto
  next
  assume f: ARR f
  have Ide (COD f)
    using f ARR-def by (simp add: Arr-rep-ARR Arr-implies-Ide-Cod)
  hence IDE (mkarr (COD f)) using IDE-mkarr-Ide by metis
  hence ide (mkarr (COD f)) using IDE-implies-ide by simp
  moreover have mkarr (COD f) · f = f
  proof −
    have 1: COD f ⋆ rep f ∈ mkarr (COD f) · f
      using f Comp-in-comp-ARR
      using IDE-implies-ARR Ide-in-Hom rep-in-ARR ‹IDE (mkarr (COD f))›
          ‹Ide (COD f)› Arr-in-mkarr DOM-mkarr
      by fastforce
```

```
    moreover have COD f ⋆ rep f ∈ f
      using 1 null-char norm-rep-ARR norm-memb-eq-rep-ARR mkarr-memb-ARR
            ‹ide (mkarr (COD f))› emptyE equiv-iff-eq-norm mkarr-extensionality ide-def
      by metis
    ultimately show ?thesis
      using f ARR-eqI ‹ide (mkarr (COD f))› null-char ide-def by auto
  qed
  ultimately show codomains f ≠ {}
    using codomains-def f not-ARR-empty null-char by auto
qed

lemma arr-iff-ARR:
shows arr f ⟷ ARR f
  using arr-def ARR-iff-has-domain ARR-iff-has-codomain by simp
```

The arrows of the category are the equivalence classes of formal arrows.

```
lemma arr-char:
shows arr f ⟷ f ≠ {} ∧ (∀ t. t ∈ f ⟶ f = mkarr t)
  using arr-iff-ARR ARR-def mkarr-def by simp

lemma seq-char:
shows seq g f ⟷ g ⋅ f ≠ null
proof
  show g ⋅ f ≠ null ⟹ seq g f
    using comp-def null-char Comp-in-comp-ARR rep-in-ARR ARR-mkarr
          Arr-rep-ARR arr-iff-ARR
    by auto
  show seq g f ⟹ g ⋅ f ≠ null
    by auto
qed

lemma seq-char':
shows seq g f ⟷ ARR f ∧ ARR g ∧ DOM g = COD f
proof
  show ARR f ∧ ARR g ∧ DOM g = COD f ⟹ seq g f
    using comp-def null-char Comp-in-comp-ARR rep-in-ARR ARR-mkarr
          Arr-rep-ARR arr-iff-ARR
    by auto
  have ¬ (ARR f ∧ ARR g ∧ DOM g = COD f) ⟹ g ⋅ f = null
    using comp-def null-char by auto
  thus seq g f ⟹ ARR f ∧ ARR g ∧ DOM g = COD f
    using ext by fastforce
qed
```

Finally, we can show that the composition *comp* determines a category.

```
interpretation category comp
proof
  show ⋀f. domains f ≠ {} ⟷ codomains f ≠ {}
    using ARR-iff-has-domain ARR-iff-has-codomain by simp
```

show *1*: $\bigwedge f\ g.\ g \cdot f \neq null \Longrightarrow seq\ g\ f$
using *comp-def ARR-comp null-char arr-iff-ARR* **by** *metis*
fix *f g h*
show *seq h g* $\Longrightarrow$ *seq* $(h \cdot g)$ *f* $\Longrightarrow$ *seq g f*
using *seq-char'* **by** *auto*
show *seq h* $(g \cdot f) \Longrightarrow$ *seq g f* $\Longrightarrow$ *seq h g*
using *seq-char'* **by** *auto*
show *seq g f* $\Longrightarrow$ *seq h g* $\Longrightarrow$ *seq* $(h \cdot g)$ *f*
using *seq-char' ARR-comp arr-iff-ARR* **by** *auto*
show *seq g f* $\Longrightarrow$ *seq h g* $\Longrightarrow$ $(h \cdot g) \cdot f = h \cdot g \cdot f$
using *seq-char comp-assoc* **by** *auto*
qed

lemma *mkarr-rep* [*simp*]:
assumes *arr f*
shows *mkarr* (*rep f*) = *f*
using *assms arr-iff-ARR* **by** *simp*

lemma *arr-mkarr* [*simp*]:
assumes *Arr t*
shows *arr* (*mkarr t*)
using *assms* **by** (*simp add*: *ARR-mkarr arr-iff-ARR*)

lemma *mkarr-memb*:
assumes $t \in f$ **and** *arr f*
shows *Arr t* **and** *mkarr t* = *f*
using *assms arr-char mkarr-extensionality* **by** *auto*

lemma *rep-in-arr* [*simp*]:
assumes *arr f*
shows *rep f* $\in$ *f*
using *assms* **by** (*simp add*: *rep-in-ARR arr-iff-ARR*)

lemma *Arr-rep* [*simp*]:
assumes *arr f*
shows *Arr* (*rep f*)
using *assms mkarr-memb rep-in-arr* **by** *blast*

lemma *rep-in-Hom*:
assumes *arr f*
shows *rep f* $\in$ *Hom* (*DOM f*) (*COD f*)
using *assms* **by** *simp*

lemma *norm-memb-eq-rep*:
assumes *arr f* **and** $t \in f$
shows $\|t\|$ = *rep f*
using *assms arr-iff-ARR norm-memb-eq-rep-ARR* **by** *auto*

lemma *norm-rep*:

assumes *arr f*
shows $\|rep\ f\| = rep\ f$
using *assms norm-memb-eq-rep* **by** *simp*

Composition, domain, and codomain on arrows reduce to the corresponding syntactic operations on their representative terms.

lemma *comp-mkarr* [*simp*]:
assumes *Arr t* **and** *Arr u* **and** *Dom t* = *Cod u*
shows *mkarr t* · *mkarr u* = *mkarr* (*t* · *u*)
using *assms*
by (*metis* (*no-types*, *lifting*) *ARR-mkarr ARR-comp ARR-def Arr-in-mkarr COD-mkarr Comp-in-comp-ARR DOM-mkarr mkarr-def*)

lemma *dom-char*:
shows *dom f* = (*if arr f then mkarr* (*DOM f*) *else null*)
proof −
have ¬*arr f* ⟹ *?thesis*
using *dom-def* **by** (*simp add*: *arr-def*)
moreover have *arr f* ⟹ *?thesis*
proof −
assume *f*: *arr f*
have *dom f* = *mkarr* (*DOM f*)
proof (*intro dom-eqI*)
have *1*: *Ide* (*DOM f*)
using *f arr-char* **by** (*metis Arr-rep Arr-implies-Ide-Dom*)
hence *2*: *IDE* (*mkarr* (*DOM f*))
using *IDE-mkarr-Ide* **by** *metis*
thus *ide* (*mkarr* (*DOM f*)) **using** *IDE-implies-ide* **by** *simp*
moreover show *seq f* (*mkarr* (*DOM f*))
proof −
have *f* · *mkarr* (*DOM f*) ≠ *null*
using *f 1 2 ARR-def DOM-mkarr IDE-implies-ARR Ide-in-Hom ARR-comp IDE-def ARR-iff-has-codomain ARR-iff-has-domain null-char arr-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
thus *?thesis* **using** *seq-char* **by** *simp*
qed
qed
thus *?thesis* **using** *f* **by** *simp*
qed
ultimately show *?thesis* **by** *blast*
qed

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

lemma *cod-char*:
shows *cod f* = (*if arr f then mkarr* (*COD f*) *else null*)

```
proof −
  have ¬arr f ⟹ ?thesis
    using cod-def by (simp add: arr-def)
  moreover have arr f ⟹ ?thesis
  proof −
    assume f: arr f
    have cod f = mkarr (COD f)
    proof (intro cod-eqI)
      have 1: Ide (COD f)
        using f arr-char by (metis Arr-rep Arr-implies-Ide-Cod)
      hence 2: IDE (mkarr (COD f))
        using IDE-mkarr-Ide by metis
      thus ide (mkarr (COD f)) using IDE-implies-ide by simp
      moreover show seq (mkarr (COD f)) f
      proof −
        have mkarr (COD f) · f ≠ null
         using f 1 2 ARR-def DOM-mkarr IDE-implies-ARR Ide-in-Hom ARR-comp IDE-def
               ARR-iff-has-codomain ARR-iff-has-domain null-char arr-def
          by (metis (mono-tags, lifting) mem-Collect-eq)
        thus ?thesis using seq-char by simp
      qed
    qed
    thus ?thesis using f by simp
  qed
  ultimately show ?thesis by blast
qed

lemma cod-simp:
assumes arr f
shows cod f = mkarr (COD f)
  using assms cod-char by simp

lemma Dom-memb:
assumes arr f and t ∈ f
shows Dom t = DOM f
  using assms DOM-mkarr mkarr-extensionality arr-char by fastforce

lemma Cod-memb:
assumes arr f and t ∈ f
shows Cod t = COD f
  using assms COD-mkarr mkarr-extensionality arr-char by fastforce

lemma dom-mkarr [simp]:
assumes Arr t
shows dom (mkarr t) = mkarr (Dom t)
  using assms dom-char DOM-mkarr arr-mkarr by auto

lemma cod-mkarr [simp]:
assumes Arr t
```

shows *cod* (*mkarr t*) = *mkarr* (*Cod t*)
using *assms cod-char COD-mkarr arr-mkarr* **by** *auto*

lemma *mkarr-in-hom*:
assumes *Arr t*
shows «*mkarr t* : *mkarr* (*Dom t*) → *mkarr* (*Cod t*)»
using *assms arr-mkarr dom-mkarr cod-mkarr* **by** *auto*

lemma *DOM-in-dom* [*intro*]:
assumes *arr f*
shows *DOM f* ∈ *dom f*
using *assms dom-char*
by (*metis Arr-in-mkarr mkarr-extensionality ideD*(*1*) *ide-dom not-arr-null null-char*)

lemma *COD-in-cod* [*intro*]:
assumes *arr f*
shows *COD f* ∈ *cod f*
using *assms cod-char*
by (*metis Arr-in-mkarr mkarr-extensionality ideD*(*1*) *ide-cod not-arr-null null-char*)

lemma *DOM-dom*:
assumes *arr f*
shows *DOM* (*dom f*) = *DOM f*
using *assms Arr-rep Arr-implies-Ide-Dom Ide-implies-Arr dom-char rep-mkarr Par-Arr-norm*
Ide-in-Hom
by *simp*

lemma *DOM-cod*:
assumes *arr f*
shows *DOM* (*cod f*) = *COD f*
using *assms Arr-rep Arr-implies-Ide-Cod Ide-implies-Arr cod-char rep-mkarr Par-Arr-norm*
Ide-in-Hom
by *simp*

lemma *memb-equiv*:
assumes *arr f* **and** *t* ∈ *f* **and** *u* ∈ *f*
shows *Par t u* **and** ⌊*t*⌋ = ⌊*u*⌋
proof −
show *Par t u*
using *assms Cod-memb Dom-memb mkarr-memb*(*1*) **by** *metis*
show ⌊*t*⌋ = ⌊*u*⌋
using *assms arr-iff-ARR ARR-def* **by** *auto*
qed

Two arrows can be proved equal by showing that they are parallel and have representatives with identical diagonalizations.

lemma *arr-eqI*:
assumes *par f g* **and** *t* ∈ *f* **and** *u* ∈ *g* **and** ⌊*t*⌋ = ⌊*u*⌋
shows *f* = *g*

proof −
have *Arr t* ∧ *Arr u* **using** *assms mkarr-memb*(*1*) **by** *blast*
moreover have *Dom t* = *Dom u* ∧ *Cod t* = *Cod u*
using *assms Dom-memb Cod-memb comp-def arr-char comp-arr-dom comp-cod-arr*
by (*metis* (*full-types*))
ultimately have *Par t u* **by** *simp*
thus *?thesis*
using *assms arr-char* **by** (*metis rep-mkarr rep-in-arr equiv-iff-eq-norm*)
qed

lemma *comp-char*:
shows *f* · *g* = (*if seq f g then mkarr* (*rep f* ⋆ *rep g*) *else null*)
using *comp-def seq-char arr-char* **by** *meson*

The mapping that takes identity terms to their equivalence classes is injective.

lemma *mkarr-inj-on-Ide*:
assumes *Ide t* **and** *Ide u* **and** *mkarr t* = *mkarr u*
shows *t* = *u*
using *assms*
by (*metis* (*mono-tags*, *lifting*) *COD-mkarr Ide-in-Hom mem-Collect-eq*)

lemma *Comp-in-comp* [*intro*]:
assumes *arr f* **and** *g* ∈ *hom* (*dom g*) (*dom f*) **and** *t* ∈ *f* **and** *u* ∈ *g*
shows *t* ⋆ *u* ∈ *f* · *g*
proof −
have *ARR f* **using** *assms arr-iff-ARR* **by** *simp*
moreover have *ARR g* **using** *assms arr-iff-ARR* **by** *auto*
moreover have *DOM f* = *COD g*
using *assms dom-char cod-char mkarr-inj-on-Ide Arr-implies-Ide-Cod Arr-implies-Ide-Dom*
by *force*
ultimately show *?thesis* **using** *assms Comp-in-comp-ARR* **by** *simp*
qed

An arrow is defined to be "canonical" if some (equivalently, all) its representatives diagonalize to an identity term.

definition *can*
where *can f* ≡ *arr f* ∧ (∃ *t*. *t* ∈ *f* ∧ *Ide* ⌊*t*⌋)

lemma *can-def-alt*:
shows *can f* ⟷ *arr f* ∧ (∀ *t*. *t* ∈ *f* ⟶ *Ide* ⌊*t*⌋)
proof
assume *arr f* ∧ (∀ *t*. *t* ∈ *f* ⟶ *Ide* ⌊*t*⌋)
thus *can f* **using** *can-def arr-char* **by** *fastforce*
next
assume *f*: *can f*
show *arr f* ∧ (∀ *t*. *t* ∈ *f* ⟶ *Ide* ⌊*t*⌋)
proof −
obtain *t* **where** *t*: *t* ∈ *f* ∧ *Ide* ⌊*t*⌋ **using** *f can-def* **by** *auto*
have *ARR f* **using** *f can-def arr-char ARR-def mkarr-def* **by** *simp*

```
    hence ∀ u. u ∈ f ⟶ ‖u‖ = ‖t‖ using t unique-norm by auto
    hence ∀ u. u ∈ f ⟶ ⌊t⌋ = ⌊u⌋
      using t by (metis ‹ARR f› equiv-iff-eq-norm arr-iff-ARR mkarr-memb(1))
    hence ∀ u. u ∈ f ⟶ Ide ⌊u⌋
      using t by metis
    thus ?thesis using f can-def by blast
  qed
qed

lemma can-implies-arr:
assumes can f
shows arr f
  using assms can-def by auto
```

The identities of the category are precisely the canonical endo-arrows.

```
lemma ide-char:
shows ide f ⟷ can f ∧ dom f = cod f
proof
  assume f: ide f
  show can f ∧ dom f = cod f
    using f can-def arr-char dom-char cod-char IDE-def Arr-implies-Ide-Cod can-def-alt
          Arr-rep IDE-mkarr-Ide
    by (metis ideD(1) ideD(3))
  next
  assume f: can f ∧ dom f = cod f
  show ide f
  proof −
    have f = dom f
    proof (intro arr-eqI)
      show par f (dom f) using f can-def by simp
      show rep f ∈ f using f can-def by simp
      show DOM f ∈ dom f using f can-def by auto
      show ⌊rep f⌋ = ⌊DOM f⌋
      proof −
        have ⌊rep f⌋ ∈ Hom ⌊DOM f⌋ ⌊COD f⌋
          using f can-def Diagonalize-in-Hom by simp
        moreover have Ide ⌊rep f⌋ using f can-def-alt rep-in-arr by simp
        ultimately show ?thesis
          using f can-def Ide-in-Hom by simp
      qed
    qed
    thus ?thesis using f can-implies-arr ide-dom [of f] by auto
  qed
qed

lemma ide-iff-IDE:
shows ide a ⟷ IDE a
  using ide-char IDE-def can-def-alt arr-iff-ARR dom-char cod-char mkarr-inj-on-Ide
        Arr-implies-Ide-Cod Arr-implies-Ide-Dom Arr-rep
```

by *auto*

lemma *ide-mkarr-Ide*:
assumes *Ide a*
shows *ide* (*mkarr a*)
using *assms IDE-mkarr-Ide ide-iff-IDE* **by** *simp*

lemma *rep-dom*:
assumes *arr f*
shows *rep* (*dom f*) = ‖*DOM f*‖
using *assms dom-simp rep-mkarr Arr-rep Arr-implies-Ide-Dom* **by** *simp*

lemma *rep-cod*:
assumes *arr f*
shows *rep* (*cod f*) = ‖*COD f*‖
using *assms cod-simp rep-mkarr Arr-rep Arr-implies-Ide-Cod* **by** *simp*

lemma *rep-preserves-seq*:
assumes *seq g f*
shows *Seq* (*rep g*) (*rep f*)
using *assms Arr-rep dom-char cod-char mkarr-inj-on-Ide Arr-implies-Ide-Dom*
Arr-implies-Ide-Cod
by *auto*

lemma *rep-comp*:
assumes *seq g f*
shows *rep* (*g* · *f*) = ‖*rep g* ⋆ *rep f*‖
proof −
have *rep* (*g* · *f*) = *rep* (*mkarr* (*rep g* ⋆ *rep f*))
using *assms comp-char* **by** *metis*
also have ... = ‖*rep g* ⋆ *rep f*‖
using *assms rep-preserves-seq rep-mkarr* **by** *simp*
finally show *?thesis* **by** *blast*
qed

The equivalence classes of canonical terms are canonical arrows.

lemma *can-mkarr-Can*:
assumes *Can t*
shows *can* (*mkarr t*)
using *assms Arr-in-mkarr Can-implies-Arr Ide-Diagonalize-Can arr-mkarr can-def* **by** *blast*

lemma *ide-implies-can*:
assumes *ide a*
shows *can a*
using *assms ide-char* **by** *blast*

lemma *Can-rep-can*:
assumes *can f*
shows *Can* (*rep f*)

proof −
  have *Can* ‖*rep f*‖
    using *assms can-def-alt Can-norm-iff-Ide-Diagonalize* **by** *auto*
  moreover have *rep f* = ‖*rep f*‖
    using *assms can-implies-arr norm-rep* **by** *simp*
  ultimately show *?thesis* **by** *simp*
qed

Parallel canonical arrows are identical.

lemma *can-coherence*:
assumes *par f g* **and** *can f* **and** *can g*
shows *f* = *g*
proof −
  have ⌊*rep f*⌋ = ⌊*rep g*⌋
  proof −
    have ⌊*rep f*⌋ = ⌊*DOM f*⌋
     using *assms Ide-Diagonalize-Can Can-rep-can Diagonalize-in-Hom Ide-in-Hom* **by** *force*
    also have ... = ⌊*DOM g*⌋
      using *assms dom-char equiv-iff-eq-norm*
      by (*metis DOM-in-dom mkarr-memb*(*1*) *rep-mkarr arr-dom-iff-arr*)
    also have ... = ⌊*rep g*⌋
     using *assms Ide-Diagonalize-Can Can-rep-can Diagonalize-in-Hom Ide-in-Hom* **by** *force*
    finally show *?thesis* **by** *blast*
  qed
  hence *rep f* = *rep g*
    using *assms rep-in-arr norm-memb-eq-rep equiv-iff-eq-norm*
    by (*metis* (*no-types*, *lifting*) *arr-eqI*)
  thus *?thesis*
    using *assms arr-eqI* [*of f g*] *rep-in-arr* [*of f*] *rep-in-arr* [*of g*] **by** *metis*
qed

Canonical arrows are invertible, and their inverses can be obtained syntactically.

lemma *inverse-arrows-can*:
assumes *can f*
shows *inverse-arrows f* (*mkarr* (*Inv* (*DOM f*↓) ⋅ ⌊*rep f*⌋ ⋅ *COD f*↓))
proof
  let *?t* = (*Inv* (*DOM f*↓) ⋅ ⌊*rep f*⌋ ⋅ *COD f*↓)
  have *1*: *rep f* ∈ *f* ∧ *Arr* (*rep f*) ∧ *Can* (*rep f*) ∧ *Ide* ⌊*rep f*⌋
    using *assms can-def-alt rep-in-arr rep-in-arr*(*1*) *Can-rep-can* **by** *simp*
  hence *2*: ⌊*DOM f*⌋ = ⌊*COD f*⌋
    using *Diagonalize-in-Hom* [*of rep f*] *Ide-in-Hom* **by** *auto*
  have *3*: *Can ?t*
    using *assms 1 2 Can-red Ide-implies-Can Diagonalize-in-Hom Inv-preserves-Can*
        Arr-implies-Ide-Cod Arr-implies-Ide-Dom Diag-Diagonalize
    by *simp*
  have *4*: *DOM f* = *Cod ?t*
    using *assms can-def Can-red*
    by (*simp add*: *Arr-implies-Ide-Dom Inv-preserves-Can*(*3*))
  have *5*: *COD f* = *Dom ?t*

```
    using assms can-def Can-red Arr-rep Arr-implies-Ide-Cod by simp
  have 6: antipar f (mkarr ?t)
    using assms 3 4 5 dom-char cod-char can-def cod-mkarr dom-mkarr Can-implies-Arr
    by simp
  show ide (f · mkarr ?t)
  proof −
    have 7: par (f · mkarr ?t) (dom (f · mkarr ?t))
      using assms 6 by auto
    moreover have can (f · mkarr ?t)
    proof −
      have 8: Comp (rep f) ?t ∈ (f · mkarr ?t)
        using assms 1 3 4 6 can-implies-arr Arr-in-mkarr COD-mkarr Comp-in-comp-ARR
            Can-implies-Arr arr-iff-ARR seq-char'
        by meson
      moreover have Can (rep f · ?t)
        using 1 3 7 8 mkarr-memb(1) by (metis Arr.simps(4) Can.simps(4))
      ultimately show ?thesis
        using can-mkarr-Can 7 mkarr-memb(2) by metis
    qed
    moreover have can (dom (f · mkarr ?t))
      using 7 ide-implies-can by force
    ultimately have f · mkarr ?t = dom (f · mkarr ?t)
      using can-coherence by meson
    thus ?thesis
      using 7 ide-dom by metis
  qed
  show ide (mkarr ?t · f)
  proof −
    have 7: par (mkarr ?t · f) (cod (mkarr ?t · f))
      using assms 6 by auto
    moreover have can (mkarr ?t · f)
    proof −
      have 8: Comp ?t (rep f) ∈ mkarr ?t · f
        using assms 1 3 6 7 Arr-in-mkarr Comp-in-comp-ARR Can-implies-Arr arr-char
            comp-def
        by meson
      moreover have Can (?t · rep f)
        using 1 3 7 8 mkarr-memb(1) by (metis Arr.simps(4) Can.simps(4))
      ultimately show ?thesis
        using can-mkarr-Can 7 mkarr-memb(2) by metis
    qed
    moreover have can (cod (mkarr ?t · f))
      using 7 ide-implies-can by force
    ultimately have mkarr ?t · f = cod (mkarr ?t · f)
      using can-coherence by meson
    thus ?thesis
      using 7 can-implies-arr ide-cod by metis
  qed
qed
```

```
lemma inv-mkarr [simp]:
assumes Can t
shows inv (mkarr t) = mkarr (Inv t)
proof −
  have t: Can t ∧ Arr t ∧ Can (Inv t) ∧ Arr (Inv t) ∧ Ide (Dom t) ∧ Ide (Cod t)
    using assms Can-implies-Arr Arr-implies-Ide-Dom Arr-implies-Ide-Cod
          Inv-preserves-Can
    by simp
  have inverse-arrows (mkarr t) (mkarr (Inv t))
  proof
    show ide (mkarr t · mkarr (Inv t))
    proof −
      have mkarr (Cod t) = mkarr (Comp t (Inv t))
     using t Inv-in-Hom Ide-in-Hom Diagonalize-Inv Diag-Diagonalize Diagonalize-preserves-Can
        by (intro mkarr-eqI, auto)
      also have ... = mkarr t · mkarr (Inv t)
        using t comp-mkarr Inv-in-Hom by simp
      finally have mkarr (Cod t) = mkarr t · mkarr (Inv t)
        by blast
      thus ?thesis using t ide-mkarr-Ide [of Cod t] by simp
    qed
    show ide (mkarr (Inv t) · mkarr t)
    proof −
      have mkarr (Dom t) = mkarr (Inv t · t)
     using t Inv-in-Hom Ide-in-Hom Diagonalize-Inv Diag-Diagonalize Diagonalize-preserves-Can
        by (intro mkarr-eqI, auto)
      also have ... = mkarr (Inv t) · mkarr t
        using t comp-mkarr Inv-in-Hom by simp
      finally have mkarr (Dom t) = mkarr (Inv t) · mkarr t
        by blast
      thus ?thesis using t ide-mkarr-Ide [of Dom t] by simp
    qed
  qed
  thus ?thesis using inverse-unique by auto
qed

lemma iso-can:
assumes can f
shows iso f
  using assms inverse-arrows-can by auto
```

The following function produces the unique canonical arrow between two given objects, if such an arrow exists.

```
definition mkcan
where mkcan a b = mkarr (Inv (COD b↓) · (DOM a↓))

lemma can-mkcan:
assumes ide a and ide b and ⌊DOM a⌋ = ⌊COD b⌋
```

shows *can* (*mkcan a b*) **and** «*mkcan a b* : *a* → *b*»
proof −
  show *can* (*mkcan a b*)
    using *assms mkcan-def Arr-rep Arr-implies-Ide-Dom Arr-implies-Ide-Cod Can-red Inv-preserves-Can can-mkarr-Can*
    by *simp*
  show «*mkcan a b* : *a* → *b*»
  using *assms mkcan-def Arr-rep Arr-implies-Ide-Dom Arr-implies-Ide-Cod Can-red Inv-in-Hom dom-char* [*of a*] *cod-char* [*of b*] *mkarr-rep mkarr-in-hom can-implies-arr*
    by *auto*
qed

lemma *dom-mkcan*:
assumes *ide a* **and** *ide b* **and** ⌊*DOM a*⌋ = ⌊*COD b*⌋
shows *dom* (*mkcan a b*) = *a*
  using *assms can-mkcan* **by** *blast*

lemma *cod-mkcan*:
assumes *ide a* **and** *ide b* **and** ⌊*DOM a*⌋ = ⌊*COD b*⌋
shows *cod* (*mkcan a b*) = *b*
  using *assms can-mkcan* **by** *blast*

lemma *can-coherence'*:
assumes *can f*
shows *mkcan* (*dom f*) (*cod f*) = *f*
proof −
  have *Ide* ⌊*rep f*⌋
    using *assms Ide-Diagonalize-Can Can-rep-can* **by** *simp*
  hence *Dom* ⌊*rep f*⌋ = *Cod* ⌊*rep f*⌋
    using *Ide-in-Hom* **by** *simp*
  hence ⌊*DOM f*⌋ = ⌊*COD f*⌋
    using *assms can-implies-arr Arr-rep Diagonalize-in-Hom* **by** *simp*
  moreover have *DOM f* = *DOM* (*dom f*)
    using *assms can-implies-arr dom-char rep-mkarr Arr-implies-Ide-Dom Ide-implies-Arr Par-Arr-norm* [*of DOM f*] *Ide-in-Hom*
    by *auto*
  moreover have *COD f* = *COD* (*cod f*)
    using *assms can-implies-arr cod-char rep-mkarr Arr-implies-Ide-Cod Ide-implies-Arr Par-Arr-norm* [*of COD f*] *Ide-in-Hom*
    by *auto*
  ultimately have *can* (*mkcan* (*dom f*) (*cod f*)) ∧ *par f* (*mkcan* (*dom f*) (*cod f*))
    using *assms can-implies-arr can-mkcan dom-mkcan cod-mkcan* **by** *simp*
  thus *?thesis* **using** *assms can-coherence* **by** *blast*
qed

lemma *Ide-Diagonalize-rep-ide*:
assumes *ide a*
shows *Ide* ⌊*rep a*⌋
  using *assms ide-implies-can can-def-alt rep-in-arr* **by** *simp*

lemma *Diagonalize-DOM*:
assumes *arr f*
shows $\lfloor DOM\ f \rfloor = Dom\ \lfloor rep\ f \rfloor$
using *assms Diag-Diagonalize* **by** *simp*

lemma *Diagonalize-COD*:
assumes *arr f*
shows $\lfloor COD\ f \rfloor = Cod\ \lfloor rep\ f \rfloor$
using *assms Diag-Diagonalize* **by** *simp*

lemma *Diagonalize-rep-preserves-seq*:
assumes *seq g f*
shows *Seq* $\lfloor rep\ g \rfloor$ $\lfloor rep\ f \rfloor$
using *assms Diagonalize-DOM Diagonalize-COD Diag-implies-Arr Diag-Diagonalize*(*1*)
rep-preserves-seq
by *force*

lemma *Dom-Diagonalize-rep*:
assumes *arr f*
shows $Dom\ \lfloor rep\ f \rfloor = \lfloor rep\ (dom\ f) \rfloor$
using *assms Diagonalize-rep-preserves-seq* [*of f dom f*] *Ide-Diagonalize-rep-ide Ide-in-Hom*
by *simp*

lemma *Cod-Diagonalize-rep*:
assumes *arr f*
shows $Cod\ \lfloor rep\ f \rfloor = \lfloor rep\ (cod\ f) \rfloor$
using *assms Diagonalize-rep-preserves-seq* [*of cod f f*] *Ide-Diagonalize-rep-ide Ide-in-Hom*
by *simp*

lemma *mkarr-Diagonalize-rep*:
assumes *arr f* **and** *Diag* (*DOM f*) **and** *Diag* (*COD f*)
shows $mkarr\ \lfloor rep\ f \rfloor = f$
proof −
have $mkarr\ (rep\ f) = mkarr\ \lfloor rep\ f \rfloor$
using *assms rep-in-Hom Diagonalize-in-Hom Diag-Diagonalize Diagonalize-Diag*
by (*intro mkarr-eqI*, *simp-all*)
thus *?thesis* **using** *assms mkarr-rep* **by** *auto*
qed

We define tensor product of arrows via the constructor ($\otimes$) on terms.

definition $tensor_{FMC}$ (**infixr** $\otimes$ *53*)
where $f \otimes g \equiv (if\ arr\ f \wedge arr\ g\ then\ mkarr\ (rep\ f \otimes rep\ g)\ else\ null)$

lemma *arr-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *arr* $(f \otimes g)$
using *assms* $tensor_{FMC}$*-def arr-mkarr* **by** *simp*

lemma *rep-tensor*:
assumes *arr f* **and** *arr g*
shows *rep* (*f* ⊗ *g*) = ‖*rep f* ⨂ *rep g*‖
using *assms tensor*$_{FMC}$*-def rep-mkarr* **by** *simp*

lemma *Par-memb-rep*:
assumes *arr f* **and** *t* ∈ *f*
shows *Par t* (*rep f*)
using *assms mkarr-memb* **apply** *simp*
using *rep-in-Hom Dom-memb Cod-memb* **by** *metis*

lemma *Tensor-in-tensor* [*intro*]:
assumes *arr f* **and** *arr g* **and** *t* ∈ *f* **and** *u* ∈ *g*
shows *t* ⊗ *u* ∈ *f* ⊗ *g*
proof −
have *equiv* (*t* ⊗ *u*) (*rep f* ⨂ *rep g*)
proof −
have *1*: *Par* (*t* ⊗ *u*) (*rep f* ⨂ *rep g*)
proof −
have *Par t* (*rep f*) ∧ *Par u* (*rep g*) **using** *assms Par-memb-rep* **by** *blast*
thus *?thesis* **by** *simp*
qed
moreover have ⌊*t* ⊗ *u*⌋ = ⌊*rep f* ⨂ *rep g*⌋
using *assms 1 equiv-iff-eq-norm rep-mkarr norm-norm mkarr-memb*(*2*)
by (*metis Arr.simps*(*3*) *Diagonalize.simps*(*3*))
ultimately show *?thesis* **by** *simp*
qed
thus *?thesis*
using *assms tensor*$_{FMC}$*-def mkarr-def* **by** *simp*
qed

lemma *DOM-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *DOM* (*f* ⊗ *g*) = *DOM f* ⨂ *DOM g*
by (*metis* (*no-types*, *lifting*) *DOM-mkarr Dom.simps*(*3*) *mkarr-extensionality arr-char arr-tensor assms*(*1*) *assms*(*2*) *tensor*$_{FMC}$*-def*)

lemma *COD-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *COD* (*f* ⊗ *g*) = *COD f* ⨂ *COD g*
by (*metis* (*no-types*, *lifting*) *COD-mkarr Cod.simps*(*3*) *mkarr-extensionality arr-char arr-tensor assms*(*1*) *assms*(*2*) *tensor*$_{FMC}$*-def*)

lemma *tensor-in-hom* [*simp*]:
assumes «*f* : *a* → *b*» **and** «*g* : *c* → *d*»
shows «*f* ⊗ *g* : *a* ⊗ *c* → *b* ⊗ *d*»
proof −
have *f*: *arr f* ∧ *dom f* = *a* ∧ *cod f* = *b* **using** *assms*(*1*) **by** *auto*
have *g*: *arr g* ∧ *dom g* = *c* ∧ *cod g* = *d* **using** *assms*(*2*) **by** *auto*

```
  have dom (f ⊗ g) = dom f ⊗ dom g
    using f g arr-tensor dom-char Tensor-in-tensor [of dom f dom g DOM f DOM g]
          DOM-in-dom mkarr-memb(2) DOM-tensor arr-dom-iff-arr
    by metis
  moreover have cod (f ⊗ g) = cod f ⊗ cod g
    using f g arr-tensor cod-char Tensor-in-tensor [of cod f cod g COD f COD g]
          COD-in-cod mkarr-memb(2) COD-tensor arr-cod-iff-arr
    by metis
  ultimately show ?thesis using assms arr-tensor by blast
qed

lemma dom-tensor [simp]:
assumes arr f and arr g
shows dom (f ⊗ g) = dom f ⊗ dom g
  using assms tensor-in-hom [of f] by blast

lemma cod-tensor [simp]:
assumes arr f and arr g
shows cod (f ⊗ g) = cod f ⊗ cod g
  using assms tensor-in-hom [of f] by blast

lemma tensor-mkarr [simp]:
assumes Arr t and Arr u
shows mkarr t ⊗ mkarr u = mkarr (t ⨂ u)
  using assms by (meson Tensor-in-tensor arr-char Arr-in-mkarr arr-mkarr arr-tensor)

lemma tensor-preserves-ide:
assumes ide a and ide b
shows ide (a ⊗ b)
proof −
  have can (a ⊗ b)
    using assms tensor_FMC-def Can-rep-can ide-implies-can can-mkarr-Can by simp
  moreover have dom (a ⊗ b) = cod (a ⊗ b)
    using assms tensor-in-hom by simp
  ultimately show ?thesis using ide-char by metis
qed

lemma tensor-preserves-can:
assumes can f and can g
shows can (f ⊗ g)
  using assms can-implies-arr Can-rep-can tensor_FMC-def can-mkarr-Can by simp

lemma comp-preserves-can:
assumes can f and can g and dom f = cod g
shows can (f · g)
proof −
  have 1: ARR f ∧ ARR g ∧ DOM f = COD g
    using assms can-implies-arr arr-iff-ARR Arr-implies-Ide-Cod Arr-implies-Ide-Dom
          mkarr-inj-on-Ide cod-char dom-char
```

by *simp*
hence *Can* (*rep f* ⋆ *rep g*)
using *assms can-implies-arr Can-rep-can* **by** *force*
thus *?thesis*
using *assms 1 can-implies-arr comp-char can-mkarr-Can seq-char′* **by** *simp*
qed

The remaining structure required of a monoidal category is also defined syntactically.

definition $unity_{FMC}$ :: *'c arr* ($\mathcal{I}$)
where $\mathcal{I}$ = *mkarr* $\boldsymbol{\mathcal{I}}$

definition $lunit_{FMC}$ :: *'c arr* ⇒ *'c arr* (l[-])
where l[*a*] = *mkarr* **l**[*rep a*]

definition $runit_{FMC}$ :: *'c arr* ⇒ *'c arr* (r[-])
where r[*a*] = *mkarr* **r**[*rep a*]

definition $assoc_{FMC}$ :: *'c arr* ⇒ *'c arr* ⇒ *'c arr* ⇒ *'c arr* (a[-, -, -])
where a[*a*, *b*, *c*] = *mkarr* **a**[*rep a*, *rep b*, *rep c*]

lemma *can-lunit*:
assumes *ide a*
shows *can* l[*a*]
using *assms* $lunit_{FMC}$*-def can-mkarr-Can*
by (*simp add*: *Can-rep-can ide-implies-can*)

lemma *lunit-in-hom*:
assumes *ide a*
shows «l[*a*] : $\mathcal{I}$ ⊗ *a* → *a*»
proof −
have *dom* l[*a*] = $\mathcal{I}$ ⊗ *a*
using *assms* $lunit_{FMC}$*-def* $unity_{FMC}$*-def Ide-implies-Arr dom-mkarr dom-char tensor-mkarr*
Arr-rep
by (*metis Arr.simps*(*2*) *Arr.simps*(*5*) *Arr-implies-Ide-Dom Dom.simps*(*5*)
ideD(*1*) *ideD*(*2*))
moreover have *cod* l[*a*] = *a*
using *assms* $lunit_{FMC}$*-def rep-in-arr*(*1*) *cod-mkarr cod-char ideD*(*3*) **by** *auto*
ultimately show *?thesis*
using *assms arr-cod-iff-arr* **by** (*intro in-homI*, *fastforce*)
qed

lemma *arr-lunit* [*simp*]:
assumes *ide a*
shows *arr* l[*a*]
using *assms can-lunit can-implies-arr* **by** *simp*

lemma *dom-lunit* [*simp*]:
assumes *ide a*

shows *dom* $\mathrm{l}[a] = \mathcal{I} \otimes a$
using *assms lunit-in-hom* **by** *auto*

lemma *cod-lunit* [*simp*]:
assumes *ide a*
shows *cod* $\mathrm{l}[a] = a$
using *assms lunit-in-hom* **by** *auto*

lemma *can-runit*:
assumes *ide a*
shows *can* $\mathrm{r}[a]$
using *assms* $runit_{FMC}$*-def can-mkarr-Can*
by (*simp add*: *Can-rep-can ide-implies-can*)

lemma *runit-in-hom* [*simp*]:
assumes *ide a*
shows «$\mathrm{r}[a] : a \otimes \mathcal{I} \to a$»
proof −
have *dom* $\mathrm{r}[a] = a \otimes \mathcal{I}$
using *assms Arr-rep Arr.simps*(*2*) *Arr.simps*(*7*) *Arr-implies-Ide-Dom Dom.simps*(*7*)
Ide-implies-Arr dom-mkarr dom-char ideD(*1*) *ideD*(*2*) $runit_{FMC}$*-def tensor-mkarr*
$unity_{FMC}$*-def*
by *metis*
moreover have *cod* $\mathrm{r}[a] = a$
using *assms* $runit_{FMC}$*-def rep-in-arr*(*1*) *cod-mkarr cod-char ideD*(*3*) **by** *auto*
ultimately show *?thesis*
using *assms arr-cod-iff-arr* **by** (*intro in-homI*, *fastforce*)
qed

lemma *arr-runit* [*simp*]:
assumes *ide a*
shows *arr* $\mathrm{r}[a]$
using *assms can-runit can-implies-arr* **by** *simp*

lemma *dom-runit* [*simp*]:
assumes *ide a*
shows *dom* $\mathrm{r}[a] = a \otimes \mathcal{I}$
using *assms runit-in-hom* **by** *blast*

lemma *cod-runit* [*simp*]:
assumes *ide a*
shows *cod* $\mathrm{r}[a] = a$
using *assms runit-in-hom* **by** *blast*

lemma *can-assoc*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *can* $\mathrm{a}[a, b, c]$
using *assms* $assoc_{FMC}$*-def can-mkarr-Can*
by (*simp add*: *Can-rep-can ide-implies-can*)

lemma *assoc-in-hom*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «a[*a*, *b*, *c*] : (*a* ⊗ *b*) ⊗ *c* → *a* ⊗ *b* ⊗ *c*»
proof −
  have *dom* a[*a*, *b*, *c*] = (*a* ⊗ *b*) ⊗ *c*
  proof −
    have *dom* a[*a*, *b*, *c*] = *mkarr* (*Dom* **a**[*rep a*, *rep b*, *rep c*])
      using *assms assoc*$_{FMC}$*-def rep-in-arr*(*1*) **by** *simp*
    also have ... = *mkarr* ((*DOM a* ⊗ *DOM b*) ⊗ *DOM c*)
      by *simp*
    also have ... = (*a* ⊗ *b*) ⊗ *c*
      by (*metis mkarr-extensionality arr-tensor assms dom-char*
          ideD(*1*) *ideD*(*2*) *not-arr-null null-char tensor-mkarr*)
    finally show *?thesis* **by** *blast*
  qed
  moreover have *cod* a[*a*, *b*, *c*] = *a* ⊗ *b* ⊗ *c*
  proof −
    have *cod* a[*a*, *b*, *c*] = *mkarr* (*Cod* **a**[*rep a*, *rep b*, *rep c*])
      using *assms assoc*$_{FMC}$*-def rep-in-arr*(*1*) **by** *simp*
    also have ... = *mkarr* (*COD a* ⊗ *COD b* ⊗ *COD c*)
      by *simp*
    also have ... = *a* ⊗ *b* ⊗ *c*
      by (*metis mkarr-extensionality arr-tensor assms*(*1*) *assms*(*2*) *assms*(*3*) *cod-char*
          ideD(*1*) *ideD*(*3*) *not-arr-null null-char tensor-mkarr*)
    finally show *?thesis* **by** *blast*
  qed
  moreover have *arr* a[*a*, *b*, *c*]
    using *assms assoc*$_{FMC}$*-def rep-in-arr*(*1*) *arr-mkarr* **by** *simp*
  ultimately show *?thesis* **by** *auto*
qed

lemma *arr-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *arr* a[*a*, *b*, *c*]
  using *assms can-assoc can-implies-arr* **by** *simp*

lemma *dom-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *dom* a[*a*, *b*, *c*] = (*a* ⊗ *b*) ⊗ *c*
  using *assms assoc-in-hom* **by** *blast*

lemma *cod-assoc* [*simp*]:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *cod* a[*a*, *b*, *c*] = *a* ⊗ *b* ⊗ *c*
  using *assms assoc-in-hom* **by** *blast*

lemma *ide-unity* [*simp*]:
shows *ide* $\mathcal{I}$

using $unity_{FMC}$-*def Arr.simps*(2) *Dom.simps*(2) *arr-mkarr dom-mkarr ide-dom*
by *metis*

lemma *Unity-in-unity* [*simp*]:
shows $\boldsymbol{\mathcal{I}} \in \mathcal{I}$
using $unity_{FMC}$-*def Arr-in-mkarr* **by** *simp*

lemma *rep-unity* [*simp*]:
shows *rep* $\mathcal{I} = \|\boldsymbol{\mathcal{I}}\|$
using $unity_{FMC}$-*def rep-mkarr* **by** *simp*

lemma *Lunit-in-lunit* [*intro*]:
assumes *arr f* **and** $t \in f$
shows $\mathbf{l}[t] \in \mathrm{l}[f]$
proof −
have *Arr t* ∧ *Arr* (*rep f*) ∧ *Dom t* = *DOM f* ∧ *Cod t* = *COD f* ∧ ⌊*t*⌋ = ⌊*rep f*⌋
using *assms*
by (*metis mkarr-memb*(1) *mkarr-memb*(2) *rep-mkarr rep-in-arr*(1) *equiv-iff-eq-norm norm-rep*)
thus *?thesis*
by (*simp add*: *mkarr-def* $lunit_{FMC}$-*def*)
qed

lemma *Runit-in-runit* [*intro*]:
assumes *arr f* **and** $t \in f$
shows $\mathbf{r}[t] \in \mathrm{r}[f]$
proof −
have *Arr t* ∧ *Arr* (*rep f*) ∧ *Dom t* = *DOM f* ∧ *Cod t* = *COD f* ∧ ⌊*t*⌋ = ⌊*rep f*⌋
using *assms*
by (*metis mkarr-memb*(1) *mkarr-memb*(2) *rep-mkarr rep-in-arr*(1) *equiv-iff-eq-norm norm-rep*)
thus *?thesis*
by (*simp add*: *mkarr-def* $runit_{FMC}$-*def*)
qed

lemma *Assoc-in-assoc* [*intro*]:
assumes *arr f* **and** *arr g* **and** *arr h*
and $t \in f$ **and** $u \in g$ **and** $v \in h$
shows $\mathbf{a}[t, u, v] \in \mathrm{a}[f, g, h]$
proof −
have *Arr t* ∧ *Arr* (*rep f*) ∧ *Dom t* = *DOM f* ∧ *Cod t* = *COD f* ∧ ⌊*t*⌋ = ⌊*rep f*⌋
using *assms*
by (*metis mkarr-memb*(1) *rep-mkarr rep-in-arr*(1) *equiv-iff-eq-norm mkarr-memb*(2) *norm-rep*)
moreover have *Arr u* ∧ *Arr* (*rep g*) ∧ *Dom u* = *DOM g* ∧ *Cod u* = *COD g* ∧ ⌊*u*⌋ = ⌊*rep g*⌋
using *assms*
by (*metis mkarr-memb*(1) *rep-mkarr rep-in-arr*(1) *equiv-iff-eq-norm mkarr-memb*(2) *norm-rep*)

moreover have *Arr v* ∧ *Arr* (*rep h*) ∧ *Dom v* = *DOM h* ∧ *Cod v* = *COD h* ∧
⌊*v*⌋ = ⌊*rep h*⌋
using *assms*
by (*metis mkarr-memb*(*1*) *rep-mkarr rep-in-arr*(*1*) *equiv-iff-eq-norm mkarr-memb*(*2*)
norm-rep)
ultimately show *?thesis*
using *assoc*$_{FMC}$*-def mkarr-def* **by** *simp*
qed

At last, we can show that we've constructed a monoidal category.

interpretation *EMC*: *elementary-monoidal-category*
comp tensor$_{FMC}$ *unity*$_{FMC}$ *lunit*$_{FMC}$ *runit*$_{FMC}$ *assoc*$_{FMC}$
proof
show *ide* $\mathcal{I}$ **using** *ide-unity* **by** *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ «l[*a*] : $\mathcal{I} \otimes a \to a$» **by** *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ «r[*a*] : $a \otimes \mathcal{I} \to a$» **by** *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ *iso* l[*a*] **using** *can-lunit iso-can* **by** *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ *iso* r[*a*] **using** *can-runit iso-can* **by** *auto*
show $\bigwedge a\ b\ c.$ ⟦ *ide a*; *ide b*; *ide c* ⟧ $\Longrightarrow$ «a[*a*, *b*, *c*] : $(a \otimes b) \otimes c \to a \otimes b \otimes c$» **by** *auto*
show $\bigwedge a\ b\ c.$ ⟦ *ide a*; *ide b*; *ide c* ⟧ $\Longrightarrow$ *iso* a[*a*, *b*, *c*] **using** *can-assoc iso-can* **by** *auto*
show $\bigwedge a\ b.$ ⟦ *ide a*; *ide b* ⟧ $\Longrightarrow$ *ide* $(a \otimes b)$ **using** *tensor-preserves-ide* **by** *auto*
fix *f a b g c d*
show ⟦ «$f : a \to b$»; «$g : c \to d$» ⟧ $\Longrightarrow$ «$f \otimes g : a \otimes c \to b \otimes d$»
using *tensor-in-hom* **by** *auto*
next

Naturality of left unitor.

fix *f*
assume *f*: *arr f*
show l[*cod f*] · ($\mathcal{I} \otimes f$) = *f* · l[*dom f*]
proof (*intro arr-eqI*)
show *par* (l[*cod f*] · ($\mathcal{I} \otimes f$)) (*f* · l[*dom f*])
using *f* **by** *simp*
show **l**[*COD f*] · ($\boldsymbol{\mathcal{I}} \otimes$ *rep f*) ∈ l[*cod f*] · ($\mathcal{I} \otimes f$)
using *f* **by** *fastforce*
show *rep f* · **l**[*DOM f*] ∈ *f* · l[*dom f*]
using *f* **by** *fastforce*
show ⌊**l**[*COD f*] · ($\boldsymbol{\mathcal{I}} \otimes$ *rep f*)⌋ = ⌊*rep f* · **l**[*DOM f*]⌋
using *f* **by** (*simp add*: *Diag-Diagonalize*(*1*) *Diagonalize-DOM Diagonalize-COD*)
qed

Naturality of right unitor.

show r[*cod f*] · ($f \otimes \mathcal{I}$) = *f* · r[*dom f*]
proof (*intro arr-eqI*)
show *par* (r[*cod f*] · ($f \otimes \mathcal{I}$)) (*f* · r[*dom f*])
using *f* **by** *simp*
show **r**[*COD f*] · (*rep f* ⊗ $\boldsymbol{\mathcal{I}}$) ∈ r[*cod f*] · ($f \otimes \mathcal{I}$)
using *f* **by** *fastforce*
show *rep f* · **r**[*DOM f*] ∈ *f* · r[*dom f*]

```
    using f by fastforce
  show ⌊r[COD f] · (rep f ⊗ 𝓘)⌋ = ⌊rep f · r[DOM f]⌋
    using f by (simp add: Diag-Diagonalize(1) Diagonalize-DOM Diagonalize-COD)
qed
next
```

Naturality of associator.

```
fix f0 :: 'c arr and f1 f2
assume f0: arr f0 and f1: arr f1 and f2: arr f2
show a[cod f0, cod f1, cod f2] · ((f0 ⊗ f1) ⊗ f2)
       = (f0 ⊗ f1 ⊗ f2) · a[dom f0, dom f1, dom f2]
proof (intro arr-eqI)
  show 1: par (a[cod f0, cod f1, cod f2] · ((f0 ⊗ f1) ⊗ f2))
              ((f0 ⊗ f1 ⊗ f2) · a[dom f0, dom f1, dom f2])
    using f0 f1 f2 by force
  show a[rep (cod f0), rep (cod f1), rep (cod f2)] · ((rep f0 ⊗ rep f1) ⊗ rep f2)
         ∈ a[cod f0, cod f1, cod f2] · ((f0 ⊗ f1) ⊗ f2)
    using f0 f1 f2 by fastforce
  show (rep f0 ⊗ rep f1 ⊗ rep f2) · a[rep (dom f0), rep (dom f1), rep (dom f2)]
         ∈ (f0 ⊗ f1 ⊗ f2) · a[dom f0, dom f1, dom f2]
    using f0 f1 f2 by fastforce
  show ⌊a[rep (cod f0), rep (cod f1), rep (cod f2)] · ((rep f0 ⊗ rep f1) ⊗ rep f2)⌋
         = ⌊(rep f0 ⊗ rep f1 ⊗ rep f2) · a[rep (dom f0), rep (dom f1), rep (dom f2)]⌋
  proof −
    have ⌊a[rep (cod f0), rep (cod f1), rep (cod f2)] · ((rep f0 ⊗ rep f1) ⊗ rep f2)⌋
           = ⌊rep f0⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep f2⌋
    proof −
      have b0: ⌊rep (cod f0)⌋ = Cod ⌊rep f0⌋
        using f0 Cod-Diagonalize-rep by simp
      have b1: ⌊rep (cod f1)⌋ = Cod ⌊rep f1⌋
        using f1 Cod-Diagonalize-rep by simp
      have b2: ⌊rep (cod f2)⌋ = Cod ⌊rep f2⌋
        using f2 Cod-Diagonalize-rep by simp
      have ⌊a[rep (cod f0), rep (cod f1), rep (cod f2)] · ((rep f0 ⊗ rep f1) ⊗ rep f2)⌋
             = (⌊rep (cod f0)⌋ ⌊⊗⌋ ⌊rep (cod f1)⌋ ⌊⊗⌋ ⌊rep (cod f2)⌋) ⌊·⌋
               (⌊rep f0⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep f2⌋)
        using f0 f1 f2 using Diag-Diagonalize(1) TensorDiag-assoc by auto
      also have ... = ⌊rep (cod f0)⌋ ⌊·⌋ ⌊rep f0⌋ ⌊⊗⌋
                      ⌊rep (cod f1)⌋ ⌊·⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep (cod f2)⌋ ⌊·⌋ ⌊rep f2⌋
      proof −
        have Seq ⌊rep (cod f0)⌋ ⌊rep f0⌋ ∧ Seq ⌊rep (cod f1)⌋ ⌊rep f1⌋ ∧
             Seq ⌊rep (cod f2)⌋ ⌊rep f2⌋
       using f0 f1 f2 rep-in-Hom Diagonalize-in-Hom Dom-Diagonalize-rep Cod-Diagonalize-rep
          by auto
        thus ?thesis
          using f0 f1 f2 b0 b1 b2 TensorDiag-in-Hom TensorDiag-preserves-Diag
                Diag-Diagonalize Arr-implies-Ide-Dom Arr-implies-Ide-Cod
                CompDiag-TensorDiag
            by simp
```

```
  qed
  also have ... = ⌊rep f0⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep f2⌋
  proof −
    have ⌊rep (cod f0)⌋ ⌊⋅⌋ ⌊rep f0⌋ = ⌊rep f0⌋
      using f0 b0 CompDiag-Cod-Diag [of ⌊rep f0⌋] Diag-Diagonalize
      by simp
    moreover have ⌊rep (cod f1)⌋ ⌊⋅⌋ ⌊rep f1⌋ = ⌊rep f1⌋
      using f1 b1 CompDiag-Cod-Diag [of ⌊rep f1⌋] Diag-Diagonalize
      by simp
    moreover have ⌊rep (cod f2)⌋ ⌊⋅⌋ ⌊rep f2⌋ = ⌊rep f2⌋
      using f2 b2 CompDiag-Cod-Diag [of ⌊rep f2⌋] Diag-Diagonalize
      by simp
    ultimately show ?thesis by argo
  qed
  finally show ?thesis by blast
qed
also have ... = ⌊(rep f0 ⊗ rep f1 ⊗ rep f2) ⋅
                  a[rep (dom f0), rep (dom f1), rep (dom f2)]⌋
proof −
  have a0: ⌊rep (dom f0)⌋ = Dom ⌊rep f0⌋
    using f0 Dom-Diagonalize-rep by simp
  have a1: ⌊rep (dom f1)⌋ = Dom ⌊rep f1⌋
    using f1 Dom-Diagonalize-rep by simp
  have a2: ⌊rep (dom f2)⌋ = Dom ⌊rep f2⌋
    using f2 Dom-Diagonalize-rep by simp
  have ⌊(rep f0 ⊗ rep f1 ⊗ rep f2) ⋅ a[rep (dom f0), rep (dom f1), rep (dom f2)]⌋
        = (⌊rep f0⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep f2⌋) ⌊⋅⌋
          (⌊rep (dom f0)⌋ ⌊⊗⌋ ⌊rep (dom f1)⌋ ⌊⊗⌋ ⌊rep (dom f2)⌋)
    using f0 f1 f2 using Diag-Diagonalize(1) TensorDiag-assoc by auto
  also have ... = ⌊rep f0⌋ ⌊⋅⌋ ⌊rep (dom f0)⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⋅⌋ ⌊rep (dom f1)⌋ ⌊⊗⌋
                  ⌊rep f2⌋ ⌊⋅⌋ ⌊rep (dom f2)⌋
  proof −
    have Seq ⌊rep f0⌋ ⌊rep (dom f0)⌋ ∧ Seq ⌊rep f1⌋ ⌊rep (dom f1)⌋ ∧
         Seq ⌊rep f2⌋ ⌊rep (dom f2)⌋
   using f0 f1 f2 rep-in-Hom Diagonalize-in-Hom Dom-Diagonalize-rep Cod-Diagonalize-rep
      by auto
    thus ?thesis
      using f0 f1 f2 a0 a1 a2 TensorDiag-in-Hom TensorDiag-preserves-Diag
            Diag-Diagonalize Arr-implies-Ide-Dom Arr-implies-Ide-Cod
            CompDiag-TensorDiag
      by force
  qed
  also have ... = ⌊rep f0⌋ ⌊⊗⌋ ⌊rep f1⌋ ⌊⊗⌋ ⌊rep f2⌋
  proof −
    have ⌊rep f0⌋ ⌊⋅⌋ ⌊rep (dom f0)⌋ = ⌊rep f0⌋
      using f0 a0 CompDiag-Diag-Dom [of Diagonalize (rep f0)] Diag-Diagonalize
      by simp
    moreover have ⌊rep f1⌋ ⌊⋅⌋ ⌊rep (dom f1)⌋ = ⌊rep f1⌋
      using f1 a1 CompDiag-Diag-Dom [of Diagonalize (rep f1)] Diag-Diagonalize
```

```
        by simp
      moreover have ⌊rep f2⌋ ⌊·⌋ ⌊rep (dom f2)⌋ = ⌊rep f2⌋
        using f2 a2 CompDiag-Diag-Dom [of Diagonalize (rep f2)] Diag-Diagonalize
        by simp
      ultimately show ?thesis by argo
    qed
    finally show ?thesis by argo
  qed
  finally show ?thesis by blast
 qed
qed
next
```

Tensor preserves composition (interchange).

```
fix f g f' g'
show ⟦ seq g f; seq g' f' ⟧ ⟹ (g ⊗ g') · (f ⊗ f') = g · f ⊗ g' · f'
proof −
  assume gf: seq g f
  assume gf': seq g' f'
  show ?thesis
  proof (intro arr-eqI)
    show par ((g ⊗ g') · (f ⊗ f')) (g · f ⊗ g' · f')
      using gf gf' by fastforce
    show (rep g ⊗ rep g') · (rep f ⊗ rep f') ∈ (g ⊗ g') · (f ⊗ f')
      using gf gf' by force
    show rep g · rep f ⊗ rep g' · rep f' ∈ g · f ⊗ g' · f'
     using gf gf'
     by (meson Comp-in-comp-ARR Tensor-in-tensor rep-in-arr seqE seq-char')
    show ⌊(rep g ⊗ rep g') · (rep f ⊗ rep f')⌋ = ⌊rep g · rep f ⊗ rep g' · rep f'⌋
    proof −
      have ⌊(rep g ⊗ rep g') · (rep f ⊗ rep f')⌋
            = (⌊rep g⌋ ⌊⊗⌋ ⌊rep g'⌋) ⌊·⌋ (⌊rep f⌋ ⌊⊗⌋ ⌊rep f'⌋)
        by auto
      also have ... =  ⌊rep g⌋ ⌊·⌋ ⌊rep f⌋ ⌊⊗⌋ ⌊rep g'⌋ ⌊·⌋ ⌊rep f'⌋
        using gf gf' Arr-rep Diagonalize-rep-preserves-seq
              CompDiag-TensorDiag [of ⌊rep g⌋ ⌊rep g'⌋ ⌊rep f⌋ ⌊rep f'⌋]
              Diag-Diagonalize Diagonalize-DOM Diagonalize-COD
        by force
      also have ... = ⌊rep g · rep f ⊗ rep g' · rep f'⌋
        by auto
      finally show ?thesis by blast
    qed
  qed
qed
next
```

The triangle.

```
fix a b
assume a: ide a
```

```
  assume b: ide b
  show (a ⊗ l[b]) · a[a, ℐ, b] = r[a] ⊗ b
  proof −
    have par ((a ⊗ l[b]) · a[a, ℐ, b]) (r[a] ⊗ b)
      using a b by simp
    moreover have can ((a ⊗ l[b]) · a[a, ℐ, b])
      using a b ide-implies-can comp-preserves-can tensor-preserves-can can-assoc can-lunit
      by simp
    moreover have can (r[a] ⊗ b)
      using a b ide-implies-can can-runit tensor-preserves-can by simp
    ultimately show ?thesis using can-coherence by blast
  qed
  next
```

The pentagon.

```
  fix a b c d
  assume a: ide a
  assume b: ide b
  assume c: ide c
  assume d: ide d
  show (a ⊗ a[b, c, d]) · a[a, b ⊗ c, d] · (a[a, b, c] ⊗ d)
         = a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
  proof −
    let ?LHS = (a ⊗ a[b, c, d]) · a[a, b ⊗ c, d] · (a[a, b, c] ⊗ d)
    let ?RHS = a[a, b, c ⊗ d] · a[a ⊗ b, c, d]
    have par ?LHS ?RHS
      using a b c d can-assoc tensor-preserves-ide by auto
    moreover have can ?LHS
      using a b c d ide-implies-can comp-preserves-can tensor-preserves-can can-assoc
          tensor-preserves-ide
      by simp
    moreover have can ?RHS
      using a b c d comp-preserves-can tensor-preserves-can can-assoc tensor-in-hom
          tensor-preserves-ide
      by simp
    ultimately show ?thesis using can-coherence by blast
  qed
qed

lemma is-elementary-monoidal-category:
shows elementary-monoidal-category
        comp tensor_FMC unity_FMC lunit_FMC runit_FMC assoc_FMC
  ..

abbreviation T_FMC where T_FMC ≡ EMC.T
abbreviation α_FMC where α_FMC ≡ EMC.α
abbreviation ι_FMC where ι_FMC ≡ EMC.ι

interpretation MC: monoidal-category comp T_FMC α_FMC ι_FMC
```

using *EMC.induces-monoidal-category* **by** *auto*

lemma *induces-monoidal-category*:
shows *monoidal-category comp* T_{FMC} α_{FMC} ι_{FMC}
..

end

sublocale *free-monoidal-category* $\subseteq$
elementary-monoidal-category
comp tensor$_{FMC}$ *unity*$_{FMC}$ *lunit*$_{FMC}$ *runit*$_{FMC}$ *assoc*$_{FMC}$
using *is-elementary-monoidal-category* **by** *auto*

sublocale *free-monoidal-category* $\subseteq$ *monoidal-category comp* T_{FMC} α_{FMC} ι_{FMC}
using *induces-monoidal-category* **by** *auto*

4.2 Proof of Freeness

Now we proceed on to establish the freeness of $\mathcal{F}C$: each functor from C to a monoidal category D extends uniquely to a strict monoidal functor from $\mathcal{F}C$ to D.

context *free-monoidal-category*
begin

lemma *rep-lunit*:
assumes *ide a*
shows *rep* l[*a*] = ‖**l**[*rep a*]‖
using *assms Lunit-in-lunit* [*of a rep a*] *rep-in-arr norm-memb-eq-rep* [*of* l[*a*]]
by *simp*

lemma *rep-runit*:
assumes *ide a*
shows *rep* r[*a*] = ‖**r**[*rep a*]‖
using *assms Runit-in-runit* [*of a rep a*] *rep-in-arr norm-memb-eq-rep* [*of* r[*a*]]
by *simp*

lemma *rep-assoc*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *rep* a[*a*, *b*, *c*] = ‖**a**[*rep a*, *rep b*, *rep c*]‖
using *assms Assoc-in-assoc* [*of a b c rep a rep b rep c*] *rep-in-arr*
norm-memb-eq-rep [*of* a[*a*, *b*, *c*]]
by *simp*

lemma *mkarr-Unity*:
shows *mkarr* $\mathcal{I}$ = $\mathcal{I}$
using *unity*$_{FMC}$*-def* **by** *simp*

The unitors and associator were given syntactic definitions in terms of corresponding terms, but these were only for the special case of identity arguments (*i.e.* the components

of the natural transformations). We need to show that *mkarr* gives the correct result for *all* terms.

lemma *mkarr-Lunit*:
assumes *Arr t*
shows *mkarr* $\mathbf{l}[t] = \mathfrak{l}\ (mkarr\ t)$
proof −
  have *mkarr* $\mathbf{l}[t] = mkarr\ (t \cdot \mathbf{l}[\|Dom\ t\|])$
    using *assms Arr-implies-Ide-Dom Ide-in-Hom Diagonalize-preserves-Ide*
        Diag-Diagonalize Par-Arr-norm
    by (*intro mkarr-eqI*) *simp-all*
  also have ... = *mkarr t* · *mkarr* $\mathbf{l}[\|Dom\ t\|]$
    using *assms Arr-implies-Ide-Dom Par-Arr-norm Ide-in-Hom* **by** *simp*
  also have ... = *mkarr t* · $\mathfrak{l}[dom\ (mkarr\ t)]$
  proof −
    have *arr* $\mathfrak{l}[mkarr\ (Dom\ t)]$
      using *assms Arr-implies-Ide-Dom ide-mkarr-Ide* **by** *simp*
    moreover have $\mathbf{l}[\|Dom\ t\|] \in \mathfrak{l}[mkarr\ (Dom\ t)]$
      using *assms Arr-implies-Ide-Dom Lunit-in-lunit rep-mkarr*
          rep-in-arr [*of mkarr* (*Dom t*)]
      by *simp*
    ultimately show *?thesis*
      using *assms mkarr-memb*(*2*) **by** *simp*
  qed
  also have ... = $\mathfrak{l}\ (mkarr\ t)$
    using *assms Arr-implies-Ide-Dom ide-mkarr-Ide lunit-agreement* **by** *simp*
  finally show *?thesis* **by** *blast*
qed

lemma *mkarr-Lunit′*:
assumes *Arr t*
shows *mkarr* $\mathbf{l}^{-1}[t] = \mathfrak{l}'\ (mkarr\ t)$
proof −
  have *mkarr* $\mathbf{l}^{-1}[t] = mkarr\ (\mathbf{l}^{-1}[\|Cod\ t\|] \cdot t)$
    using *assms Arr-implies-Ide-Cod Ide-in-Hom Diagonalize-preserves-Ide*
        Diag-Diagonalize Par-Arr-norm
    by (*intro mkarr-eqI*) *simp-all*
  also have ... = *mkarr* $\mathbf{l}^{-1}[\|Cod\ t\|]$ · *mkarr t*
    using *assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm* **by** *simp*
  also have ... = *mkarr* $(Inv\ \mathbf{l}[\|Cod\ t\|])$ · *mkarr t*
  proof −
    have *mkarr* $\mathbf{l}^{-1}[\|Cod\ t\|] = mkarr\ (Inv\ \mathbf{l}[\|Cod\ t\|])$
      using *assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm Inv-in-Hom*
          Ide-implies-Can norm-preserves-Can Diagonalize-Inv Diagonalize-preserves-Ide
      by (*intro mkarr-eqI*, *simp-all*)
    thus *?thesis* **by** *argo*
  qed
  also have ... = $\mathfrak{l}'\ (cod\ (mkarr\ t))$ · *mkarr t*
  proof −
    have *mkarr* $(Inv\ \mathbf{l}[\|Cod\ t\|])$ · *mkarr t* = *lunit′* $(cod\ (mkarr\ t))$ · *mkarr t*

```
    using assms Arr-implies-Ide-Cod rep-mkarr Par-Arr-norm inv-mkarr
          norm-preserves-Can Ide-implies-Can lunit-agreement ι'-ide-simp
          Can-implies-Arr arr-mkarr cod-mkarr ide-cod lunit_FMC-def
    by (metis (no-types, lifting) Can.simps(5))
  also have ... = ι' (cod (mkarr t)) · mkarr t
    using assms ι'-ide-simp arr-mkarr ide-cod by presburger
  finally show ?thesis by blast
qed
also have ... = ι' (mkarr t)
  using assms ι'.is-natural-2 [of mkarr t] by simp
finally show ?thesis by blast
qed

lemma mkarr-Runit:
assumes Arr t
shows mkarr r[t] = ϱ (mkarr t)
proof −
  have mkarr r[t] = mkarr (t · r[||Dom t||])
  proof −
    have ¬ Diag (Dom t ⊗ 𝓘) by (cases Dom t) simp-all
    thus ?thesis
      using assms Par-Arr-norm Arr-implies-Ide-Dom Ide-in-Hom Diag-Diagonalize
            Diagonalize-preserves-Ide
      by (intro mkarr-eqI) simp-all
  qed
  also have ... = mkarr t · mkarr r[||Dom t||]
    using assms Arr-implies-Ide-Dom Par-Arr-norm Ide-in-Hom by simp
  also have ... = mkarr t · r[dom (mkarr t)]
  proof −
    have arr r[mkarr (Dom t)]
      using assms Arr-implies-Ide-Dom ide-mkarr-Ide by simp
    moreover have r[||Dom t||] ∈ r[mkarr (Dom t)]
      using assms Arr-implies-Ide-Dom Runit-in-runit rep-mkarr
            rep-in-arr [of mkarr (Dom t)]
      by simp
    moreover have mkarr (Dom t) = mkarr ||Dom t||
      using assms mkarr-rep rep-mkarr arr-mkarr Ide-implies-Arr Arr-implies-Ide-Dom
      by metis
    ultimately show ?thesis
      using assms mkarr-memb(2) by simp
  qed
  also have ... = ϱ (mkarr t)
    using assms Arr-implies-Ide-Dom ide-mkarr-Ide runit-agreement by simp
  finally show ?thesis by blast
qed

lemma mkarr-Runit':
assumes Arr t
shows mkarr r⁻¹[t] = ϱ' (mkarr t)
```

```
proof −
  have mkarr r⁻¹[t] = mkarr (r⁻¹[||Cod t||] · t)
  proof −
    have ¬ Diag (Cod t ⊗ 𝓘) by (cases Cod t) simp-all
    thus ?thesis
      using assms Par-Arr-norm Arr-implies-Ide-Cod Ide-in-Hom
            Diagonalize-preserves-Ide Diag-Diagonalize
      by (intro mkarr-eqI) simp-all
  qed
  also have ... = mkarr r⁻¹[||Cod t||] · mkarr t
    using assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm by simp
  also have ... = mkarr (Inv r[||Cod t||]) · mkarr t
  proof −
    have mkarr (Runit' (norm (Cod t))) = mkarr (Inv (Runit (norm (Cod t))))
      using assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm Inv-in-Hom
            Ide-implies-Can norm-preserves-Can Diagonalize-Inv Diagonalize-preserves-Ide
      by (intro mkarr-eqI) simp-all
    thus ?thesis by argo
  qed
  also have ... = ϱ' (cod (mkarr t)) · mkarr t
  proof −
    have mkarr (Inv r[||Cod t||]) · mkarr t = runit' (cod (mkarr t)) · mkarr t
      using assms Arr-implies-Ide-Cod rep-mkarr inv-mkarr norm-preserves-Can
            Ide-implies-Can runit-agreement Can-implies-Arr arr-mkarr cod-mkarr
            ide-cod runit_FMC-def
      by (metis (no-types, lifting) Can.simps(7))
    also have ... = ϱ' (cod (mkarr t)) · mkarr t
    proof −
      have runit' (cod (mkarr t)) = ϱ' (cod (mkarr t))
        using assms ϱ'-ide-simp arr-mkarr ide-cod by blast
      thus ?thesis by argo
    qed
    finally show ?thesis by blast
  qed
  also have ... = ϱ' (mkarr t)
    using assms ϱ'.is-natural-2 [of mkarr t] by simp
  finally show ?thesis by blast
qed

lemma mkarr-Assoc:
assumes Arr t and Arr u and Arr v
shows mkarr a[t, u, v] = α (mkarr t, mkarr u, mkarr v)
proof −
  have mkarr a[t, u, v] = mkarr ((t ⊗ u ⊗ v) · a[||Dom t||, ||Dom u||, ||Dom v||])
    using assms Arr-implies-Ide-Dom Arr-implies-Ide-Cod Ide-in-Hom
          Diag-Diagonalize Diagonalize-preserves-Ide TensorDiag-preserves-Ide
          TensorDiag-preserves-Diag TensorDiag-assoc Par-Arr-norm
    by (intro mkarr-eqI, simp-all)
  also have ... = α (mkarr t, mkarr u, mkarr v)
```

using *assms Arr-implies-Ide-Dom rep-mkarr Ide-in-Hom* $assoc_{FMC}$*-def*
Par-Arr-norm [*of Dom t*] *Par-Arr-norm* [*of Dom u*] *Par-Arr-norm* [*of Dom v*]
α*-simp*
by *simp*
finally show *?thesis* **by** *blast*
qed

lemma *mkarr-Assoc'*:
assumes *Arr t* **and** *Arr u* **and** *Arr v*
shows *mkarr* $\mathbf{a}^{-1}[t, u, v] = \alpha'$ (*mkarr t*, *mkarr u*, *mkarr v*)
proof −
have *mkarr* $\mathbf{a}^{-1}[t, u, v] =$ *mkarr* ($\mathbf{a}^{-1}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|] \cdot (t \otimes u \otimes v)$)
using *assms Par-Arr-norm Arr-implies-Ide-Cod Ide-in-Hom Diag-Diagonalize*
TensorDiag-preserves-Diag CompDiag-Cod-Diag [*of* $\lfloor t \rfloor$ $\lfloor\otimes\rfloor$ $\lfloor u \rfloor$ $\lfloor\otimes\rfloor$ $\lfloor v \rfloor$]
by (*intro mkarr-eqI*, *simp-all*)
also have ... = *mkarr* $\mathbf{a}^{-1}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|]$ · *mkarr* ($t \otimes u \otimes v$)
using *assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm* **by** *simp*
also have ... = *mkarr* (*Inv* $\mathbf{a}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|]$) · *mkarr* ($t \otimes u \otimes v$)
proof −
have *mkarr* $\mathbf{a}^{-1}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|] =$
mkarr (*Inv* $\mathbf{a}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|]$)
using *assms Arr-implies-Ide-Cod Ide-in-Hom Par-Arr-norm Inv-in-Hom Ide-implies-Can*
norm-preserves-Can Diagonalize-Inv Diagonalize-preserves-Ide
by (*intro mkarr-eqI*, *simp-all*)
thus *?thesis* **by** *argo*
qed
also have ... = *inv* (*mkarr* $\mathbf{a}[\|Cod\ t\|, \|Cod\ u\|, \|Cod\ v\|]$) · *mkarr* ($t \otimes u \otimes v$)
using *assms Arr-implies-Ide-Cod Ide-implies-Can norm-preserves-Can* **by** *simp*
also have ... = α' (*mkarr t*, *mkarr u*, *mkarr v*)
proof −
have *mkarr* ($\mathbf{a}^{-1}[Inv\ \|Cod\ t\|, Inv\ \|Cod\ u\|, Inv\ \|Cod\ v\|] \cdot (Cod\ t \otimes Cod\ u \otimes Cod\ v)$)
= *mkarr* $\mathbf{a}^{-1}[Inv\ \|Cod\ t\|, Inv\ \|Cod\ u\|, Inv\ \|Cod\ v\|]$
using *assms Arr-implies-Ide-Cod Inv-in-Hom norm-preserves-Can Diagonalize-Inv*
Ide-implies-Can Diag-Diagonalize Ide-in-Hom Diagonalize-preserves-Ide
Par-Arr-norm TensorDiag-preserves-Diag
CompDiag-Cod-Diag [*of* $\lfloor Cod\ t \rfloor$ $\lfloor\otimes\rfloor$ $\lfloor Cod\ u \rfloor$ $\lfloor\otimes\rfloor$ $\lfloor Cod\ v \rfloor$]
by (*intro mkarr-eqI*) *simp-all*
thus *?thesis*
using *assms Arr-implies-Ide-Cod rep-mkarr* $assoc_{FMC}$*-def* α'*.map-simp* **by** *simp*
qed
finally show *?thesis* **by** *blast*
qed

Next, we define the "inclusion of generators" functor from C to $\mathcal{F}C$.

definition *inclusion-of-generators*
where *inclusion-of-generators* $\equiv \lambda f.$ *if C.arr f then mkarr* $\langle f \rangle$ *else null*

lemma *inclusion-is-functor*:
shows *functor C comp inclusion-of-generators*

```
    unfolding inclusion-of-generators-def
    apply unfold-locales
      apply auto[4]
    by (elim C.seqE, simp, intro mkarr-eqI, auto)

end
```

We now show that, given a functor V from C to a a monoidal category D, the evaluation map that takes formal arrows of the monoidal language of C to arrows of D induces a strict monoidal functor from $\mathcal{F}C$ to D.

```
locale evaluation-functor =
  C: category C +
  D: monoidal-category D T_D α_D ι_D +
  evaluation-map C D T_D α_D ι_D V +
  FC: free-monoidal-category C
for C :: 'c comp      (infixr ·_C 55)
and D :: 'd comp      (infixr ·_D 55)
and T_D :: 'd * 'd ⇒ 'd
and α_D :: 'd * 'd * 'd ⇒ 'd
and ι_D :: 'd
and V :: 'c ⇒ 'd
begin

  notation eval       (⦇-⦈)

  definition map
  where map f ≡ if FC.arr f then ⦇FC.rep f⦈ else D.null
```

It follows from the coherence theorem that a formal arrow and its normal form always have the same evaluation.

```
  lemma eval-norm:
  assumes Arr t
  shows ⦇‖t‖⦈ = ⦇t⦈
    using assms FC.Par-Arr-norm FC.Diagonalize-norm coherence canonical-factorization
    by simp

  interpretation functor FC.comp D map
  proof
    fix f
    show ¬FC.arr f ⟹ map f = D.null using map-def by simp
    assume f: FC.arr f
    show D.arr (map f) using f map-def FC.arr-char by simp
    show D.dom (map f) = map (FC.dom f)
      using f map-def eval-norm FC.rep-dom Arr-implies-Ide-Dom by auto
    show D.cod (map f) = map (FC.cod f)
      using f map-def eval-norm FC.rep-cod Arr-implies-Ide-Cod by auto
    next
    fix f g
    assume fg: FC.seq g f
```

```
    show map (FC.comp g f) = D (map g) (map f)
      using fg map-def FC.rep-comp FC.rep-preserves-seq eval-norm by auto
  qed

  lemma is-functor:
  shows functor FC.comp D map ..

  interpretation FF: product-functor FC.comp FC.comp D D map map ..
  interpretation FoT: composite-functor FC.CC.comp FC.comp D FC.T_FMC map ..
  interpretation ToFF: composite-functor FC.CC.comp D.CC.comp D FF.map T_D ..

  interpretation strict-monoidal-functor
                   FC.comp FC.T_FMC FC.α FC.ι D T_D α_D ι_D map
  proof
    show map FC.ι = ι_D
      using FC.ι-def FC.lunit-agreement map-def FC.rep-lunit FC.Arr-rep [of I]
            eval-norm FC.lunit-agreement D.unitor-coincidence D.comp-cod-arr D.unit-in-hom
      by auto
    show ⋀f g. ⟦ FC.arr f; FC.arr g ⟧ ⟹
                 map (FC.tensor f g) = D.tensor (map f) (map g)
      using map-def FC.rep-tensor FC.Arr-rep eval-norm by simp
    show ⋀a b c. ⟦ FC.ide a; FC.ide b; FC.ide c ⟧ ⟹
                   map (FC.assoc a b c) = D.assoc (map a) (map b) (map c)
      using map-def FC.assoc_FMC-def FC.rep-mkarr eval-norm by auto
  qed

  lemma is-strict-monoidal-functor:
  shows strict-monoidal-functor FC.comp FC.T_FMC FC.α FC.ι D T_D α_D ι_D map
    ..

end

sublocale evaluation-functor ⊆ strict-monoidal-functor
                                 FC.comp FC.T_FMC FC.α_FMC FC.ι_FMC D T_D α_D ι_D map
  using is-strict-monoidal-functor by auto
```

The final step in proving freeness is to show that the evaluation functor is the *unique* strict monoidal extension of the functor V to $\mathcal{F}C$. This is done by induction, exploiting the syntactic construction of $\mathcal{F}C$.

To ease the statement and proof of the result, we define a locale that expresses that F is a strict monoidal extension to monoidal category C, of a functor V from C_0 to a monoidal category D, along a functor I from C_0 to C.

```
locale strict-monoidal-extension =
  C₀: category C₀ +
  C: monoidal-category C T_C α_C ι_C +
  D: monoidal-category D T_D α_D ι_D +
  I: functor C₀ C I +
  V: functor C₀ D V +
```

strict-monoidal-functor $C\ T_C\ \alpha_C\ \iota_C\ D\ T_D\ \alpha_D\ \iota_D\ F$
for $C_0 :: 'c_0\ comp$
and $C :: 'c\ comp$ (**infixr** $\cdot_C$ 55)
and $T_C :: 'c * 'c \Rightarrow 'c$
and $\alpha_C :: 'c * 'c * 'c \Rightarrow 'c$
and $\iota_C :: 'c$
and $D :: 'd\ comp$ (**infixr** $\cdot_D$ 55)
and $T_D :: 'd * 'd \Rightarrow 'd$
and $\alpha_D :: 'd * 'd * 'd \Rightarrow 'd$
and $\iota_D :: 'd$
and $I :: 'c_0 \Rightarrow 'c$
and $V :: 'c_0 \Rightarrow 'd$
and $F :: 'c \Rightarrow 'd +$
assumes *is-extension*: $\forall f.\ C_0.arr\ f \longrightarrow F\ (I\ f) = V\ f$

sublocale *evaluation-functor* $\subseteq$
strict-monoidal-extension $C\ \mathcal{F}C.comp\ \mathcal{F}C.T_{FMC}\ \mathcal{F}C.\alpha\ \mathcal{F}C.\iota\ D\ T_D\ \alpha_D\ \iota_D$
$\mathcal{F}C.inclusion\text{-}of\text{-}generators\ V\ map$
proof −
interpret *inclusion*: *functor* $C\ \mathcal{F}C.comp\ \mathcal{F}C.inclusion\text{-}of\text{-}generators$
using $\mathcal{F}C.inclusion\text{-}is\text{-}functor$ **by** *auto*
show *strict-monoidal-extension* $C\ \mathcal{F}C.comp\ \mathcal{F}C.T_{FMC}\ \mathcal{F}C.\alpha\ \mathcal{F}C.\iota\ D\ T_D\ \alpha_D\ \iota_D$
$\mathcal{F}C.inclusion\text{-}of\text{-}generators\ V\ map$
apply *unfold-locales*
using *map-def* $\mathcal{F}C.rep\text{-}mkarr$ *eval-norm* $\mathcal{F}C.inclusion\text{-}of\text{-}generators\text{-}def$ **by** *simp*
qed

A special case of interest is a strict monoidal extension to $\mathcal{F}C$, of a functor V from a category C to a monoidal category D, along the inclusion of generators from C to $\mathcal{F}C$. The evaluation functor induced by V is such an extension.

locale *strict-monoidal-extension-to-free-monoidal-category* =
C: *category* C +
monoidal-language C +
$\mathcal{F}C$: *free-monoidal-category* C +
strict-monoidal-extension $C\ \mathcal{F}C.comp\ \mathcal{F}C.T_{FMC}\ \mathcal{F}C.\alpha\ \mathcal{F}C.\iota\ D\ T_D\ \alpha_D\ \iota_D$
$\mathcal{F}C.inclusion\text{-}of\text{-}generators\ V\ F$
for $C :: 'c\ comp$ (**infixr** $\cdot_C$ 55)
and $D :: 'd\ comp$ (**infixr** $\cdot_D$ 55)
and $T_D :: 'd * 'd \Rightarrow 'd$
and $\alpha_D :: 'd * 'd * 'd \Rightarrow 'd$
and $\iota_D :: 'd$
and $V :: 'c \Rightarrow 'd$
and $F :: 'c\ free\text{-}monoidal\text{-}category.arr \Rightarrow 'd$
begin

lemma *strictly-preserves-everything*:
shows $C.arr\ f \Longrightarrow F\ (\mathcal{F}C.mkarr\ \langle f \rangle) = V\ f$
and $F\ (\mathcal{F}C.mkarr\ \boldsymbol{\mathcal{I}}) = \mathcal{I}_D$
and $[\![\ Arr\ t;\ Arr\ u\ ]\!] \Longrightarrow F\ (\mathcal{F}C.mkarr\ (t \otimes u)) = F\ (\mathcal{F}C.mkarr\ t) \otimes_D F\ (\mathcal{F}C.mkarr\ u)$

and $[\![\mathit{Arr}\ t;\ \mathit{Arr}\ u;\ \mathit{Dom}\ t = \mathit{Cod}\ u]\!] \Longrightarrow$
$F\ (\mathcal{FC}.\mathit{mkarr}\ (t \cdot u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \cdot_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
and $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{l}[t]) = D.\mathfrak{l}\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
and $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{l}^{-1}[t]) = D.\mathfrak{l}'.\mathit{map}\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
and $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{r}[t]) = D.\varrho\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
and $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{r}^{-1}[t]) = D.\varrho'.\mathit{map}\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
and $[\![\mathit{Arr}\ t;\ \mathit{Arr}\ u;\ \mathit{Arr}\ v]\!] \Longrightarrow$
$F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{a}[t,\ u,\ v]) = \alpha_D\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t),\ F\ (\mathcal{FC}.\mathit{mkarr}\ u),\ F\ (\mathcal{FC}.\mathit{mkarr}\ v))$
and $[\![\mathit{Arr}\ t;\ \mathit{Arr}\ u;\ \mathit{Arr}\ v]\!] \Longrightarrow$
$F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{a}^{-1}[t,\ u,\ v])$
$= D.\alpha'\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t),\ F\ (\mathcal{FC}.\mathit{mkarr}\ u),\ F\ (\mathcal{FC}.\mathit{mkarr}\ v))$
proof −
show $C.\mathit{arr}\ f \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \langle f \rangle) = V\ f$
using *is-extension* $\mathcal{FC}.\mathit{inclusion\text{-}of\text{-}generators\text{-}def}$ **by** *simp*
show $F\ (\mathcal{FC}.\mathit{mkarr}\ \boldsymbol{\mathcal{I}}) = \mathcal{I}_D$
using $\mathcal{FC}.\mathit{mkarr\text{-}Unity}\ \mathcal{FC}.\iota\text{-}\mathit{def}$ *strictly-preserves-unity* $\mathcal{FC}.\mathcal{I}\text{-}\mathit{agreement}$ **by** *auto*
show *tensor-case*:
$\bigwedge t\ u.[\![\mathit{Arr}\ t;\ \mathit{Arr}\ u]\!] \Longrightarrow$
$F\ (\mathcal{FC}.\mathit{mkarr}\ (t \otimes u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \otimes_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
proof −
fix $t\ u$
assume t: $\mathit{Arr}\ t$ **and** u: $\mathit{Arr}\ u$
have $F\ (\mathcal{FC}.\mathit{mkarr}\ (t \otimes u)) = F\ (\mathcal{FC}.\mathit{tensor}\ (\mathcal{FC}.\mathit{mkarr}\ t)\ (\mathcal{FC}.\mathit{mkarr}\ u))$
using $t\ u\ \mathcal{FC}.\mathit{tensor\text{-}mkarr}\ \mathcal{FC}.\mathit{arr\text{-}mkarr}$ **by** *simp*
also have $\ldots = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \otimes_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
using $t\ u\ \mathcal{FC}.\mathit{arr\text{-}mkarr}$ *strictly-preserves-tensor* **by** *blast*
finally show $F\ (\mathcal{FC}.\mathit{mkarr}\ (t \otimes u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \otimes_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
by *fast*
qed
show $[\![\mathit{Arr}\ t;\ \mathit{Arr}\ u;\ \mathit{Dom}\ t = \mathit{Cod}\ u]\!] \Longrightarrow$
$F\ (\mathcal{FC}.\mathit{mkarr}\ (t \cdot u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \cdot_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
proof −
fix $t\ u$
assume t: $\mathit{Arr}\ t$ **and** u: $\mathit{Arr}\ u$ **and** tu: $\mathit{Dom}\ t = \mathit{Cod}\ u$
show $F\ (\mathcal{FC}.\mathit{mkarr}\ (t \cdot u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \cdot_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
proof −
have $F\ (\mathcal{FC}.\mathit{mkarr}\ (t \cdot u)) = F\ (\mathcal{FC}.\mathit{mkarr}\ t \cdot \mathcal{FC}.\mathit{mkarr}\ u)$
using $t\ u\ tu\ \mathcal{FC}.\mathit{comp\text{-}mkarr}$ **by** *simp*
also have $\ldots = F\ (\mathcal{FC}.\mathit{mkarr}\ t) \cdot_D F\ (\mathcal{FC}.\mathit{mkarr}\ u)$
using $t\ u\ tu\ \mathcal{FC}.\mathit{arr\text{-}mkarr}$ **by** *fastforce*
finally show *?thesis* **by** *blast*
qed
qed
show $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{l}[t]) = D.\mathfrak{l}\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
using $\mathcal{FC}.\mathit{mkarr\text{-}Lunit}\ \mathit{Arr\text{-}implies\text{-}Ide\text{-}Dom}\ \mathcal{FC}.\mathit{ide\text{-}mkarr\text{-}Ide}$ *strictly-preserves-lunit*
by *simp*
show $\mathit{Arr}\ t \Longrightarrow F\ (\mathcal{FC}.\mathit{mkarr}\ \mathbf{r}[t]) = D.\varrho\ (F\ (\mathcal{FC}.\mathit{mkarr}\ t))$
using $\mathcal{FC}.\mathit{mkarr\text{-}Runit}\ \mathit{Arr\text{-}implies\text{-}Ide\text{-}Dom}\ \mathcal{FC}.\mathit{ide\text{-}mkarr\text{-}Ide}$ *strictly-preserves-runit*
by *simp*

show ⟦ *Arr t*; *Arr u*; *Arr v* ⟧ ⟹
  F (*FC.mkarr* **a**[*t*, *u*, *v*])
    = *α*$_D$ (*F* (*FC.mkarr t*), *F* (*FC.mkarr u*), *F* (*FC.mkarr v*))
  using *FC.mkarr-Assoc strictly-preserves-assoc FC.ide-mkarr-Ide tensor-case*
  by *simp*
show *Arr t* ⟹ *F* (*FC.mkarr* **l**$^{-1}$[*t*]) = *D.𝔩′.map* (*F* (*FC.mkarr t*))
proof −
  assume *t*: *Arr t*
  have *F* (*FC.mkarr* **l**$^{-1}$[*t*]) = *F* (*FC.lunit′* (*FC.mkarr* (*Cod t*))) ⋅$_D$ *F* (*FC.mkarr t*)
    using *t FC.mkarr-Lunit′ Arr-implies-Ide-Cod FC.ide-mkarr-Ide FC.𝔩′.map-simp*
        FC.comp-cod-arr
    by *simp*
  also have ... = *D.lunit′* (*D.cod* (*F* (*FC.mkarr t*))) ⋅$_D$ *F* (*FC.mkarr t*)
    using *t Arr-implies-Ide-Cod FC.ide-mkarr-Ide strictly-preserves-lunit*
        preserves-inv
    by *simp*
  also have ... = *D.𝔩′.map* (*F* (*FC.mkarr t*))
    using *t D.𝔩′.map-simp D.comp-cod-arr* **by** *simp*
  finally show *?thesis* **by** *blast*
qed
show *Arr t* ⟹ *F* (*FC.mkarr* **r**$^{-1}$[*t*]) = *D.ϱ′.map* (*F* (*FC.mkarr t*))
proof −
  assume *t*: *Arr t*
  have *F* (*FC.mkarr* **r**$^{-1}$[*t*]) = *F* (*FC.runit′* (*FC.mkarr* (*Cod t*))) ⋅$_D$ *F* (*FC.mkarr t*)
    using *t FC.mkarr-Runit′ Arr-implies-Ide-Cod FC.ide-mkarr-Ide FC.ϱ′.map-simp*
        FC.comp-cod-arr
    by *simp*
  also have ... = *D.runit′* (*D.cod* (*F* (*FC.mkarr t*))) ⋅$_D$ *F* (*FC.mkarr t*)
    using *t Arr-implies-Ide-Cod FC.ide-mkarr-Ide strictly-preserves-runit*
        preserves-inv
    by *simp*
  also have ... = *D.ϱ′.map* (*F* (*FC.mkarr t*))
    using *t D.ϱ′.map-simp D.comp-cod-arr* **by** *simp*
  finally show *?thesis* **by** *blast*
qed
show ⟦ *Arr t*; *Arr u*; *Arr v* ⟧ ⟹
  F (*FC.mkarr* **a**$^{-1}$[*t*, *u*, *v*])
    = *D.α′.map* (*F* (*FC.mkarr t*), *F* (*FC.mkarr u*), *F* (*FC.mkarr v*))
proof −
  assume *t*: *Arr t* **and** *u*: *Arr u* **and** *v*: *Arr v*
  have *F* (*FC.mkarr* **a**$^{-1}$[*t*, *u*, *v*]) =
      F (*FC.assoc′* (*FC.mkarr* (*Cod t*)) (*FC.mkarr* (*Cod u*)) (*FC.mkarr* (*Cod v*))) ⋅$_D$
        (*F* (*FC.mkarr t*) ⊗$_D$ *F* (*FC.mkarr u*) ⊗$_D$ *F* (*FC.mkarr v*))
    using *t u v FC.mkarr-Assoc′ Arr-implies-Ide-Cod FC.ide-mkarr-Ide FC.α′.map-simp*
        tensor-case FC.iso-assoc
    by *simp*
  also have ... = *D.assoc′* (*D.cod* (*F* (*FC.mkarr t*))) (*D.cod* (*F* (*FC.mkarr u*)))
                      (*D.cod* (*F* (*FC.mkarr v*))) ⋅$_D$
                      (*F* (*FC.mkarr t*) ⊗$_D$ *F* (*FC.mkarr u*) ⊗$_D$ *F* (*FC.mkarr v*))

```
        using t u v FC.ide-mkarr-Ide Arr-implies-Ide-Cod preserves-inv FC.iso-assoc
              strictly-preserves-assoc
                [of FC.mkarr (Cod t) FC.mkarr (Cod u) FC.mkarr (Cod v)]
        by simp
      also have ... = D.α'.map (F (FC.mkarr t), F (FC.mkarr u), F (FC.mkarr v))
        using t u v D.α'.map-simp by simp
      finally show ?thesis by blast
    qed
  qed

end

sublocale evaluation-functor ⊆ strict-monoidal-extension-to-free-monoidal-category
                                 C D T_D α_D ι_D V map
  ..

context free-monoidal-category
begin
```

The evaluation functor induced by V is the unique strict monoidal extension of V to $\mathcal{F}C$.

```
  theorem is-free:
  assumes strict-monoidal-extension-to-free-monoidal-category C D T_D α_D ι_D V F
  shows F = evaluation-functor.map C D T_D α_D ι_D V
  proof −
    interpret F: strict-monoidal-extension-to-free-monoidal-category C D T_D α_D ι_D V F
      using assms by auto
    interpret E: evaluation-functor C D T_D α_D ι_D V ..
    have Ide-case: ⋀a. Ide a ⟹ F (mkarr a) = E.map (mkarr a)
    proof −
      fix a
      show Ide a ⟹ F (mkarr a) = E.map (mkarr a)
        using E.strictly-preserves-everything F.strictly-preserves-everything Ide-implies-Arr
        by (induct a) auto
    qed
    show ?thesis
    proof
      fix f
      have ¬arr f ⟹ F f = E.map f
        using E.is-extensional F.is-extensional by simp
      moreover have arr f ⟹ F f = E.map f
      proof −
        assume f: arr f
        have Arr (rep f) ∧ f = mkarr (rep f) using f mkarr-rep by simp
        moreover have ⋀t. Arr t ⟹ F (mkarr t) = E.map (mkarr t)
        proof −
          fix t
          show Arr t ⟹ F (mkarr t) = E.map (mkarr t)
            using Ide-case E.strictly-preserves-everything F.strictly-preserves-everything
```

```
            Arr-implies-Ide-Dom Arr-implies-Ide-Cod
        by (induct t) auto
      qed
      ultimately show F f = E.map f by metis
    qed
    ultimately show F f = E.map f by blast
  qed
qed

end
```

4.3 Strict Subcategory

```
context free-monoidal-category
begin
```

In this section we show that $\mathcal{F}C$ is monoidally equivalent to its full subcategory $\mathcal{F}_S C$ whose objects are the equivalence classes of diagonal identity terms, and that this subcategory is the free strict monoidal category generated by C.

```
interpretation \mathcal{F}_S C: full-subcategory comp ‹λf. ide f ∧ Diag (DOM f)›
  by (unfold-locales) auto
```

The mapping defined on equivalence classes by diagonalizing their representatives is a functor from the free monoidal category to the subcategory $\mathcal{F}_S C$.

```
definition D
where D ≡ λf. if arr f then mkarr ⌊rep f⌋ else \mathcal{F}_S C.null
```

The arrows of $\mathcal{F}_S C$ are those equivalence classes whose canonical representative term has diagonal formal domain and codomain.

```
lemma strict-arr-char:
shows \mathcal{F}_S C.arr f ⟷ arr f ∧ Diag (DOM f) ∧ Diag (COD f)
proof
  show arr f ∧ Diag (DOM f) ∧ Diag (COD f) ⟹ \mathcal{F}_S C.arr f
    using \mathcal{F}_S C.arr-char_SbC DOM-dom DOM-cod by simp
  show \mathcal{F}_S C.arr f ⟹ arr f ∧ Diag (DOM f) ∧ Diag (COD f)
   using \mathcal{F}_S C.arr-char_SbC Arr-rep Arr-implies-Ide-Cod Ide-implies-Arr DOM-dom DOM-cod
    by force
qed
```

Alternatively, the arrows of $\mathcal{F}_S C$ are those equivalence classes that are preserved by diagonalization of representatives.

```
lemma strict-arr-char':
shows \mathcal{F}_S C.arr f ⟷ arr f ∧ D f = f
proof
  fix f
  assume f: \mathcal{F}_S C.arr f
  show arr f ∧ D f = f
  proof
```

```
    show arr f using f \<F>_S C.arr-char_SbC by blast
    show D f = f
      using f strict-arr-char mkarr-Diagonalize-rep D-def by simp
  qed
  next
  assume f: arr f ∧ D f = f
  show \<F>_S C.arr f
  proof −
    have arr f using f by simp
    moreover have Diag (DOM f)
    proof −
      have DOM f = DOM (mkarr ⌊rep f⌋) using f D-def by auto
      also have ... = Dom ‖⌊rep f⌋‖
        using f Arr-rep Diagonalize-in-Hom rep-mkarr by simp
      also have ... = Dom ⌊rep f⌋
        using f Arr-rep Diagonalize-in-Hom Par-Arr-norm [of ⌊rep f⌋] by force
      finally have DOM f = Dom ⌊rep f⌋ by blast
      thus ?thesis using f Arr-rep Diag-Diagonalize Dom-preserves-Diag by metis
    qed
    moreover have Diag (COD f)
    proof −
      have COD f = COD (mkarr ⌊rep f⌋) using f D-def by auto
      also have ... = Cod ‖⌊rep f⌋‖
        using f Arr-rep Diagonalize-in-Hom rep-mkarr by simp
      also have ... = Cod ⌊rep f⌋
        using f Arr-rep Diagonalize-in-Hom Par-Arr-norm [of ⌊rep f⌋] by force
      finally have COD f = Cod ⌊rep f⌋ by blast
      thus ?thesis using f Arr-rep Diag-Diagonalize Cod-preserves-Diag by metis
    qed
    ultimately show ?thesis using strict-arr-char by auto
  qed
qed

interpretation D: functor comp \<F>_S C.comp D
proof −
  have 1: ⋀f. arr f ⟹ \<F>_S C.arr (D f)
    unfolding strict-arr-char D-def
    using arr-mkarr Diagonalize-in-Hom Arr-rep rep-mkarr Par-Arr-norm
          Arr-implies-Ide-Dom Arr-implies-Ide-Cod Diag-Diagonalize
    by force
  show functor comp \<F>_S C.comp D
  proof
    show ⋀f. ¬ arr f ⟹ D f = \<F>_S C.null using D-def by simp
    show ⋀f. arr f ⟹ \<F>_S C.arr (D f) by fact
    show ⋀f. arr f ⟹ \<F>_S C.dom (D f) = D (dom f)
      using D-def Diagonalize-in-Hom \<F>_S C.dom-char_SbC \<F>_S C.arr-char_SbC
            rep-mkarr rep-dom Arr-implies-Ide-Dom Arr-implies-Ide-Cod
            Diagonalize-preserves-Ide ide-mkarr-Ide Diag-Diagonalize Dom-norm
      by simp
```

```
    show 2: ⋀f. arr f ⟹ F_S C.cod (D f) = D (cod f)
      using D-def Diagonalize-in-Hom F_S C.cod-char_SbC F_S C.arr-char_SbC
            rep-mkarr rep-cod Arr-implies-Ide-Dom Arr-implies-Ide-Cod
            Diagonalize-preserves-Ide ide-mkarr-Ide Diag-Diagonalize Dom-norm
      by simp
    fix f g
    assume fg: seq g f
    hence fg': arr f ∧ arr g ∧ dom g = cod f by blast
    show D (g · f) = F_S C.comp (D g) (D f)
    proof −
      have seq: F_S C.seq (mkarr ⌊rep g⌋) (mkarr ⌊rep f⌋)
      proof −
        have 3: F_S C.arr (mkarr ⌊rep g⌋) ∧ F_S C.arr (mkarr ⌊rep f⌋)
          using fg' 1 arr-char D-def by force
        moreover have F_S C.dom (mkarr ⌊rep g⌋) = F_S C.cod (mkarr ⌊rep f⌋)
          using fg' 2 3 F_S C.dom-char_SbC rep-in-Hom mkarr-in-hom D-def
                Dom-Diagonalize-rep Diag-implies-Arr Diag-Diagonalize(1) F_S C.arr-char_SbC
          by force
        ultimately show ?thesis using F_S C.seqI by auto
      qed
      have mkarr ⌊rep (g · f)⌋ = F_S C.comp (mkarr ⌊rep g⌋) (mkarr ⌊rep f⌋)
      proof −
        have Seq: Seq ⌊rep g⌋ ⌊rep f⌋
          using fg rep-preserves-seq Diagonalize-in-Hom by force
        hence 4: ⌊rep g⌋ · ⌊rep f⌋ ∈ Hom ⌊DOM f⌋ ⌊COD g⌋
          using fg' Seq Diagonalize-in-Hom by auto
        have F_S C.comp (mkarr ⌊rep g⌋) (mkarr ⌊rep f⌋) = mkarr ⌊rep g⌋ · mkarr ⌊rep f⌋
          using seq F_S C.comp-char F_S C.seq-char_SbC by meson
        also have ... = mkarr (⌊rep g⌋ · ⌊rep f⌋)
          using Seq comp-mkarr by fastforce
        also have ... = mkarr ⌊rep (g · f)⌋
        proof (intro mkarr-eqI)
          show Par (⌊rep g⌋ · ⌊rep f⌋) ⌊rep (g · f)⌋
            using fg 4 rep-in-Hom rep-preserves-seq rep-in-Hom Diagonalize-in-Hom
                  Par-Arr-norm
            apply (elim seqE, auto)
            by (simp-all add: rep-comp)
          show ⌊⌊rep g⌋ · ⌊rep f⌋⌋ = ⌊⌊rep (g · f)⌋⌋
            using fg rep-preserves-seq norm-in-Hom Diag-Diagonalize Diagonalize-Diag
            apply auto
            by (simp add: rep-comp)
        qed
        finally show ?thesis by blast
      qed
      thus ?thesis using fg D-def by auto
    qed
  qed
qed
```

lemma *diagonalize-is-functor*:
shows *functor comp* $\mathcal{F}_S C.comp\ D$ **..**

lemma *diagonalize-strict-arr*:
assumes $\mathcal{F}_S C.arr\ f$
shows $D\ f = f$
using *assms arr-char D-def strict-arr-char Arr-rep Arr-implies-Ide-Dom Ide-implies-Arr mkarr-Diagonalize-rep* [*of f*]
by *auto*

lemma *diagonalize-is-idempotent*:
shows $D\ o\ D = D$
unfolding *D-def*
using *D.is-extensional* $\mathcal{F}_S C.null\text{-}char$ *Arr-rep Diagonalize-in-Hom mkarr-Diagonalize-rep strict-arr-char rep-mkarr*
by *fastforce*

lemma *diagonalize-tensor*:
assumes *arr f* **and** *arr g*
shows $D\ (f \otimes g) = D\ (D\ f \otimes D\ g)$
unfolding *D-def*
using *assms strict-arr-char rep-in-Hom Diagonalize-in-Hom tensor-mkarr rep-tensor Diagonalize-in-Hom rep-mkarr Diagonalize-norm Diagonalize-Tensor*
by *force*

lemma *ide-diagonalize-can*:
assumes *can f*
shows *ide* $(D\ f)$
using *assms D-def Can-rep-can Ide-Diagonalize-Can ide-mkarr-Ide can-implies-arr*
by *simp*

We next show that the diagonalization functor and the inclusion of the full subcategory $\mathcal{F}_S C$ underlie an equivalence of categories. The arrows *mkarr* $(DOM\ a\downarrow)$, determined by reductions of canonical representatives, are the components of a natural isomorphism.

interpretation *S*: *full-inclusion-functor comp* ‹$\lambda f.\ ide\ f \wedge Diag\ (DOM\ f)$› **..**
interpretation *DoS*: *composite-functor* $\mathcal{F}_S C.comp$ *comp* $\mathcal{F}_S C.comp$ $\mathcal{F}_S C.map$ *D*
..
interpretation *SoD*: *composite-functor comp* $\mathcal{F}_S C.comp$ *comp D* $\mathcal{F}_S C.map$ **..**

interpretation ν: *transformation-by-components*
comp comp map SoD.map ‹$\lambda a.\ mkarr\ (DOM\ a\downarrow)$›
proof
fix *a*
assume *a*: *ide a*
show «*mkarr* $(DOM\ a\downarrow) : map\ a \rightarrow SoD.map\ a$»
proof −
have «*mkarr* $(DOM\ a\downarrow) : a \rightarrow mkarr\ \lfloor DOM\ a \rfloor$»
using *a Arr-implies-Ide-Dom red-in-Hom dom-char* [*of a*] **by** *auto*

```
    moreover have map a = a
      using a map-simp by simp
    moreover have SoD.map a = mkarr ⌊DOM a⌋
      using a D.preserves-ide \mathcal{F}_S C.ideD \mathcal{F}_S C.map-simp D-def Ide-Diagonalize-rep-ide
          Ide-in-Hom Diagonalize-in-Hom
      by force
    ultimately show ?thesis by simp
  qed
  next
  fix f
  assume f: arr f
  show mkarr (DOM (cod f)↓) · map f = SoD.map f · mkarr (DOM (dom f)↓)
  proof −
    have SoD.map f · mkarr (DOM (dom f)↓) = mkarr ⌊rep f⌋ · mkarr (DOM f↓)
      using f DOM-dom D.preserves-arr \mathcal{F}_S C.map-simp D-def by simp
    also have ... = mkarr (⌊rep f⌋ · DOM f↓)
      using f Diagonalize-in-Hom red-in-Hom comp-mkarr Arr-implies-Ide-Dom
      by simp
    also have ... = mkarr (COD f↓ · rep f)
    proof (intro mkarr-eqI)
      show Par (⌊rep f⌋ · DOM f↓) (COD f↓ · rep f)
        using f Diagonalize-in-Hom red-in-Hom Arr-implies-Ide-Dom Arr-implies-Ide-Cod
        by simp
      show ⌊⌊rep f⌋ · DOM f↓⌋ = ⌊COD f↓ · rep f⌋
      proof −
        have ⌊⌊rep f⌋ · DOM f↓⌋ = ⌊rep f⌋ ⌊·⌋ ⌊DOM f↓⌋
          using f by simp
        also have ... = ⌊rep f⌋
          using f Arr-implies-Ide-Dom Can-red Ide-Diagonalize-Can [of DOM f↓]
              Diag-Diagonalize CompDiag-Diag-Ide
          by force
        also have ... = ⌊COD f↓⌋ ⌊·⌋ ⌊rep f⌋
          using f Arr-implies-Ide-Cod Can-red Ide-Diagonalize-Can [of COD f↓]
              Diag-Diagonalize CompDiag-Diag-Ide
          by force
        also have ... = ⌊COD f↓ · rep f⌋
          by simp
        finally show ?thesis by blast
      qed
    qed
    also have ... = mkarr (COD f↓) · mkarr (rep f)
      using f comp-mkarr rep-in-Hom red-in-Hom Arr-implies-Ide-Cod by blast
    also have ... = mkarr (DOM (cod f)↓) · map f
      using f DOM-cod by simp
    finally show ?thesis by blast
  qed
qed

interpretation ν: natural-isomorphism comp comp map SoD.map ν.map
```

apply *unfold-locales*
using *ν.map-simp-ide rep-in-Hom Arr-implies-Ide-Dom Can-red can-mkarr-Can iso-can*
by *simp*

The restriction of the diagonalization functor to the subcategory $\mathcal{F}_S C$ is the identity.

lemma *DoS-eq-*$\mathcal{F}_S C$:
shows *DoS.map* = $\mathcal{F}_S C$*.map*
proof
fix *f*
have ¬ $\mathcal{F}_S C$*.arr f* ⟹ *DoS.map f* = $\mathcal{F}_S C$*.map f*
using *DoS.is-extensional* $\mathcal{F}_S C$*.map-def* **by** *simp*
moreover have $\mathcal{F}_S C$*.arr f* ⟹ *DoS.map f* = $\mathcal{F}_S C$*.map f*
using $\mathcal{F}_S C$*.map-simp strict-arr-char Diagonalize-Diag D-def mkarr-Diagonalize-rep*
by *simp*
ultimately show *DoS.map f* = $\mathcal{F}_S C$*.map f* **by** *blast*
qed

interpretation *μ*: *transformation-by-components*
$\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.comp DoS.map* $\mathcal{F}_S C$*.map* ‹λ*a*. *a*›
using $\mathcal{F}_S C$*.ideD* $\mathcal{F}_S C$*.map-simp DoS-eq-*$\mathcal{F}_S C$ $\mathcal{F}_S C$*.map-simp* $\mathcal{F}_S C$*.comp-cod-arr* $\mathcal{F}_S C$*.comp-arr-dom*
apply *unfold-locales*
by (*intro* $\mathcal{F}_S C$*.in-homI*) *auto*

interpretation *μ*: *natural-isomorphism* $\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.comp DoS.map* $\mathcal{F}_S C$*.map μ.map*
apply *unfold-locales* **using** *μ.map-simp-ide* $\mathcal{F}_S C$*.ide-is-iso* **by** *simp*

interpretation *equivalence-of-categories* $\mathcal{F}_S C$*.comp comp D* $\mathcal{F}_S C$*.map ν.map μ.map* **..**

We defined the natural isomorphisms μ and ν by giving their components (*i.e.* their values at objects). However, it is helpful in exporting these facts to have simple characterizations of their values for all arrows.

definition *μ*
where *μ* ≡ λ*f*. *if* $\mathcal{F}_S C$*.arr f then f else* $\mathcal{F}_S C$*.null*

definition *ν*
where *ν* ≡ λ*f*. *if arr f then mkarr* (*COD f*↓) · *f else null*

lemma *μ-char*:
shows *μ.map* = *μ*
proof (*intro NaturalTransformation.eqI*)
show *natural-transformation* $\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.comp DoS.map* $\mathcal{F}_S C$*.map μ.map* **..**
have *natural-transformation* $\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.map* $\mathcal{F}_S C$*.map* $\mathcal{F}_S C$*.map*
using *DoS.as-nat-trans.natural-transformation-axioms DoS-eq-*$\mathcal{F}_S C$ **by** *simp*
moreover have $\mathcal{F}_S C$*.map* = *μ* **unfolding** *μ-def* **using** $\mathcal{F}_S C$*.map-def* **by** *blast*
ultimately show *natural-transformation* $\mathcal{F}_S C$*.comp* $\mathcal{F}_S C$*.comp DoS.map* $\mathcal{F}_S C$*.map μ*
using $\mathcal{F}_S C$*.as-nat-trans.natural-transformation-axioms DoS-eq-*$\mathcal{F}_S C$ **by** *simp*
show ⋀*a*. $\mathcal{F}_S C$*.ide a* ⟹ *μ.map a* = *μ a*
using *μ.map-simp-ide* $\mathcal{F}_S C$*.ideD μ-def* **by** *simp*
qed

lemma *ν-char*:
shows $\nu.map = \nu$
unfolding *ν.map-def ν-def* **using** *map-simp DOM-cod* **by** *fastforce*

lemma *is-equivalent-to-strict-subcategory*:
shows *equivalence-of-categories* $\mathcal{F}_S C.comp$ *comp D* $\mathcal{F}_S C.map$ ν μ
proof −
have *equivalence-of-categories* $\mathcal{F}_S C.comp$ *comp D* $\mathcal{F}_S C.map$ *ν.map μ.map* **..**
thus *equivalence-of-categories* $\mathcal{F}_S C.comp$ *comp D* $\mathcal{F}_S C.map$ ν μ
using *ν-char μ-char* **by** *simp*
qed

The inclusion of generators functor from C to $\mathcal{F}C$ corestricts to a functor from C to $\mathcal{F}_S C$.

interpretation *I*: *functor C comp inclusion-of-generators*
using *inclusion-is-functor* **by** *auto*
interpretation *DoI*: *composite-functor C comp* $\mathcal{F}_S C.comp$ *inclusion-of-generators D* **..**

lemma *DoI-eq-I*:
shows *DoI.map* = *inclusion-of-generators*
proof
fix *f*
have ¬ *C.arr f* ⟹ *DoI.map f* = *inclusion-of-generators f*
using *DoI.is-extensional I.is-extensional* $\mathcal{F}_S C.null\text{-}char$ **by** *blast*
moreover have *C.arr f* ⟹ *DoI.map f* = *inclusion-of-generators f*
proof −
assume *f*: *C.arr f*
have *DoI.map f* = *D* (*inclusion-of-generators f*) **using** *f* **by** *simp*
also have ... = *inclusion-of-generators f*
proof −
have $\mathcal{F}_S C.arr$ (*inclusion-of-generators f*)
using *f arr-mkarr rep-mkarr Par-Arr-norm* [*of* ⟨*f*⟩] *strict-arr-char*
inclusion-of-generators-def
by *simp*
thus *?thesis* **using** *f strict-arr-char'* **by** *blast*
qed
finally show *DoI.map f* = *inclusion-of-generators f* **by** *blast*
qed
ultimately show *DoI.map f* = *inclusion-of-generators f* **by** *blast*
qed

end

Next, we show that the subcategory $\mathcal{F}_S C$ inherits monoidal structure from the ambient category $\mathcal{F}C$, and that this monoidal structure is strict.

locale *free-strict-monoidal-category* =
monoidal-language C +
$\mathcal{F}C$: *free-monoidal-category C* +

full-subcategory $\mathcal{FC}.comp\ \lambda f.\ \mathcal{FC}.ide\ f \wedge Diag\ (\mathcal{FC}.DOM\ f)$
for $C :: 'c\ comp$
begin

interpretation D: *functor* $\mathcal{FC}.comp\ comp\ \mathcal{FC}.D$
using $\mathcal{FC}.diagonalize\text{-}is\text{-}functor$ **by** *auto*

notation *comp* (**infixr** $\cdot_S$ *55*)

definition $tensor_S$ (**infixr** $\otimes_S$ *53*)
where $f \otimes_S g \equiv \mathcal{FC}.D\ (\mathcal{FC}.tensor\ f\ g)$

definition $assoc_S$ (a_S[-, -, -])
where $assoc_S\ a\ b\ c \equiv a \otimes_S b \otimes_S c$

lemma *tensor-char*:
assumes *arr f* **and** *arr g*
shows $f \otimes_S g = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor \otimes \rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor)$
unfolding $\mathcal{FC}.D\text{-}def\ tensor_S\text{-}def$
using *assms* $arr\text{-}char_{S \flat C}\ \mathcal{FC}.rep\text{-}tensor$ **by** *simp*

lemma *tensor-in-hom* [*simp*]:
assumes «$f : a \to b$» **and** «$g : c \to d$»
shows «$f \otimes_S g : a \otimes_S c \to b \otimes_S d$»
unfolding $tensor_S\text{-}def$
using *assms* $D.preserves\text{-}hom\ arr\text{-}char_{S \flat C}\ in\text{-}hom\text{-}char_{S \flat C}$
by (*metis* (*no-types*, *lifting*) $\mathcal{FC}.T\text{-}simp\ \mathcal{FC}.tensor\text{-}in\text{-}hom\ in\text{-}homE$)

lemma *arr-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *arr* $(f \otimes_S g)$
using *assms* *arr-iff-in-hom* [*of f*] *arr-iff-in-hom* [*of g*] *tensor-in-hom* **by** *blast*

lemma *dom-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *dom* $(f \otimes_S g) = dom\ f \otimes_S dom\ g$
using *assms* *arr-iff-in-hom* [*of f*] *arr-iff-in-hom* [*of g*] *tensor-in-hom* **by** *blast*

lemma *cod-tensor* [*simp*]:
assumes *arr f* **and** *arr g*
shows *cod* $(f \otimes_S g) = cod\ f \otimes_S cod\ g$
using *assms* *arr-iff-in-hom* [*of f*] *arr-iff-in-hom* [*of g*] *tensor-in-hom* **by** *blast*

lemma *tensor-preserves-ide*:
assumes *ide a* **and** *ide b*
shows *ide* $(a \otimes_S b)$
using *assms* $tensor_S\text{-}def\ D.preserves\text{-}ide\ \mathcal{FC}.tensor\text{-}preserves\text{-}ide\ ide\text{-}char_{S \flat C}$
by *fastforce*

lemma *tensor-tensor*:
assumes *arr f* **and** *arr g* **and** *arr h*
shows $(f \otimes_S g) \otimes_S h = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
and $f \otimes_S g \otimes_S h = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
proof −
  show $(f \otimes_S g) \otimes_S h = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
  proof −
    have $(f \otimes_S g) \otimes_S h = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ (f \otimes_S g) \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
      using *assms Diag-Diagonalize TensorDiag-preserves-Diag Diag-implies-Arr*
          $\mathcal{FC}.COD$-*mkarr* $\mathcal{FC}.DOM$-*mkarr* $\mathcal{FC}.strict$-*arr-char tensor-char*
      by *simp*
    also have
      $... = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ (\mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor))\rfloor\ \lfloor\otimes\rfloor$
                  $\lfloor \mathcal{FC}.rep\ h \rfloor)$
      using *assms* $arr\text{-}char_{SbC}$ *tensor-char* **by** *simp*
    also have $... = \mathcal{FC}.mkarr\ (\lfloor\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
      using *assms* $\mathcal{FC}.rep$-*mkarr TensorDiag-in-Hom Diag-Diagonalize*
          TensorDiag-preserves-Diag $arr\text{-}char_{SbC}$
      by *force*
    also have $... = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
     using *assms Diag-Diagonalize TensorDiag-preserves-Diag TensorDiag-assoc* $arr\text{-}char_{SbC}$
      by *force*
    finally show *?thesis* **by** *blast*
  qed
  show $f \otimes_S g \otimes_S h = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor)$
  proof −
    have $... = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor\lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor\rfloor)$
      using *assms Diag-Diagonalize TensorDiag-preserves-Diag* $arr\text{-}char_{SbC}$ **by** *force*
    also have $... = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor$
                  $(\lfloor \mathcal{FC}.rep\ (\mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ g \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ h \rfloor))\rfloor))$
     using *assms* $\mathcal{FC}.rep$-*mkarr TensorDiag-in-Hom Diag-Diagonalize* $arr\text{-}char_{SbC}$ **by** *force*
    also have $... = \mathcal{FC}.mkarr\ (\lfloor \mathcal{FC}.rep\ f \rfloor\ \lfloor\otimes\rfloor\ \lfloor \mathcal{FC}.rep\ (g \otimes_S h) \rfloor)$
      using *assms tensor-char* **by** *simp*
    also have $... = f \otimes_S g \otimes_S h$
      using *assms Diag-Diagonalize TensorDiag-preserves-Diag Diag-implies-Arr*
          $\mathcal{FC}.COD$-*mkarr* $\mathcal{FC}.DOM$-*mkarr* $\mathcal{FC}.strict$-*arr-char tensor-char*
      by *simp*
    finally show *?thesis* **by** *blast*
  qed
qed

lemma *tensor-assoc*:
assumes *arr f* **and** *arr g* **and** *arr h*
shows $(f \otimes_S g) \otimes_S h = f \otimes_S g \otimes_S h$
  using *assms tensor-tensor* **by** *presburger*

lemma *arr-unity*:
shows *arr* $\mathcal{I}$
  using $\mathcal{FC}.rep$-*unity* $\mathcal{FC}.Par$-*Arr-norm* $\mathcal{FC}.\mathcal{I}$-*agreement* $\mathcal{FC}.strict$-*arr-char* **by** *force*

lemma *tensor-unity-arr*:
assumes *arr f*
shows $\mathcal{I} \otimes_S f = f$
using *assms arr-unity tensor-char* $\mathcal{F}C$*.strict-arr-char* $\mathcal{F}C$*.mkarr-Diagonalize-rep*
by *simp*

lemma *tensor-arr-unity*:
assumes *arr f*
shows $f \otimes_S \mathcal{I} = f$
using *assms arr-unity tensor-char* $\mathcal{F}C$*.strict-arr-char* $\mathcal{F}C$*.mkarr-Diagonalize-rep*
by *simp*

lemma *assoc-char*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathrm{a}_S[a,\ b,\ c] = \mathcal{F}C.mkarr\ (\lfloor \mathcal{F}C.rep\ a \rfloor\ \lfloor \otimes \rfloor\ \lfloor \mathcal{F}C.rep\ b \rfloor\ \lfloor \otimes \rfloor\ \lfloor \mathcal{F}C.rep\ c \rfloor)$
using *assms tensor-tensor*(2) *assoc*$_S$*-def ideD*(1) **by** *simp*

lemma *assoc-in-hom*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows «$\mathrm{a}_S[a,\ b,\ c] : (a \otimes_S b) \otimes_S c \to a \otimes_S b \otimes_S c$»
using *assms tensor-preserves-ide ideD tensor-assoc assoc*$_S$*-def*
by (*metis* (*no-types*, *lifting*) *ide-in-hom*)

The category $\mathcal{F}_S C$ is a monoidal category.

interpretation *EMC*: *elementary-monoidal-category comp tensor*$_S$ $\mathcal{I}$ ‹$\lambda a.\ a$› ‹$\lambda a.\ a$› *assoc*$_S$
proof
show *ide* $\mathcal{I}$
using *ide-char*$_{S\flat C}$ *arr-char*$_{S\flat C}$ $\mathcal{F}C$*.rep-mkarr* $\mathcal{F}C$*.Dom-norm* $\mathcal{F}C$*.Cod-norm* $\mathcal{F}C$*.*$\mathcal{I}$*-agreement*
by *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ *iso a*
using *ide-char*$_{S\flat C}$ *arr-char*$_{S\flat C}$ *iso-char*$_{S\flat C}$ **by** *auto*
show $\bigwedge f\ a\ b\ g\ c\ d.$ ⟦ *in-hom a b f*; *in-hom c d g* ⟧ $\Longrightarrow$ *in-hom* $(a \otimes_S c)\ (b \otimes_S d)\ (f \otimes_S g)$
using *tensor-in-hom* **by** *blast*
show $\bigwedge a\ b.$ ⟦ *ide a*; *ide b* ⟧ $\Longrightarrow$ *ide* $(a \otimes_S b)$
using *tensor-preserves-ide* **by** *blast*
show $\bigwedge a\ b\ c.$ ⟦ *ide a*; *ide b*; *ide c*⟧ $\Longrightarrow$ *iso* $\mathrm{a}_S[a,\ b,\ c]$
using *tensor-preserves-ide ide-is-iso assoc*$_S$*-def* **by** *presburger*
show $\bigwedge a\ b\ c.$ ⟦ *ide a*; *ide b*; *ide c*⟧ $\Longrightarrow$ «$\mathrm{a}_S[a,\ b,\ c] : (a \otimes_S b) \otimes_S c \to a \otimes_S b \otimes_S c$»
using *assoc-in-hom* **by** *blast*
show $\bigwedge a\ b.$ ⟦ *ide a*; *ide b* ⟧ $\Longrightarrow (a \otimes_S b) \cdot_S \mathrm{a}_S[a, \mathcal{I}, b] = a \otimes_S b$
using *ide-def tensor-unity-arr assoc*$_S$*-def ideD*(1) *tensor-preserves-ide comp-ide-self*
by *simp*
show $\bigwedge f.$ *arr f* $\Longrightarrow$ *cod f* $\cdot_S$ $(\mathcal{I} \otimes_S f) = f \cdot_S$ *dom f*
using *tensor-unity-arr comp-arr-dom comp-cod-arr* **by** *presburger*
show $\bigwedge f.$ *arr f* $\Longrightarrow$ *cod f* $\cdot_S$ $(f \otimes_S \mathcal{I}) = f \cdot_S$ *dom f*
using *tensor-arr-unity comp-arr-dom comp-cod-arr* **by** *presburger*
next
fix *a*

assume a: *ide a*
show «$a : \mathcal{I} \otimes_S a \to a$»
using *a tensor-unity-arr ide-in-hom* [*of a*] **by** *fast*
show «$a : a \otimes_S \mathcal{I} \to a$»
using *a tensor-arr-unity ide-in-hom* [*of a*] **by** *fast*
next
fix *f g f′ g′*
assume *fg*: *seq g f*
assume *fg′*: *seq g′ f′*
show $(g \otimes_S g') \cdot_S (f \otimes_S f') = g \cdot_S f \otimes_S g' \cdot_S f'$
proof −
have *A*: $\mathcal{FC}.seq\ g\ f$ **and** *B*: $\mathcal{FC}.seq\ g'\ f'$
using *fg fg′ seq-char*$_{S\flat C}$ **by** *auto*
have $(g \otimes_S g') \cdot_S (f \otimes_S f') = \mathcal{FC}.D\ ((g \otimes g') \cdot (f \otimes f'))$
using *A B tensor*$_S$*-def* **by** *fastforce*
also have $... = \mathcal{FC}.D\ (g \cdot f \otimes g' \cdot f')$
using *A B* $\mathcal{FC}.interchange\ \mathcal{FC}.T\text{-}simp\ \mathcal{FC}.seqE$ **by** *metis*
also have $... = \mathcal{FC}.D\ (g \cdot f) \otimes_S \mathcal{FC}.D\ (g' \cdot f')$
using *A B tensor*$_S$*-def* $\mathcal{FC}.T\text{-}simp\ \mathcal{FC}.seqE\ \mathcal{FC}.diagonalize\text{-}tensor$ *arr-char*$_{S\flat C}$
by (*metis* (*no-types*, *lifting*) *D.preserves-reflects-arr*)
also have $... = \mathcal{FC}.D\ g \cdot_S \mathcal{FC}.D\ f \otimes_S \mathcal{FC}.D\ g' \cdot_S \mathcal{FC}.D\ f'$
using *A B* **by** *simp*
also have $... = g \cdot_S f \otimes_S g' \cdot_S f'$
using *fg fg′* $\mathcal{FC}.diagonalize\text{-}strict\text{-}arr$ **by** (*elim seqE*, *simp*)
finally show *?thesis* **by** *blast*
qed
next
fix *f0 f1 f2*
assume *f0*: *arr f0* **and** *f1*: *arr f1* **and** *f2*: *arr f2*
show $\mathrm{a}_S[cod\ f0,\ cod\ f1,\ cod\ f2] \cdot_S ((f0 \otimes_S f1) \otimes_S f2)$
$= (f0 \otimes_S f1 \otimes_S f2) \cdot_S \mathrm{a}_S[dom\ f0,\ dom\ f1,\ dom\ f2]$
using *f0 f1 f2 assoc*$_S$*-def tensor-assoc dom-tensor cod-tensor arr-tensor*
comp-cod-arr [*of f0* $\otimes_S$ *f1* $\otimes_S$ *f2 cod f0* $\otimes_S$ *cod f1* $\otimes_S$ *cod f2*]
comp-arr-dom [*of f0* $\otimes_S$ *f1* $\otimes_S$ *f2 dom f0* $\otimes_S$ *dom f1* $\otimes_S$ *dom f2*]
by *presburger*
next
fix *a b c d*
assume *a*: *ide a* **and** *b*: *ide b* **and** *c*: *ide c* **and** *d*: *ide d*
show $(a \otimes_S \mathrm{a}_S[b,\ c,\ d]) \cdot_S \mathrm{a}_S[a,\ b \otimes_S c,\ d] \cdot_S (\mathrm{a}_S[a,\ b,\ c] \otimes_S d)$
$= \mathrm{a}_S[a,\ b,\ c \otimes_S d] \cdot_S \mathrm{a}_S[a \otimes_S b,\ c,\ d]$
unfolding *assoc*$_S$*-def*
using *a b c d tensor-assoc tensor-preserves-ide ideD tensor-in-hom*
comp-arr-dom [*of a* $\otimes_S$ *b* $\otimes_S$ *c* $\otimes_S$ *d*]
by *simp*
qed

lemma *is-elementary-monoidal-category*:
shows *elementary-monoidal-category comp tensor*$_S$ $\mathcal{I}$ $(\lambda a.\ a)$ $(\lambda a.\ a)$ *assoc*$_S$ **..**

abbreviation T_{FSMC} **where** $T_{FSMC} \equiv EMC.T$
abbreviation α_{FSMC} **where** $\alpha_{FSMC} \equiv EMC.\alpha$
abbreviation ι_{FSMC} **where** $\iota_{FSMC} \equiv EMC.\iota$

lemma *is-monoidal-category*:
shows *monoidal-category comp* T_{FSMC} α_{FSMC} ι_{FSMC}
using *EMC.induces-monoidal-category* **by** *auto*

end

sublocale *free-strict-monoidal-category* $\subseteq$
elementary-monoidal-category comp $tensor_S$ $\mathcal{I}$ $\lambda a.\ a$ $\lambda a.\ a$ $assoc_S$
using *is-elementary-monoidal-category* **by** *auto*

sublocale *free-strict-monoidal-category* $\subseteq$ *monoidal-category comp* T_{FSMC} α_{FSMC} ι_{FSMC}
using *is-monoidal-category* **by** *auto*

sublocale *free-strict-monoidal-category* $\subseteq$
strict-monoidal-category comp T_{FSMC} α_{FSMC} ι_{FSMC}
using *tensor-preserves-ide lunit-agreement runit-agreement* α*-ide-simp* $assoc_S$*-def*
by *unfold-locales auto*

context *free-strict-monoidal-category*
begin

The inclusion of generators functor from C to $\mathcal{F}_S C$ is the composition of the inclusion of generators from C to $\mathcal{F}C$ and the diagonalization functor, which projects $\mathcal{F}C$ to $\mathcal{F}_S C$. As the diagonalization functor is the identity map on the image of C, the composite functor amounts to the corestriction to $\mathcal{F}_S C$ of the inclusion of generators of $\mathcal{F}C$.

interpretation *D*: *functor* $\mathcal{F}C$*.comp comp* $\mathcal{F}C$*.D*
using $\mathcal{F}C$*.diagonalize-is-functor* **by** *auto*

interpretation *I*: *composite-functor C* $\mathcal{F}C$*.comp comp* $\mathcal{F}C$*.inclusion-of-generators* $\mathcal{F}C$*.D*
proof −
interpret *functor C* $\mathcal{F}C$*.comp* $\mathcal{F}C$*.inclusion-of-generators*
using $\mathcal{F}C$*.inclusion-is-functor* **by** *blast*
show *composite-functor C* $\mathcal{F}C$*.comp comp* $\mathcal{F}C$*.inclusion-of-generators* $\mathcal{F}C$*.D* **..**
qed

definition *inclusion-of-generators*
where *inclusion-of-generators* $\equiv \mathcal{F}C$*.inclusion-of-generators*

lemma *inclusion-is-functor*:
shows *functor C comp inclusion-of-generators*
using $\mathcal{F}C$*.DoI-eq-I I.functor-axioms inclusion-of-generators-def*
by *auto*

The diagonalization functor is strict monoidal.

interpretation *D*: *strict-monoidal-functor* $\mathcal{F}C$*.comp* $\mathcal{F}C.T_{FMC}$ $\mathcal{F}C.\alpha_{FMC}$ $\mathcal{F}C.\iota_{FMC}$

```
                                    comp T_FSMC α_FSMC ι_FSMC
                                    FC.D
proof
  show FC.D FC.ι = ι
  proof −
    have FC.D FC.ι = FC.mkarr ⌊FC.rep FC.ι⌋
      unfolding FC.D-def using FC.ι-in-hom by auto
    also have ... = FC.mkarr ⌊𝟙[‖𝓘‖]⌋
      using FC.ι-def FC.rep-unity FC.rep-lunit FC.Par-Arr-norm FC.Diagonalize-norm
      by auto
    also have ... = ι
      using FC.unity_FMC-def FC.𝓘-agreement ι-def by simp
    finally show ?thesis by blast
  qed
  show ⋀f g. ⟦ FC.arr f; FC.arr g ⟧ ⟹
                FC.D (FC.tensor f g) = tensor (FC.D f) (FC.D g)
  proof −
    fix f g
    assume f: FC.arr f and g: FC.arr g
    have fg: arr (FC.D f) ∧ arr (FC.D g)
      using f g D.preserves-arr by blast
    have FC.D (FC.tensor f g) = f ⊗_S g
      using tensor_S-def by simp
    also have f ⊗_S g = FC.D (f ⊗ g)
      using f g tensor_S-def by simp
    also have ... = FC.D f ⊗_S FC.D g
      using f g fg tensor_S-def FC.T-simp FC.diagonalize-tensor arr-char_SbC
      by (metis (no-types, lifting))
    also have ... = tensor (FC.D f) (FC.D g)
      using fg T-simp by simp
    finally show FC.D (FC.tensor f g) = tensor (FC.D f) (FC.D g)
      by blast
  qed
  show ⋀a b c. ⟦ FC.ide a; FC.ide b; FC.ide c ⟧ ⟹
                  FC.D (FC.assoc a b c) = assoc (FC.D a) (FC.D b) (FC.D c)
  proof −
    fix a b c
    assume a: FC.ide a and b: FC.ide b and c: FC.ide c
    have abc: ide (FC.D a) ∧ ide (FC.D b) ∧ ide (FC.D c)
      using a b c D.preserves-ide by blast
    have abc': FC.ide (FC.D a) ∧ FC.ide (FC.D b) ∧ FC.ide (FC.D c)
       using a b c D.preserves-ide ide-char_SbC by simp
    have 1: ⋀f g. FC.arr f ⟹ FC.arr g ⟹ f ⊗_S g = FC.D (f ⊗ g)
      using tensor_S-def by simp
    have 2: ⋀f. ide f ⟹ FC.ide f using ide-char_SbC by blast
    have assoc (FC.D a) (FC.D b) (FC.D c) = FC.D a ⊗_S FC.D b ⊗_S FC.D c
      using abc α-ide-simp assoc_S-def by simp
    also have ... = FC.D a ⊗_S FC.D (FC.D b ⊗ FC.D c)
      using abc' 1 by auto
```

```
    also have ... = FC.D a ⊗_S FC.D (b ⊗ c)
      using b c FC.diagonalize-tensor by force
    also have ... = FC.D (FC.D a ⊗ FC.D (b ⊗ c))
      using 1 b c abc D.preserves-ide FC.tensor-preserves-ide ide-char_SbC
      by simp
    also have ... = FC.D (a ⊗ b ⊗ c)
      using a b c FC.diagonalize-tensor by force
    also have ... = FC.D a[a, b, c]
    proof −
      have FC.can a[a, b, c] using a b c FC.can-assoc by simp
      hence FC.ide (FC.D a[a, b, c])
        using a b c FC.ide-diagonalize-can by simp
      moreover have FC.cod (FC.D a[a, b, c]) = FC.D (a ⊗ b ⊗ c)
        using a b c FC.assoc-in-hom D.preserves-hom
        by (metis (no-types, lifting) cod-char_SbC in-homE)
      ultimately show ?thesis by simp
    qed
    also have ... = FC.D (FC.assoc a b c)
      using a b c by simp
    finally show FC.D (FC.assoc a b c) = assoc (FC.D a) (FC.D b) (FC.D c)
      by blast
  qed
qed

lemma diagonalize-is-strict-monoidal-functor:
shows strict-monoidal-functor FC.comp FC.T_FMC FC.α_FMC FC.ι_FMC
                              comp T_FSMC α_FSMC ι_FSMC
                              FC.D
  ..

interpretation φ: natural-isomorphism
                  FC.CC.comp comp D.T_DoFF.map D.FoT_C.map D.φ
  using D.structure-is-natural-isomorphism by simp
```

The diagonalization functor is part of a monoidal equivalence between the free monoidal category and the subcategory $\mathcal{F}_S C$.

```
interpretation E: equivalence-of-categories comp FC.comp FC.D map FC.ν FC.μ
  using FC.is-equivalent-to-strict-subcategory by auto

interpretation D: monoidal-functor FC.comp FC.T_FMC FC.α_FMC FC.ι_FMC
                                   comp T_FSMC α_FSMC ι_FSMC
                                   FC.D D.φ
  using D.monoidal-functor-axioms by metis

interpretation equivalence-of-monoidal-categories comp T_FSMC α_FSMC ι_FSMC
                                                  FC.comp FC.T_FMC FC.α_FMC FC.ι_FMC
                                                  FC.D D.φ I
                                                  map FC.ν FC.μ
  ..
```

The category $\mathcal{F}C$ is monoidally equivalent to its subcategory $\mathcal{F}_S C$.

theorem *monoidally-equivalent-to-free-monoidal-category*:
shows *equivalence-of-monoidal-categories comp* T_{FSMC} α_{FSMC} ι_{FSMC}
$\mathcal{F}C.comp$ $\mathcal{F}C.T_{FMC}$ $\mathcal{F}C.\alpha_{FMC}$ $\mathcal{F}C.\iota_{FMC}$
$\mathcal{F}C.D$ $D.\varphi$
map $\mathcal{F}C.\nu$ $\mathcal{F}C.\mu$
..

end

We next show that the evaluation functor induced on the free monoidal category generated by C by a functor V from C to a strict monoidal category D restricts to a strict monoidal functor on the subcategory $\mathcal{F}_S C$.

locale *strict-evaluation-functor* =
D: *strict-monoidal-category* D T_D α_D ι_D +
evaluation-map C D T_D α_D ι_D V +
$\mathcal{F}C$: *free-monoidal-category* C +
E: *evaluation-functor* C D T_D α_D ι_D V +
$\mathcal{F}_S C$: *free-strict-monoidal-category* C
for C :: $'c$ *comp* (**infixr** $\cdot_C$ 55)
and D :: $'d$ *comp* (**infixr** $\cdot_D$ 55)
and T_D :: $'d * 'd \Rightarrow 'd$
and α_D :: $'d * 'd * 'd \Rightarrow 'd$
and ι_D :: $'d$
and V :: $'c \Rightarrow 'd$
begin

notation $\mathcal{F}C.in\text{-}hom$ (« - : - → -»)
notation $\mathcal{F}_S C.in\text{-}hom$ (« - : - →$_S$ -»)

definition *map*
where *map* ≡ λf. *if* $\mathcal{F}_S C.arr\ f$ *then* $E.map\ f$ *else* $D.null$

interpretation *functor* $\mathcal{F}_S C.comp$ D *map*
unfolding *map-def*
apply *unfold-locales*
apply *simp*
using $\mathcal{F}_S C.arr\text{-}char_{SbC}$ *E.preserves-arr*
apply *simp*
using $\mathcal{F}_S C.arr\text{-}char_{SbC}$ $\mathcal{F}_S C.dom\text{-}char_{SbC}$ *E.preserves-dom*
apply *simp*
using $\mathcal{F}_S C.arr\text{-}char_{SbC}$ $\mathcal{F}_S C.cod\text{-}char_{SbC}$ *E.preserves-cod*
apply *simp*
using $\mathcal{F}_S C.arr\text{-}char_{SbC}$ $\mathcal{F}_S C.dom\text{-}char_{SbC}$ $\mathcal{F}_S C.cod\text{-}char_{SbC}$ $\mathcal{F}_S C.comp\text{-}char$ *E.preserves-comp*
by (*elim* $\mathcal{F}_S C.seqE$, *auto*)

lemma *is-functor*:

shows *functor* $\mathcal{F}_S\, C.comp\ D\ map$ **..**

Every canonical arrow is an equivalence class of canonical terms. The evaluations in D of all such terms are identities, due to the strictness of D.

lemma *ide-eval-Can*:
shows $Can\ t \Longrightarrow D.ide\ \{\!|t|\!\}$
proof (*induct t*)
  show $\bigwedge x.\ Can\ \langle x \rangle \Longrightarrow D.ide\ \{\!|\langle x \rangle|\!\}$ **by** *simp*
  show $Can\ \boldsymbol{\mathcal{I}} \Longrightarrow D.ide\ \{\!|\boldsymbol{\mathcal{I}}|\!\}$ **by** *simp*
  show $\bigwedge t1\ t2.\ [\![\ Can\ t1 \Longrightarrow D.ide\ \{\!|t1|\!\};\ Can\ t2 \Longrightarrow D.ide\ \{\!|t2|\!\};\ Can\ (t1 \otimes t2)\ ]\!] \Longrightarrow$
          $D.ide\ \{\!|t1 \otimes t2|\!\}$
    by *simp*
  show $\bigwedge t1\ t2.\ [\![\ Can\ t1 \Longrightarrow D.ide\ \{\!|t1|\!\};\ Can\ t2 \Longrightarrow D.ide\ \{\!|t2|\!\};\ Can\ (t1 \cdot t2)\ ]\!] \Longrightarrow$
          $D.ide\ \{\!|t1 \cdot t2|\!\}$
  proof −
    fix *t1 t2*
    assume *t1*: $Can\ t1 \Longrightarrow D.ide\ \{\!|t1|\!\}$
    and *t2*: $Can\ t2 \Longrightarrow D.ide\ \{\!|t2|\!\}$
    and *t12*: $Can\ (t1 \cdot t2)$
    show $D.ide\ \{\!|t1 \cdot t2|\!\}$
      using *t1 t2 t12 Can-implies-Arr eval-in-hom* [*of t1*] *eval-in-hom* [*of t2*] *D.comp-ide-arr*
      by *fastforce*
  qed
  show $\bigwedge t.\ (Can\ t \Longrightarrow D.ide\ \{\!|t|\!\}) \Longrightarrow Can\ \mathbf{l}[t] \Longrightarrow D.ide\ \{\!|\mathbf{l}[t]|\!\}$
    using *D.strict-lunit* **by** *simp*
  show $\bigwedge t.\ (Can\ t \Longrightarrow D.ide\ \{\!|t|\!\}) \Longrightarrow Can\ \mathbf{l}^{-1}[t] \Longrightarrow D.ide\ \{\!|\mathbf{l}^{-1}[t]|\!\}$
    using *D.strict-lunit* **by** *simp*
  show $\bigwedge t.\ (Can\ t \Longrightarrow D.ide\ \{\!|t|\!\}) \Longrightarrow Can\ \mathbf{r}[t] \Longrightarrow D.ide\ \{\!|\mathbf{r}[t]|\!\}$
    using *D.strict-runit* **by** *simp*
  show $\bigwedge t.\ (Can\ t \Longrightarrow D.ide\ \{\!|t|\!\}) \Longrightarrow Can\ \mathbf{r}^{-1}[t] \Longrightarrow D.ide\ \{\!|\mathbf{r}^{-1}[t]|\!\}$
    using *D.strict-runit* **by** *simp*
  fix *t1 t2 t3*
  assume *t1*: $Can\ t1 \Longrightarrow D.ide\ \{\!|t1|\!\}$
  and *t2*: $Can\ t2 \Longrightarrow D.ide\ \{\!|t2|\!\}$
  and *t3*: $Can\ t3 \Longrightarrow D.ide\ \{\!|t3|\!\}$
  show $Can\ \mathbf{a}[t1,\ t2,\ t3] \Longrightarrow D.ide\ \{\!|\mathbf{a}[t1,\ t2,\ t3]|\!\}$
  proof −
    assume $Can\ \mathbf{a}[t1,\ t2,\ t3]$
    hence *t123*: $D.ide\ \{\!|t1|\!\} \wedge D.ide\ \{\!|t2|\!\} \wedge D.ide\ \{\!|t3|\!\}$
      using *t1 t2 t3* **by** *simp*
    have $\{\!|\mathbf{a}[t1,\ t2,\ t3]|\!\} = \{\!|t1|\!\} \otimes_D \{\!|t2|\!\} \otimes_D \{\!|t3|\!\}$
      using *t123 D.strict-assoc D.assoc-in-hom* [*of* $\{\!|t1|\!\}$ $\{\!|t2|\!\}$ $\{\!|t3|\!\}$] **apply** *simp*
      by (*elim D.in-homE*, *simp*)
    thus *?thesis* **using** *t123* **by** *simp*
  qed
  show $Can\ \mathbf{a}^{-1}[t1,\ t2,\ t3] \Longrightarrow D.ide\ \{\!|\mathbf{a}^{-1}[t1,\ t2,\ t3]|\!\}$
  proof −
    assume $Can\ \mathbf{a}^{-1}[t1,\ t2,\ t3]$
    hence *t123*: $Can\ t1 \wedge Can\ t2 \wedge Can\ t3 \wedge D.ide\ \{\!|t1|\!\} \wedge D.ide\ \{\!|t2|\!\} \wedge D.ide\ \{\!|t3|\!\}$

```
      using t1 t2 t3 by simp
    have ⦇a⁻¹[t1, t2, t3]⦈
          = D.inv a_D[D.cod ⦇t1⦈, D.cod ⦇t2⦈, D.cod ⦇t3⦈] ·_D (⦇t1⦈ ⊗_D ⦇t2⦈ ⊗_D ⦇t3⦈)
      using t123 eval-Assoc' [of t1 t2 t3] Can-implies-Arr by simp
    also have ... = ⦇t1⦈ ⊗_D ⦇t2⦈ ⊗_D ⦇t3⦈
    proof −
      have D.dom a_D[⦇t1⦈, ⦇t2⦈, ⦇t3⦈] = ⦇t1⦈ ⊗_D ⦇t2⦈ ⊗_D ⦇t3⦈
      proof −
        have D.dom a_D[⦇t1⦈, ⦇t2⦈, ⦇t3⦈] = D.cod a_D[⦇t1⦈, ⦇t2⦈, ⦇t3⦈]
          using t123 D.strict-assoc by simp
        also have ... = ⦇t1⦈ ⊗_D ⦇t2⦈ ⊗_D ⦇t3⦈
          using t123 by simp
        finally show ?thesis by blast
      qed
      thus ?thesis
        using t123 D.strict-assoc D.comp-arr-dom by auto
    qed
    finally have ⦇a⁻¹[t1, t2, t3]⦈ = ⦇t1⦈ ⊗_D ⦇t2⦈ ⊗_D ⦇t3⦈ by blast
    thus ?thesis using t123 by auto
  qed
qed

lemma ide-eval-can:
assumes FC.can f
shows D.ide (E.map f)
proof −
  have f = FC.mkarr (FC.rep f)
    using assms FC.can-implies-arr FC.mkarr-rep by blast
  moreover have 1: Can (FC.rep f)
    using assms FC.Can-rep-can by simp
  moreover have D.ide ⦇FC.rep f⦈
    using assms ide-eval-Can by (simp add: 1)
  ultimately show ?thesis using assms FC.can-implies-arr E.map-def by force
qed
```

Diagonalization transports formal arrows naturally along reductions, which are canonical terms and therefore evaluate to identities of D. It follows that the evaluation in D of a formal arrow is equal to the evaluation of its diagonalization.

```
lemma map-diagonalize:
assumes f: FC.arr f
shows E.map (FC.D f) = E.map f
proof −
  interpret EQ: equivalence-of-categories
                  F_S C.comp FC.comp FC.D F_S C.map FC.ν FC.μ
    using FC.is-equivalent-to-strict-subcategory by auto
  have 1: FC.seq (F_S C.map (FC.D f)) (FC.ν (FC.dom f))
  proof
    show «FC.ν (FC.dom f) : FC.dom f → FC.D (FC.dom f)»
      using f F_S C.map-simp EQ.F.preserves-arr
```

```
    by (intro FC.in-homI, simp-all)
  show «FSC.map (FC.D f) : FC.D (FC.dom f) → FC.cod (FC.D f)»
    by (metis (no-types, lifting) EQ.F.preserves-dom EQ.F.preserves-reflects-arr
        FSC.arr-iff-in-hom FSC.cod-simp FSC.in-hom-charSbC FSC.map-simp f)
qed
have E.map (FC.ν (FC.cod f)) ·D E.map f =
     E.map (FC.D f) ·D E.map (FC.ν (FC.dom f))
proof −
  have E.map (FC.ν (FC.cod f)) ·D E.map f = E.map (FC.ν (FC.cod f) · f)
    using f by simp
  also have ... = E.map (FC.D f · FC.ν (FC.dom f))
    using f EQ.η.naturality FSC.map-simp EQ.F.preserves-arr by simp
  also have ... = E.map (FSC.map (FC.D f)) ·D E.map (FC.ν (FC.dom f))
    using f 1 E.as-nat-trans.preserves-comp-2 EQ.F.preserves-arr FSC.map-simp
    by (metis (no-types, lifting))
  also have ... = E.map (FC.D f) ·D E.map (FC.ν (FC.dom f))
    using f EQ.F.preserves-arr FSC.map-simp by simp
  finally show ?thesis by blast
qed
moreover have ⋀a. FC.ide a ⟹ D.ide (E.map (FC.ν a))
  using FC.ν-def FC.Arr-rep Arr-implies-Ide-Cod Can-red FC.can-mkarr-Can
      ide-eval-can
  by (metis (no-types, lifting) EQ.η.preserves-reflects-arr FC.seqE
      FC.comp-preserves-can FC.ideD(1) FC.ide-implies-can)
moreover have D.cod (E.map f) = D.dom (E.map (FC.ν (FC.cod f)))
  using f E.preserves-hom EQ.η.preserves-hom by simp
moreover have D.dom (E.map (FC.D f)) = D.cod (E.map (FC.ν (FC.dom f)))
    using f 1 E.preserves-seq EQ.F.preserves-arr FSC.map-simp by auto
ultimately show ?thesis
  using f D.comp-arr-dom D.ideD D.arr-dom-iff-arr E.as-nat-trans.is-natural-2
  by (metis E.preserves-cod FC.ide-cod FC.ide-dom)
qed

lemma strictly-preserves-tensor:
assumes FSC.arr f and FSC.arr g
shows map (FSC.tensor f g) = map f ⊗D map g
proof −
  have 1: FC.arr (f ⊗ g)
    using assms FSC.arr-charSbC FC.tensor-in-hom by auto
  have 2: FSC.arr (FSC.tensor f g)
    using assms FSC.tensor-in-hom [of f g] FSC.T-simp by fastforce
  have map (FSC.tensor f g) = E.map (f ⊗ g)
  proof −
    have map (FSC.tensor f g) = map (f ⊗S g)
      using assms FSC.T-simp by simp
    also have ... = map (FC.D (f ⊗ g))
      using assms FC.tensorFMC-def FSC.tensorS-def FSC.arr-charSbC by force
    also have ... = E.map (f ⊗ g)
    proof −
```

```
      interpret Diag: functor FC.comp FSC.comp FC.D
        using FC.diagonalize-is-functor by auto
      show ?thesis
        using assms 1 map-diagonalize [of f ⊗ g] Diag.preserves-arr map-def by simp
    qed
    finally show ?thesis by blast
  qed
  thus ?thesis
    using assms FSC.arr-charSbC E.strictly-preserves-tensor map-def by simp
qed

lemma is-strict-monoidal-functor:
shows strict-monoidal-functor FSC.comp FSC.TFSMC FSC.α FSC.ι D TD αD ιD map
proof
  show ⋀f g. FSC.arr f ⟹ FSC.arr g ⟹ map (FSC.tensor f g) = map f ⊗D map g
    using strictly-preserves-tensor by fast
  show map FSC.ι = ιD
  using FSC.arr-unity FSC.ι-def map-def E.map-def FC.rep-mkarr E.eval-norm D.strict-unit
    by auto
  fix a b c
  assume a: FSC.ide a and b: FSC.ide b and c: FSC.ide c
  show map (FSC.assoc a b c) = aD[map a, map b, map c]
  proof −
    have map (FSC.assoc a b c) = map a ⊗D map b ⊗D map c
      using a b c FSC.α-def FSC.assocS-def FSC.arr-tensor FSC.T-simp FSC.ideD(1)
            strictly-preserves-tensor FSC.α-ide-simp
      by presburger
    also have ... = aD[map a, map b, map c]
      using a b c D.strict-assoc D.assoc-in-hom [of map a map b map c] by auto
    finally show ?thesis by blast
  qed
qed

end

sublocale strict-evaluation-functor ⊆
          strict-monoidal-functor FSC.comp FSC.TFSMC FSC.α FSC.ι D TD αD ιD map
  using is-strict-monoidal-functor by auto

locale strict-monoidal-extension-to-free-strict-monoidal-category =
  C: category C +
  monoidal-language C +
  FSC: free-strict-monoidal-category C +
  strict-monoidal-extension C FSC.comp FSC.TFSMC FSC.α FSC.ι D TD αD ιD
                            FSC.inclusion-of-generators V F
for C :: 'c comp      (infixr ·C 55)
and D :: 'd comp      (infixr ·D 55)
and TD :: 'd ∗ 'd ⇒ 'd
and αD :: 'd ∗ 'd ∗ 'd ⇒ 'd
```

and $\iota_D :: 'd$
and $V :: 'c \Rightarrow 'd$
and $F :: 'c$ *free-monoidal-category.arr* $\Rightarrow 'd$

sublocale *strict-evaluation-functor* $\subseteq$
strict-monoidal-extension C $\mathcal{F}_S C.comp$ $\mathcal{F}_S C.T_{FSMC}$ $\mathcal{F}_S C.\alpha$ $\mathcal{F}_S C.\iota$ D T_D α_D ι_D
$\mathcal{F}_S C.inclusion\text{-}of\text{-}generators$ V *map*
proof −
interpret V: *functor* C $\mathcal{F}_S C.comp$ $\mathcal{F}_S C.inclusion\text{-}of\text{-}generators$
using $\mathcal{F}_S C.inclusion\text{-}is\text{-}functor$ **by** *auto*
show *strict-monoidal-extension* C $\mathcal{F}_S C.comp$ $\mathcal{F}_S C.T_{FSMC}$ $\mathcal{F}_S C.\alpha$ $\mathcal{F}_S C.\iota$ D T_D α_D ι_D
$\mathcal{F}_S C.inclusion\text{-}of\text{-}generators$ V *map*
proof
show $\forall f.\ C.arr\ f \longrightarrow map\ (\mathcal{F}_S C.inclusion\text{-}of\text{-}generators\ f) = V\ f$
using $V.preserves\text{-}arr$ $E.is\text{-}extension$ *map-def* $\mathcal{F}_S C.inclusion\text{-}of\text{-}generators\text{-}def$ **by** *simp*
qed
qed

context *free-strict-monoidal-category*
begin

We now have the main result of this section: the evaluation functor on $\mathcal{F}_S C$ induced by a functor V from C to a strict monoidal category D is the unique strict monoidal extension of V to $\mathcal{F}_S C$.

theorem *is-free*:
assumes *strict-monoidal-category* D T_D α_D ι_D
and *strict-monoidal-extension-to-free-strict-monoidal-category* C D T_D α_D ι_D V F
shows $F =$ *strict-evaluation-functor.map* C D T_D α_D ι_D V
proof −
interpret D: *strict-monoidal-category* D T_D α_D ι_D
using *assms*(*1*) **by** *auto*

Let F be a given extension of V to a strict monoidal functor defined on $\mathcal{F}_S C$.

interpret F: *strict-monoidal-extension-to-free-strict-monoidal-category*
C D T_D α_D ι_D V F
using *assms*(*2*) **by** *auto*

Let E_S be the evaluation functor from $\mathcal{F}_S C$ to D induced by V. Then E_S is also a strict monoidal extension of V.

interpret E_S: *strict-evaluation-functor* C D T_D α_D ι_D V **..**

Let D be the strict monoidal functor $\mathcal{F}C.D$ that projects $\mathcal{F}C$ to the subcategory $\mathcal{F}_S C$.

interpret D: *functor* $\mathcal{F}C.comp$ *comp* $\mathcal{F}C.D$
using $\mathcal{F}C.diagonalize\text{-}is\text{-}functor$ **by** *auto*
interpret D: *strict-monoidal-functor* $\mathcal{F}C.comp$ $\mathcal{F}C.T_{FMC}$ $\mathcal{F}C.\alpha$ $\mathcal{F}C.\iota$
comp T_{FSMC} α ι
$\mathcal{F}C.D$
using *diagonalize-is-strict-monoidal-functor* **by** *blast*

The composite functor $F\ o\ D$ is also an extension of V to a strict monoidal functor on $\mathcal{F}C$.

```
    interpret FoD: composite-functor FC.comp comp D FC.D F ..
    interpret FoD: strict-monoidal-functor
                   FC.comp FC.T_FMC FC.α FC.ι D T_D α_D ι_D ‹F o FC.D›
      using D.strict-monoidal-functor-axioms F.strict-monoidal-functor-axioms
            strict-monoidal-functors-compose
      by fast
    interpret FoD: strict-monoidal-extension-to-free-monoidal-category
                   C D T_D α_D ι_D V FoD.map
    proof
      show ∀f. C.arr f ⟶ FoD.map (FC.inclusion-of-generators f) = V f
      proof −
        have ⋀f. C.arr f ⟹ FoD.map (FC.inclusion-of-generators f) = V f
        proof −
          fix f
          assume f: C.arr f
          have FoD.map (FC.inclusion-of-generators f)
                 = F (FC.D (FC.inclusion-of-generators f))
            using f by simp
          also have ... = F (inclusion-of-generators f)
            using f FC.strict-arr-char' F.I.preserves-arr inclusion-of-generators-def by simp
          also have ... = V f
            using f F.is-extension by simp
          finally show FoD.map (FC.inclusion-of-generators f) = V f
            by blast
        qed
        thus ?thesis by blast
      qed
    qed
```

By the freeness of $\mathcal{F}C$, we have that $F\ o\ D$ is equal to the evaluation functor $E_S.E.map$ induced by V on $\mathcal{F}C$. Moreover, $E_S.map$ coincides with $E_S.E.map$ on $\mathcal{F}_S C$ and $F\ o\ D$ coincides with F on $\mathcal{F}_S C$. Therefore, F coincides with E on their common domain $\mathcal{F}_S C$, showing $F = E_S.map$.

```
    have ⋀f. arr f ⟹ F f = E_S.map f
      using FC.strict-arr-char' FC.is-free [of D] E_S.E.evaluation-functor-axioms
            FoD.strict-monoidal-extension-to-free-monoidal-category-axioms E_S.map-def
      by simp
    moreover have ⋀f. ¬arr f ⟹ F f = E_S.map f
      using F.is-extensional E_S.is-extensional arr-char_SbC by auto
    ultimately show F = E_S.map by blast
  qed

 end

end
```

Chapter 5

Cartesian Monoidal Category

theory *CartesianMonoidalCategory*
imports *MonoidalCategory Category3.CartesianCategory*
begin

5.1 Symmetric Monoidal Category

locale *symmetric-monoidal-category* =
  monoidal-category C T α ι +
  S: *symmetry-functor C C* +
  ToS: *composite-functor CC.comp CC.comp C S.map T* +
  σ: *natural-isomorphism CC.comp C T ToS.map σ*
for *C* :: *'a comp* (**infixr** · *55*)
and *T* :: *'a* ∗ *'a* ⇒ *'a*
and *α* :: *'a* ∗ *'a* ∗ *'a* ⇒ *'a*
and *ι* :: *'a*
and *σ* :: *'a* ∗ *'a* ⇒ *'a* +
assumes *sym-inverse*: ⟦ *ide a*; *ide b* ⟧ ⟹ *inverse-arrows* (*σ* (*a*, *b*)) (*σ* (*b*, *a*))
and *unitor-coherence*: *ide a* ⟹ l[*a*] · *σ* (*a*, *I*) = r[*a*]
and *assoc-coherence*: ⟦ *ide a*; *ide b*; *ide c* ⟧ ⟹
  α (*b*, *c*, *a*) · *σ* (*a*, *b* ⊗ *c*) · *α* (*a*, *b*, *c*)
    = (*b* ⊗ *σ* (*a*, *c*)) · *α* (*b*, *a*, *c*) · (*σ* (*a*, *b*) ⊗ *c*)
begin

  abbreviation *sym* (s[-, -])
  where *sym a b* ≡ *σ* (*a*, *b*)

end

locale *elementary-symmetric-monoidal-category* =
  elementary-monoidal-category C tensor unity lunit runit assoc
for *C* :: *'a comp* (**infixr** · *55*)
and *tensor* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⊗ *53*)
and *unity* :: *'a* (*I*)
and *lunit* :: *'a* ⇒ *'a* (l[-])

and *runit* :: $'a \Rightarrow 'a$ (r[-])
and *assoc* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (a[-, -, -])
and *sym* :: $'a \Rightarrow 'a \Rightarrow 'a$ (s[-, -]) +
assumes *sym-in-hom*: $[\![\mathit{ide}\ a;\ \mathit{ide}\ b]\!] \Longrightarrow$ «s[a, b] : $a \otimes b \to b \otimes a$»
and *sym-naturality*: $[\![\mathit{arr}\ f;\ \mathit{arr}\ g]\!] \Longrightarrow$ s[*cod* f, *cod* g] $\cdot$ ($f \otimes g$) = ($g \otimes f$) $\cdot$ s[*dom* f, *dom* g]
and *sym-inverse*: $[\![\mathit{ide}\ a;\ \mathit{ide}\ b]\!] \Longrightarrow$ *inverse-arrows* s[a, b] s[b, a]
and *unitor-coherence*: *ide* $a \Longrightarrow$ l[a] $\cdot$ s[a, $\mathcal{I}$] = r[a]
and *assoc-coherence*: $[\![\mathit{ide}\ a;\ \mathit{ide}\ b;\ \mathit{ide}\ c]\!] \Longrightarrow$
a[b, c, a] $\cdot$ s[a, $b \otimes c$] $\cdot$ a[a, b, c]
= ($b \otimes$ s[a, c]) $\cdot$ a[b, a, c] $\cdot$ (s[a, b] $\otimes$ c)
begin

lemma *sym-simps* [*simp*]:
assumes *ide a* **and** *ide b*
shows *arr* s[a, b]
and *dom* s[a, b] = $a \otimes b$
and *cod* s[a, b] = $b \otimes a$
using *assms sym-in-hom* **by** *auto*

interpretation *CC*: *product-category C C* **..**
sublocale *MC*: *monoidal-category C T* α ι
using *induces-monoidal-category* **by** *simp*

interpretation *S*: *symmetry-functor C C* **..**
interpretation *ToS*: *composite-functor CC.comp CC.comp C S.map T* **..**

definition σ :: $'a * 'a \Rightarrow 'a$
where $\sigma\ f \equiv$ *if CC.arr f then* s[*cod* (*fst f*), *cod* (*snd f*)] $\cdot$ (*fst f* $\otimes$ *snd f*) *else null*

interpretation σ: *natural-isomorphism CC.comp C T ToS.map* σ
proof −
interpret σ: *transformation-by-components CC.comp C T ToS.map* $\lambda a.$ s[*fst a*, *snd a*]
using *sym-in-hom sym-naturality*
by *unfold-locales auto*
interpret σ: *natural-isomorphism CC.comp C T ToS.map* σ*.map*
using *sym-inverse* σ*.map-simp-ide*
by *unfold-locales auto*
have $\sigma = \sigma$*.map*
using σ*-def* σ*.map-def sym-naturality* **by** *fastforce*
thus *natural-isomorphism CC.comp C T ToS.map* σ
using σ*.natural-isomorphism-axioms* **by** *presburger*
qed

interpretation *symmetric-monoidal-category C T* α ι σ
proof
show $\bigwedge a\ b.\ [\![\mathit{ide}\ a;\ \mathit{ide}\ b]\!] \Longrightarrow$ *inverse-arrows* (σ (a, b)) (σ (b, a))
using *sym-inverse comp-arr-dom* σ*-def* **by** *auto*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ *MC.lunit a* $\cdot$ σ (a, *MC.unity*) = *MC.runit a*
using *lunit-agreement* $\mathcal{I}$*-agreement sym-in-hom comp-arr-dom*

```
        unitor-coherence runit-agreement σ-def
    by simp
  show ⋀a b c. ⟦ ide a; ide b; ide c ⟧ ⟹
                 MC.assoc b c a · σ (a, MC.tensor b c) · MC.assoc a b c =
                 MC.tensor b (σ (a, c)) · MC.assoc b a c · MC.tensor (σ (a, b)) c
    using sym-in-hom tensor-preserves-ide σ-def assoc-coherence
        comp-arr-dom comp-cod-arr
    by simp
qed

lemma induces-symmetric-monoidal-category_CMC:
shows symmetric-monoidal-category C T α ι σ
  ..

end

context symmetric-monoidal-category
begin

  interpretation EMC: elementary-monoidal-category C tensor unity lunit runit assoc
    using induces-elementary-monoidal-category by auto

  lemma induces-elementary-symmetric-monoidal-category_CMC:
  shows elementary-symmetric-monoidal-category
          C tensor unity lunit runit assoc (λa b. σ (a, b))
    using σ.naturality unitor-coherence assoc-coherence sym-inverse
    by unfold-locales auto

end
```

5.2 Cartesian Monoidal Category

Here we define "cartesian monoidal category" by imposing additional properties, but not additional structure, on top of "monoidal category". The additional properties are that the unit is a terminal object and that the tensor is a categorical product, with projections defined in terms of unitors, terminators, and tensor. It then follows that the associators are induced by the product structure.

```
locale cartesian-monoidal-category =
  monoidal-category C T α ι
for C :: 'a comp                          (infixr · 55)
and T :: 'a * 'a ⇒ 'a
and α :: 'a * 'a * 'a ⇒ 'a
and ι :: 'a +
assumes terminal-unity: terminal ℐ
and tensor-is-product:
      ⟦ide a; ide b; «t_a : a → ℐ»; «t_b : b → ℐ»⟧ ⟹
        has-as-binary-product a b (r[a] · (a ⊗ t_b)) (l[b] · (t_a ⊗ b))
begin
```

sublocale *category-with-terminal-object*
  using *terminal-unity* **by** *unfold-locales blast*

lemma *is-category-with-terminal-object*:
shows *category-with-terminal-object C*
  ..

definition *the-trm* (t[-])
where *the-trm* ≡ λ*f*. *THE t*. «*t* : *dom f* → *I*»

lemma *trm-in-hom* [*intro*]:
assumes *ide a*
shows «t[*a*] : *a* → *I*»
  unfolding *the-trm-def*
  using *assms theI* [*of* λ*t*. «*t* : *dom a* → *I*»] *terminal-unity terminal-arr-unique*
  by (*metis ideD*(2) *terminalE*)

lemma *trm-simps* [*simp*]:
assumes *ide a*
shows *arr* t[*a*] **and** *dom* t[*a*] = *a* **and** *cod* t[*a*] = *I*
  using *assms trm-in-hom* **by** *auto*

interpretation *elementary-category-with-terminal-object C I the-trm*
proof
  show *ide I*
    using *ide-unity* **by** *blast*
  fix *a*
  show *ide a* ⟹ «*the-trm a* : *a* → *I*»
    using *the-trm-def theI* [*of* λ*t*. «*t* : *dom a* → *I*»] *terminalE terminal-unity* **by** *auto*
  thus ⋀*f*. ⟦*ide a*; «*f* : *a* → *I*»⟧ ⟹ *f* = *the-trm a*
    using *theI* [*of* λ*t*. «*t* : *dom a* → *I*»]
    by (*metis terminalE terminal-unity*)
qed

lemma *extends-to-elementary-category-with-terminal-object*$_{CMC}$:
shows *elementary-category-with-terminal-object C I the-trm*
  ..

definition pr_0 ($\mathfrak{p}_0$[-, -])
where pr_0 *a b* ≡ l[*b*] · (t[*a*] ⊗ *b*)

definition pr_1 ($\mathfrak{p}_1$[-, -])
where pr_1 *a b* ≡ r[*a*] · (*a* ⊗ t[*b*])

sublocale *ECC*: *elementary-category-with-binary-products C* pr_0 pr_1
proof
  fix *f g*

assume *fg*: *span f g*
have *has-as-binary-product* (*cod f*) (*cod g*) $\mathfrak{p}_1[cod\ f,\ cod\ g]$ $\mathfrak{p}_0[cod\ f,\ cod\ g]$
using *fg tensor-is-product* pr_0*-def* pr_1*-def* **by** *auto*
thus $\exists !l.\ \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$
using *fg*
by (*elim has-as-binary-productE*) *blast*
qed (*unfold* pr_0*-def* pr_1*-def*, *auto*)

lemma *induces-elementary-category-with-binary-products*$_{CMC}$:
shows *elementary-category-with-binary-products C* pr_0 pr_1
..

lemma *is-category-with-binary-products*:
shows *category-with-binary-products C*
using *ECC.is-category-with-binary-products* **by** *blast*

sublocale *category-with-binary-products C*
using *is-category-with-binary-products* **by** *blast*

sublocale *ECC*: *elementary-cartesian-category C* pr_0 pr_1 $\mathcal{I}$ *the-trm* **..**

lemma *extends-to-elementary-cartesian-category*$_{CMC}$:
shows *elementary-cartesian-category C* pr_0 pr_1 $\mathcal{I}$ *the-trm*
..

lemma *is-cartesian-category*:
shows *cartesian-category C*
using *ECC.is-cartesian-category* **by** *simp*

sublocale *cartesian-category C*
using *is-cartesian-category* **by** *blast*

abbreviation *dup* (d[-])
where *dup* $\equiv$ *ECC.dup*

abbreviation *tuple* ($\langle$-, -$\rangle$)
where $\langle f,\ g\rangle \equiv$ *ECC.tuple f g*

lemma *prod-eq-tensor*:
shows *ECC.prod* $=$ *tensor*
proof −
have $\bigwedge f\ g.$ *ECC.prod f g* $= f \otimes g$
proof −
fix *f g*
show *ECC.prod f g* $= f \otimes g$
proof (*cases arr f* $\wedge$ *arr g*)
show $\neg$ (*arr f* $\wedge$ *arr g*) $\Longrightarrow$ *?thesis*
by (*metis CC.arrE ECC.prod-def ECC.tuple-ext T.is-extensional fst-conv seqE snd-conv*)

assume *0*: $arr\ f \wedge arr\ g$
have *1*: $span\ (f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g])\ (g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g])$
  using *0* **by** *simp*
have $\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot ECC.prod\ f\ g = \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot (f \otimes g)$
proof −
  have $\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot ECC.prod\ f\ g =$
      $\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot \langle f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g],\ g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g]\rangle$
    unfolding *ECC.prod-def* **by** *simp*
  also have $... = f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g]$
    using *0 1 ECC.pr-tuple(1)* **by** *fastforce*
  also have $... = (f \cdot \mathrm{r}[dom\ f]) \cdot (dom\ f \otimes \mathrm{t}[dom\ g])$
    unfolding *pr_1-def*
    using *comp-assoc* **by** *simp*
  also have $... = (\mathrm{r}[cod\ f] \cdot (f \otimes \mathcal{I})) \cdot (dom\ f \otimes \mathrm{t}[dom\ g])$
    using *0 runit-naturality* **by** *auto*
  also have $... = \mathrm{r}[cod\ f] \cdot (f \otimes \mathcal{I}) \cdot (dom\ f \otimes \mathrm{t}[dom\ g])$
    using *comp-assoc* **by** *simp*
  also have $... = \mathrm{r}[cod\ f] \cdot (cod\ f \otimes \mathrm{t}[cod\ g]) \cdot (f \otimes g)$
    using *0 interchange comp-arr-dom comp-cod-arr trm-naturality trm-simps(1)*
    by *force*
  also have $... = (\mathrm{r}[cod\ f] \cdot (cod\ f \otimes \mathrm{t}[cod\ g])) \cdot (f \otimes g)$
    using *comp-assoc* **by** *simp*
  also have $... = \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot (f \otimes g)$
    unfolding *pr_1-def* **by** *simp*
  finally show *?thesis* **by** *blast*
qed
moreover have $\mathfrak{p}_0[cod\ f,\ cod\ g] \cdot ECC.prod\ f\ g = \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot (f \otimes g)$
proof −
  have $\mathfrak{p}_0[cod\ f,\ cod\ g] \cdot ECC.prod\ f\ g =$
      $\mathfrak{p}_0[cod\ f,\ cod\ g] \cdot \langle f \cdot \mathfrak{p}_1[dom\ f,\ dom\ g],\ g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g]\rangle$
    unfolding *ECC.prod-def* **by** *simp*
  also have $... = g \cdot \mathfrak{p}_0[dom\ f,\ dom\ g]$
    using *0 1 ECC.pr-tuple* **by** *fastforce*
  also have $... = (g \cdot \mathrm{l}[dom\ g]) \cdot (\mathrm{t}[dom\ f] \otimes dom\ g)$
    unfolding *pr_0-def*
    using *comp-assoc* **by** *simp*
  also have $... = (\mathrm{l}[cod\ g] \cdot (\mathcal{I} \otimes g)) \cdot (\mathrm{t}[dom\ f] \otimes dom\ g)$
    using *0 lunit-naturality* **by** *auto*
  also have $... = \mathrm{l}[cod\ g] \cdot (\mathcal{I} \otimes g) \cdot (\mathrm{t}[dom\ f] \otimes dom\ g)$
    using *comp-assoc* **by** *simp*
  also have $... = \mathrm{l}[cod\ g] \cdot (\mathrm{t}[cod\ f] \otimes cod\ g) \cdot (f \otimes g)$
    using *0 interchange comp-arr-dom comp-cod-arr trm-naturality trm-simps(1)*
    by *force*
  also have $... = (\mathrm{l}[cod\ g] \cdot (\mathrm{t}[cod\ f] \otimes cod\ g)) \cdot (f \otimes g)$
    using *comp-assoc* **by** *simp*
  also have $... = \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot (f \otimes g)$
    unfolding *pr_0-def* **by** *simp*
  finally show *?thesis* **by** *blast*
qed

```
      ultimately show ?thesis
        by (metis 0 1 ECC.pr-naturality(1−2) ECC.tuple-pr-arr ide-cod)
    qed
  qed
  thus ?thesis by blast
qed

lemma Prod-eq-T:
shows ECC.Prod = T
proof
  fix fg
  show ECC.Prod fg = T fg
    using prod-eq-tensor
    by (cases CC.arr fg) auto
qed
```

It is somewhat amazing that once the tensor product has been assumed to be a categorical product with the indicated projections, then the associators are forced to be those induced by the categorical product.

```
lemma pr-assoc:
assumes ide a and ide b and ide c
shows 𝔭₁[a, b ⊗ c] · a[a, b, c] = 𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]
and 𝔭₁[b, c] · 𝔭₀[a, b ⊗ c] · a[a, b, c] = 𝔭₀[a, b] · 𝔭₁[a ⊗ b, c]
and 𝔭₀[b, c] · 𝔭₀[a, b ⊗ c] · a[a, b, c] = 𝔭₀[a ⊗ b, c]
proof −
  show 𝔭₁[a, b ⊗ c] · a[a, b, c] = 𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]
  proof −
    have 𝔭₁[a, b ⊗ c] · a[a, b, c] = (r[a] · (a ⊗ ι · (t[b] ⊗ t[c]))) · a[a, b, c]
      by (metis ECC.trm-tensor ECC.unit-eq-trm arr-cod-iff-arr assms(2−3) comp-cod-arr
          dom-lunit ide-unity pr₁-def prod-eq-tensor trm-naturality trm-one trm-simps(1)
          unitor-coincidence(1))
    also have ... = (r[a] · (a ⊗ ι) · (a ⊗ t[b] ⊗ t[c])) · a[a, b, c]
      using assms interchange unit-in-hom-ax by auto
    also have ... = r[a] · (a ⊗ ι) · (a ⊗ t[b] ⊗ t[c]) · a[a, b, c]
      using comp-assoc by simp
    also have ... = r[a] · (a ⊗ ι) · a[a, I, I] · ((a ⊗ t[b]) ⊗ t[c])
      using assms assoc-naturality [of a t[b] t[c]] by force
    also have ... = r[a] · (r[a] ⊗ I) · ((a ⊗ t[b]) ⊗ t[c])
      using assms runit-char comp-assoc by simp
    also have ... = r[a] · (𝔭₁[a, b] ⊗ t[c])
      using assms comp-arr-dom comp-cod-arr interchange [of r[a] a ⊗ t[b] I t[c]]
      by (metis ECC.pr-simps(4) pr₁-def trm-simps(1) trm-simps(3))
    also have ... = r[a] · (𝔭₁[a, b] · (a ⊗ b) ⊗ I · t[c])
      using assms comp-arr-dom comp-cod-arr
      by (metis (no-types, lifting) ECC.pr-simps(4−5) prod-eq-tensor trm-simps(1,3))
    also have ... = r[a] · (𝔭₁[a, b] ⊗ I) · ((a ⊗ b) ⊗ t[c])
      using assms interchange [of 𝔭₁[a, b] a ⊗ b I t[c]]
     by (metis (no-types, lifting) ECC.pr-simps(4−5) Prod-eq-T comp-arr-dom comp-cod-arr
         fst-conv snd-conv trm-simps(1,3))
```

also have ... = $(\mathrm{r}[a] \cdot (\mathfrak{p}_1[a,\ b] \otimes \mathcal{I})) \cdot ((a \otimes b) \otimes \mathrm{t}[c])$
using *comp-assoc* **by** *simp*
also have ... = $(\mathfrak{p}_1[a,\ b] \cdot \mathrm{r}[a \otimes b]) \cdot ((a \otimes b) \otimes \mathrm{t}[c])$
using *assms runit-naturality*
by (*metis* (*no-types*, *lifting*) *ECC.cod-pr1 ECC.pr-simps*(*4*,*5*) *prod-eq-tensor*)
also have ... = $\mathfrak{p}_1[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$
using pr_1*-def comp-assoc* **by** *simp*
finally show *?thesis* **by** *blast*
qed
show $\mathfrak{p}_1[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] = \mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$
proof −
have $\mathfrak{p}_1[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] =$
$\mathrm{r}[b] \cdot (b \otimes \mathrm{t}[c]) \cdot \mathrm{l}[b \otimes c] \cdot \mathrm{a}[\mathcal{I},\ b,\ c] \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms* pr_0*-def* pr_1*-def assoc-naturality* [*of* t[*a*] *b c*] *comp-assoc* **by** *auto*
also have ... = $\mathrm{r}[b] \cdot ((b \otimes \mathrm{t}[c]) \cdot \mathrm{l}[b \otimes c]) \cdot \mathrm{a}[\mathcal{I},\ b,\ c] \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *comp-assoc* **by** *simp*
also have ... = $\mathrm{r}[b] \cdot (\mathrm{l}[b \otimes \mathcal{I}] \cdot (\mathcal{I} \otimes b \otimes \mathrm{t}[c])) \cdot \mathrm{a}[\mathcal{I},\ b,\ c] \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms lunit-naturality* [*of* $b \otimes \mathrm{t}[c]$] **by** *auto*
also have ... = $\mathrm{r}[b] \cdot \mathrm{l}[b \otimes \mathcal{I}] \cdot ((\mathcal{I} \otimes b \otimes \mathrm{t}[c]) \cdot \mathrm{a}[\mathcal{I},\ b,\ c]) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *comp-assoc* **by** *simp*
also have ... = $\mathrm{r}[b] \cdot \mathrm{l}[b \otimes \mathcal{I}] \cdot (\mathrm{a}[\mathcal{I},\ b,\ \mathcal{I}] \cdot ((\mathcal{I} \otimes b) \otimes \mathrm{t}[c])) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms assoc-naturality* [*of* $\mathcal{I}$ *b* t[*c*]] **by** *auto*
also have ... = $\mathrm{r}[b] \cdot (\mathrm{l}[b] \otimes \mathcal{I}) \cdot ((\mathcal{I} \otimes b) \otimes \mathrm{t}[c]) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms lunit-tensor* [*of b* $\mathcal{I}$] *comp-assoc*
by (*metis ide-unity lunit-tensor'*)
also have ... = $\mathrm{r}[b] \cdot (\mathrm{l}[b] \otimes \mathcal{I}) \cdot ((\mathrm{t}[a] \otimes b) \otimes \mathcal{I}) \cdot ((a \otimes b) \otimes \mathrm{t}[c])$
using *assms comp-arr-dom comp-cod-arr interchange* **by** *simp*
also have ... = $(\mathrm{r}[b] \cdot (\mathfrak{p}_0[a,\ b] \otimes \mathcal{I})) \cdot ((a \otimes b) \otimes \mathrm{t}[c])$
using *assms* pr_0*-def ECC.pr-simps*(*1*) *R.preserves-comp comp-assoc* **by** *simp*
also have ... = $(\mathfrak{p}_0[a,\ b] \cdot \mathrm{r}[a \otimes b]) \cdot ((a \otimes b) \otimes \mathrm{t}[c])$
using *assms* pr_0*-def runit-naturality* [*of* $\mathfrak{p}_0[a,\ b]$] *comp-assoc* **by** *simp*
also have ... = $\mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$
using pr_0*-def* pr_1*-def comp-assoc* **by** *simp*
finally show *?thesis* **by** *blast*
qed
show $\mathfrak{p}_0[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] = \mathfrak{p}_0[a \otimes b,\ c]$
proof −
have $\mathfrak{p}_0[b,\ c] \cdot \mathfrak{p}_0[a,\ b \otimes c] \cdot \mathrm{a}[a,\ b,\ c] =$
$\mathrm{l}[c] \cdot (\mathrm{t}[b] \otimes c) \cdot \mathrm{l}[b \otimes c] \cdot (\mathrm{t}[a] \otimes b \otimes c) \cdot \mathrm{a}[a,\ b,\ c]$
using pr_0*-def comp-assoc* **by** *simp*
also have ... = $\mathrm{l}[c] \cdot ((\mathrm{t}[b] \otimes c) \cdot \mathrm{l}[b \otimes c]) \cdot \mathrm{a}[\mathcal{I},\ b,\ c] \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms assoc-naturality* [*of* t[*a*] *b c*] *comp-assoc* **by** *simp*
also have ... = $\mathrm{l}[c] \cdot (\mathrm{l}[\mathcal{I} \otimes c] \cdot (\mathcal{I} \otimes \mathrm{t}[b] \otimes c)) \cdot \mathrm{a}[\mathcal{I},\ b,\ c] \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms lunit-naturality* [*of* t[*b*] ⊗ *c*] **by** *simp*
also have ... = $\mathrm{l}[c] \cdot \mathrm{l}[\mathcal{I} \otimes c] \cdot (\mathrm{a}[\mathcal{I},\ \mathcal{I},\ c] \cdot ((\mathcal{I} \otimes \mathrm{t}[b]) \otimes c)) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *assms assoc-naturality* [*of* $\mathcal{I}$ t[*b*] *c*] *comp-assoc* **by** *simp*
also have ... = $\mathrm{l}[c] \cdot (\mathrm{l}[\mathcal{I} \otimes c] \cdot \mathrm{a}[\mathcal{I},\ \mathcal{I},\ c]) \cdot ((\mathcal{I} \otimes \mathrm{t}[b]) \otimes c) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$
using *comp-assoc* **by** *simp*
also have ... = $\mathrm{l}[c] \cdot (\iota \otimes c) \cdot ((\mathcal{I} \otimes \mathrm{t}[b]) \otimes c) \cdot ((\mathrm{t}[a] \otimes b) \otimes c)$

```
    using assms lunit-tensor′ unitor-coincidence(1) by simp
  also have ... = l[c] · (ι ⊗ c) · ((I ⊗ t[b]) · (t[a] ⊗ b) ⊗ c)
    using assms comp-arr-dom comp-cod-arr
    by (metis arr-tensor ide-char interchange trm-simps(1−3))
  also have ... = l[c] · (ι ⊗ c) · ((t[a] ⊗ t[b]) ⊗ c)
    using assms comp-arr-dom comp-cod-arr interchange by simp
  also have ... = l[c] · (ι · (t[a] ⊗ t[b]) ⊗ c)
    using assms interchange unit-in-hom-ax by auto
  also have ... = 𝔭₀[a ⊗ b, c]
  using assms pr₀-def ECC.trm-tensor category.comp-arr-dom category-axioms prod-eq-tensor
        trm-one unit-in-hom-ax unitor-coincidence(1)
    by fastforce
  finally show ?thesis by blast
 qed
qed

lemma assoc-agreement:
assumes ide a and ide b and ide c
shows ECC.assoc a b c = a[a, b, c]
proof −
 have 𝔭₁[a, b ⊗ c] · ECC.assoc a b c = 𝔭₁[a, b ⊗ c] · a[a, b, c]
   using assms ECC.pr-assoc(3) pr-assoc(1) prod-eq-tensor by force
 moreover have 𝔭₀[a, b ⊗ c] · ECC.assoc a b c = 𝔭₀[a, b ⊗ c] · a[a, b, c]
 proof −
   have 𝔭₁[b, c] · 𝔭₀[a, b ⊗ c] · ECC.assoc a b c = 𝔭₁[b, c] · 𝔭₀[a, b ⊗ c] · a[a, b, c]
     using assms ECC.pr-assoc(2) pr-assoc(2) prod-eq-tensor by force
   moreover have 𝔭₀[b, c] · 𝔭₀[a, b ⊗ c] · ECC.assoc a b c =
                 𝔭₀[b, c] · 𝔭₀[a, b ⊗ c] · a[a, b, c]
     using assms prod-eq-tensor ECC.pr-assoc(1) pr-assoc(3) by force
   ultimately show ?thesis
     using assms prod-eq-tensor
           ECC.pr-joint-monic
             [of b c 𝔭₀[a, b ⊗ c] · ECC.assoc a b c  𝔭₀[a, b ⊗ c] · a[a, b, c]]
     by fastforce
 qed
 ultimately show ?thesis
   using assms prod-eq-tensor
         ECC.pr-joint-monic [of a b ⊗ c ECC.assoc a b c a[a, b, c]]
   by fastforce
qed

lemma lunit-eq:
assumes ide a
shows 𝔭₀[I, a] = l[a]
 by (simp add: assms comp-arr-dom pr₀-def trm-one)

lemma runit-eq:
assumes ide a
shows 𝔭₁[a, I] = r[a]
```

by (*simp add*: *assms comp-arr-dom* pr_1*-def trm-one*)

interpretation *S*: *symmetry-functor C C* **..**
interpretation *ToS*: *composite-functor CC.comp CC.comp C S.map T* **..**

interpretation σ: *natural-transformation CC.comp C T ToS.map ECC.*σ
proof −
have *ECC.Prod*′ = *ToS.map*
proof
fix *fg*
show *ECC.Prod*′ *fg* = *ToS.map fg*
using *prod-eq-tensor*
by (*metis CC.arr-char ECC.prod-def ECC.tuple-ext S.map-def ToS.is-extensional o-apply seqE*)
qed
thus *natural-transformation CC.comp C T ToS.map ECC.*σ
using *Prod-eq-T ECC.*σ*-is-natural-transformation* **by** *simp*
qed

interpretation σ: *natural-isomorphism CC.comp C T ToS.map ECC.*σ
using *ECC.sym-inverse-arrows comp-arr-dom*
by *unfold-locales auto*

sublocale *SMC*: *symmetric-monoidal-category C T* α ι *ECC.*σ
proof
show $\bigwedge a\ b.$ ⟦*ide a*; *ide b*⟧ ⟹ *inverse-arrows* (*ECC.*σ (*a*, *b*)) (*ECC.*σ (*b*, *a*))
using *comp-arr-dom* **by** *auto*
show $\bigwedge a.$ *ide a* ⟹ l[*a*] ⋅ *ECC.*σ (*a*, $\mathcal{I}$) = r[*a*]
using σ*.naturality prod-eq-tensor*
by (*metis* (*no-types*, *lifting*) *CC.arr-char ECC.prj-sym*(*1*) *R.preserves-ide* l*-ide-simp* ϱ*-ide-simp* σ*.preserves-reflects-arr comp-arr-ide fst-conv ideD*(*1*) *ideD*(*3*) *ide-unity lunit-naturality* pr_0*-def* pr_1*-def runit-naturality snd-conv trm-one*)
show $\bigwedge a\ b\ c.$ ⟦*ide a*; *ide b*; *ide c*⟧ ⟹
a[*b*, *c*, *a*] ⋅ *ECC.*σ (*a*, *b* ⊗ *c*) ⋅ a[*a*, *b*, *c*] =
(*b* ⊗ *ECC.*σ (*a*, *c*)) ⋅ a[*b*, *a*, *c*] ⋅ (*ECC.*σ (*a*, *b*) ⊗ *c*)
proof −
fix *a b c*
assume *a*: *ide a* **and** *b*: *ide b* **and** *c*: *ide c*
show a[*b*, *c*, *a*] ⋅ *ECC.*σ (*a*, *b* ⊗ *c*) ⋅ a[*a*, *b*, *c*] =
(*b* ⊗ *ECC.*σ (*a*, *c*)) ⋅ a[*b*, *a*, *c*] ⋅ (*ECC.*σ (*a*, *b*) ⊗ *c*)
using *a b c prod-eq-tensor assoc-agreement comp-arr-dom ECC.sym-assoc-coherence* [*of a b c*]
by *simp*
qed
qed

end

5.3 Elementary Cartesian Monoidal Category

locale *elementary-cartesian-monoidal-category* =
  elementary-monoidal-category C tensor unity lunit runit assoc
for C :: *'a comp* (**infixr** $\cdot$ *55*)
and *tensor* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (**infixr** $\otimes$ *53*)
and *unity* :: *'a* ($\mathcal{I}$)
and *lunit* :: *'a* $\Rightarrow$ *'a* (l[-])
and *runit* :: *'a* $\Rightarrow$ *'a* (r[-])
and *assoc* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (a[-, -, -])
and *trm* :: *'a* $\Rightarrow$ *'a* (t[-])
and *dup* :: *'a* $\Rightarrow$ *'a* (d[-]) +
assumes *trm-in-hom*: *ide a* $\Longrightarrow$ «t[a] : $a \to \mathcal{I}$»
and *trm-unity*: t[$\mathcal{I}$] = $\mathcal{I}$
and *trm-naturality*: *arr f* $\Longrightarrow$ t[*cod f*] $\cdot$ f = t[*dom f*]
and *dup-in-hom* [*intro*]: *ide a* $\Longrightarrow$ «d[a] : $a \to a \otimes a$»
and *dup-naturality*: *arr f* $\Longrightarrow$ d[*cod f*] $\cdot$ f = ($f \otimes f$) $\cdot$ d[*dom f*]
and *prj0-dup*: *ide a* $\Longrightarrow$ r[a] $\cdot$ ($a \otimes$ t[a]) $\cdot$ d[a] = a
and *prj1-dup*: *ide a* $\Longrightarrow$ l[a] $\cdot$ (t[a] $\otimes$ a) $\cdot$ d[a] = a
and *tuple-prj*: ⟦ *ide a*; *ide b* ⟧ $\Longrightarrow$ (r[a] $\cdot$ ($a \otimes$ t[b]) $\otimes$ l[b] $\cdot$ (t[a] $\otimes$ b)) $\cdot$ d[$a \otimes b$] = $a \otimes b$

context *cartesian-monoidal-category*
begin

interpretation *elementary-category-with-terminal-object C* $\mathcal{I}$ *the-trm*
using *extends-to-elementary-category-with-terminal-object*$_{CMC}$ **by** *blast*

interpretation *elementary-monoidal-category C tensor unity lunit runit assoc*
using *induces-elementary-monoidal-category* **by** *simp*

interpretation *elementary-cartesian-monoidal-category C*
tensor unity lunit runit assoc the-trm dup
using *ECC.trm-one ECC.trm-naturality ECC.tuple-in-hom′ prod-eq-tensor ECC.dup-naturality in-homI*
ECC.comp-runit-term-dup runit-eq ECC.comp-lunit-term-dup lunit-eq ECC.tuple-expansion comp-cod-arr
apply *unfold-locales*
apply *auto*
proof −
fix *a b*
assume *a*: *ide a* **and** *b*: *ide b*
show (r[a] $\cdot$ ($a \otimes$ t[b]) $\otimes$ l[b] $\cdot$ (t[a] $\otimes$ b)) $\cdot$ d[$a \otimes b$] = $a \otimes b$
using *a b ECC.tuple-pr* pr_0*-def* pr_1*-def prod-eq-tensor*
by (*metis ECC.pr-simps*(*5*) *ECC.span-pr ECC.tuple-expansion*)
qed

lemma *induces-elementary-cartesian-monoidal-category*$_{CMC}$:
shows *elementary-cartesian-monoidal-category C tensor* $\mathcal{I}$ *lunit runit assoc the-trm dup*
..

end

context *elementary-cartesian-monoidal-category*
begin

lemma *trm-simps* [*simp*]:
assumes *ide a*
shows *arr* t[*a*] **and** *dom* t[*a*] = *a* **and** *cod* t[*a*] = $\mathcal{I}$
using *assms trm-in-hom* **by** *auto*

lemma *dup-simps* [*simp*]:
assumes *ide a*
shows *arr* d[*a*] **and** *dom* d[*a*] = *a* **and** *cod* d[*a*] = *a* ⊗ *a*
using *assms dup-in-hom* **by** *auto*

interpretation *elementary-category-with-terminal-object C* $\mathcal{I}$ *trm*
apply *unfold-locales*
apply *auto*
by (*metis comp-cod-arr in-homE trm-naturality trm-unity*)

lemma *is-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object C* $\mathcal{I}$ *trm*
..

interpretation *MC*: *monoidal-category C T* α ι
using *induces-monoidal-category* **by** *auto*

interpretation *ECBP*: *elementary-category-with-binary-products C*
‹λ*a b*. l[*b*] · (t[*a*] ⊗ *b*)› ‹λ*a b*. r[*a*] · (*a* ⊗ t[*b*])›
proof −
let *?pr*$_0$ = λ*a b*. l[*b*] · (t[*a*] ⊗ *b*)
let *?pr*$_1$ = λ*a b*. r[*a*] · (*a* ⊗ t[*b*])
show *elementary-category-with-binary-products C ?pr*$_0$ *?pr*$_1$
proof
fix *a b*
assume *a*: *ide a* **and** *b*: *ide b*
show *0*: *cod* (*?pr*$_0$ *a b*) = *b*
by (*metis a arr-tensor b cod-comp cod-tensor ide-char in-homE lunit-in-hom*
seqI trm-simps(*1*,*3*))
show *1*: *cod* (*?pr*$_1$ *a b*) = *a*
by (*metis a arr-tensor b cod-comp cod-tensor ideD*(*1*,*3*) *in-homE runit-in-hom*
seqI trm-simps(*1*,*3*))
show *span* (*?pr*$_1$ *a b*) (*?pr*$_0$ *a b*)
by (*metis 0 1 a arr-cod-iff-arr b dom-cod dom-comp dom-tensor ideD*(*1*) *trm-simps*(*1*−*2*))
next
fix *f g*
assume *fg*: *span f g*

show $\exists!l.\ ?pr_1\ (cod\ f)\ (cod\ g) \cdot l = f \wedge\ ?pr_0\ (cod\ f)\ (cod\ g) \cdot l = g$
proof
show *1*: $?pr_1\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] = f \wedge$
$?pr_0\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] = g$
proof
show $?pr_1\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] = f$
proof −
have $?pr_1\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] =$
$MC.runit\ (cod\ f) \cdot (MC.tensor\ (cod\ f)\ \mathrm{t}[cod\ g] \cdot (f \otimes g)) \cdot \mathrm{d}[dom\ f]$
by (*simp add*: *fg comp-assoc runit-agreement*)
also have ... $= MC.runit\ (cod\ f) \cdot (MC.tensor\ f\ \mathcal{I} \cdot (dom\ f \otimes \mathrm{t}[dom\ g])) \cdot \mathrm{d}[dom\ f]$
using *fg*
by (*simp add*: *comp-arr-dom comp-cod-arr interchange trm-naturality*)
also have ... $= (MC.runit\ (cod\ f) \cdot MC.tensor\ f\ \mathcal{I}\ ) \cdot (dom\ f \otimes \mathrm{t}[dom\ g]) \cdot \mathrm{d}[dom\ f]$
using *comp-assoc* **by** *simp*
also have ... $= f \cdot\ ?pr_1\ (dom\ f)\ (dom\ g) \cdot \mathrm{d}[dom\ f]$
using *MC.runit-naturality* $\mathcal{I}$*-agreement fg comp-assoc runit-agreement* **by** *force*
also have ... $= f$
using *fg comp-arr-dom comp-assoc prj0-dup runit-agreement* **by** *fastforce*
finally show *?thesis* **by** *blast*
qed
show $?pr_0\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] = g$
proof −
have $?pr_0\ (cod\ f)\ (cod\ g) \cdot (f \otimes g) \cdot \mathrm{d}[dom\ f] =$
$MC.lunit\ (cod\ g) \cdot (MC.tensor\ \mathrm{t}[cod\ f]\ (cod\ g) \cdot (f \otimes g)) \cdot \mathrm{d}[dom\ f]$
by (*simp add*: *fg comp-assoc lunit-agreement*)
also have ... $= MC.lunit\ (cod\ g) \cdot (MC.tensor\ \mathcal{I}\ g \cdot (\mathrm{t}[dom\ f] \otimes dom\ g)) \cdot \mathrm{d}[dom\ f]$
using *fg*
by (*simp add*: *comp-arr-dom comp-cod-arr interchange trm-naturality*)
also have ... $= (MC.lunit\ (cod\ g) \cdot MC.tensor\ \mathcal{I}\ g) \cdot (\mathrm{t}[dom\ f] \otimes dom\ g) \cdot \mathrm{d}[dom\ f]$
using *comp-assoc* **by** *simp*
also have ... $= g \cdot\ ?pr_0\ (dom\ f)\ (dom\ g) \cdot \mathrm{d}[dom\ f]$
using *MC.lunit-naturality* $\mathcal{I}$*-agreement fg comp-assoc lunit-agreement* **by** *force*
also have ... $= g$
using *fg comp-arr-dom comp-assoc prj1-dup lunit-agreement* **by** *fastforce*
finally show *?thesis* **by** *blast*
qed
qed
fix l
assume *l*: $?pr_1\ (cod\ f)\ (cod\ g) \cdot l = f \wedge\ ?pr_0\ (cod\ f)\ (cod\ g) \cdot l = g$
show $l = (f \otimes g) \cdot \mathrm{d}[dom\ f]$
proof −
have *2*: «$l : dom\ f \to cod\ f \otimes cod\ g$»
by (*metis 1 arr-iff-in-hom cod-comp cod-tensor dom-comp fg l seqE*)
have $l = ((?pr_1\ (cod\ f)\ (cod\ g) \otimes\ ?pr_0\ (cod\ f)\ (cod\ g)) \cdot \mathrm{d}[cod\ f \otimes cod\ g]) \cdot l$
using *fg 2 tuple-prj* [*of cod f cod g*] *lunit-agreement runit-agreement comp-cod-arr*
by *auto*
also have ... $= (?pr_1\ (cod\ f)\ (cod\ g) \otimes\ ?pr_0\ (cod\ f)\ (cod\ g)) \cdot \mathrm{d}[cod\ f \otimes cod\ g] \cdot l$
using *comp-assoc* **by** *simp*

```
      also have ... = ((?pr₁ (cod f) (cod g) ⊗ ?pr₀ (cod f) (cod g)) · (l ⊗ l)) · d[dom f]
        using 2 dup-naturality [of l] comp-assoc by auto
      also have ... = (f ⊗ g) · d[dom f]
        using fg l interchange [of ?pr₁ (cod f) (cod g) l ?pr₀ (cod f) (cod g) l] by simp
      finally show ?thesis by blast
    qed
  qed
 qed
 qed

 lemma induces-elementary-category-with-binary-products_ECMC:
 shows elementary-category-with-binary-products C
         (λa b. l[b] · (t[a] ⊗ b)) (λa b. r[a] · (a ⊗ t[b]))
  ..

 sublocale cartesian-monoidal-category C T α ι
 proof
  show terminal MC.unity
   by (simp add: ℐ-agreement terminal-one)
  show ⋀a b t_a t_b. ⟦ide a; ide b; «t_a : a → MC.unity»; «t_b : b → MC.unity»⟧ ⟹
                     has-as-binary-product a b
                       (MC.runit a · MC.tensor a t_b) (MC.lunit b · MC.tensor t_a b)
   by (metis ECBP.has-as-binary-product T-simp ℐ-agreement arrI ideD(1)
       lunit-agreement runit-agreement trm-eqI)
 qed

 lemma induces-cartesian-monoidal-category_ECMC:
 shows cartesian-monoidal-category C T α ι
  ..

end


locale diagonal-functor =
  C: category C +
  CC: product-category C C
for C :: 'a comp
begin

  abbreviation map
  where map f ≡ if C.arr f then (f, f) else CC.null

  lemma is-functor:
  shows functor C CC.comp map
    using map-def by unfold-locales auto

  sublocale functor C CC.comp map
    using is-functor by simp
```

end

context *cartesian-monoidal-category*
begin

sublocale Δ: *diagonal-functor C* **..**
interpretation *To*Δ: *composite-functor C CC.comp C* Δ*.map T* **..**

sublocale δ: *natural-transformation C C map* ‹*T o* Δ*.map*› *dup*
proof
show $\bigwedge f.\ \neg$ *arr f* $\Longrightarrow$ d[*f*] = *null*
using *ECC.tuple-ext* **by** *blast*
show $\bigwedge f.$ *arr f* $\Longrightarrow$ *dom* d[*f*] = *map* (*dom f*)
using *dup-def* **by** *simp*
show $\bigwedge f.$ *arr f* $\Longrightarrow$ *cod* d[*f*] = *To*Δ*.map* (*cod f*)
by (*simp add*: *prod-eq-tensor*)
show $\bigwedge f.$ *arr f* $\Longrightarrow$ *To*Δ*.map f* $\cdot$ d[*dom f*] = d[*f*]
using *ECC.tuple-expansion prod-eq-tensor* **by** *force*
show $\bigwedge f.$ *arr f* $\Longrightarrow$ d[*cod f*] $\cdot$ *map f* = d[*f*]
by (*simp add*: *comp-cod-arr dup-def*)
qed

end

5.4 Cartesian Monoidal Category from Cartesian Category

A cartesian category extends to a cartesian monoidal category by using the product structure to obtain the various canonical maps.

context *elementary-cartesian-category*
begin

interpretation *CC*: *product-category C C* **..**
interpretation *CCC*: *product-category C CC.comp* **..**
interpretation *T*: *binary-functor C C C Prod*
using *binary-functor-Prod* **by** *simp*
interpretation *T*: *binary-endofunctor C Prod* **..**
interpretation *ToTC*: *functor CCC.comp C T.ToTC*
using *T.functor-ToTC* **by** *auto*
interpretation *ToCT*: *functor CCC.comp C T.ToCT*
using *T.functor-ToCT* **by** *auto*

interpretation α: *natural-isomorphism CCC.comp C T.ToTC T.ToCT* α
using α*-is-natural-isomorphism* **by** *blast*

interpretation *L*: *functor C C* ‹$\lambda f.$ *Prod* (*cod* ι, *f*)›
using *unit-is-terminal-arr T.fixing-ide-gives-functor-1* **by** *simp*
interpretation *L*: *endofunctor C* ‹$\lambda f.$ *Prod* (*cod* ι, *f*)› **..**
interpretation l: *transformation-by-components C C*

$\langle \lambda f.\ Prod\ (cod\ \iota,\ f) \rangle$ *map* $\langle \lambda a.\ pr0\ (cod\ \iota)\ a \rangle$
using *unit-is-terminal-arr*
by *unfold-locales auto*
interpretation l: *natural-isomorphism C C* $\langle \lambda f.\ Prod\ (cod\ \iota,\ f) \rangle$ *map* l.*map*
using l.*map-simp-ide inverse-arrows-lunit ide-one*
by *unfold-locales auto*
interpretation *L*: *equivalence-functor C C* $\langle \lambda f.\ Prod\ (cod\ \iota,\ f) \rangle$
using l.*natural-isomorphism-axioms naturally-isomorphic-def*
L.isomorphic-to-identity-is-equivalence
by *blast*

interpretation *R*: *functor C C* $\langle \lambda f.\ Prod\ (f,\ cod\ \iota) \rangle$
using *unit-is-terminal-arr T.fixing-ide-gives-functor-2* **by** *simp*
interpretation *R*: *endofunctor C* $\langle \lambda f.\ Prod\ (f,\ cod\ \iota) \rangle$ **..**
interpretation ϱ: *transformation-by-components C C*
$\langle \lambda f.\ Prod\ (f,\ cod\ \iota) \rangle$ *map* $\langle \lambda a.\ \mathfrak{p}_1[a,\ cod\ \iota] \rangle$
using *unit-is-terminal-arr*
by *unfold-locales auto*
interpretation ϱ: *natural-isomorphism C C* $\langle \lambda f.\ Prod\ (f,\ cod\ \iota) \rangle$ *map* ϱ.*map*
using ϱ.*map-simp-ide inverse-arrows-runit ide-one*
by *unfold-locales auto*
interpretation *R*: *equivalence-functor C C* $\langle \lambda f.\ Prod\ (f,\ cod\ \iota) \rangle$
using ϱ.*natural-isomorphism-axioms naturally-isomorphic-def*
R.isomorphic-to-identity-is-equivalence
by *blast*

interpretation *MC*: *monoidal-category C Prod* α ι
using *ide-one* ι*-is-iso pentagon comp-assoc* α*-simp-ide comp-cod-arr*
by *unfold-locales auto*

lemma *induces-monoidal-category*$_{ECC}$:
shows *monoidal-category C Prod* α ι
..

lemma *unity-agreement*:
shows *MC.unity* = **1**
using *ide-one* **by** *simp*

lemma *assoc-agreement*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *MC.assoc a b c* = $\mathrm{a}[a,\ b,\ c]$
using *assms assoc-def* α*-simp-ide* **by** *auto*

lemma *assoc'-agreement*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *MC.assoc' a b c* = $\mathrm{a}^{-1}[a,\ b,\ c]$
using *assms inverse-arrows-assoc inverse-unique* α*-simp-ide* **by** *auto*

lemma *runit-char-eqn*:

assumes *ide a*
shows $\mathrm{r}[a] \otimes \mathbf{1} = (a \otimes \iota) \cdot \mathrm{a}[a, \mathbf{1}, \mathbf{1}]$
using *assms ide-one assoc-def comp-assoc prod-tuple comp-cod-arr*
by (*intro pr-joint-monic* [*of a* **1** $\mathrm{r}[a] \otimes \mathbf{1}$ $(a \otimes \iota) \cdot \mathrm{a}[a, \mathbf{1}, \mathbf{1}]$]) *auto*

lemma *runit-agreement*:
assumes *ide a*
shows *MC.runit a* = $\mathrm{r}[a]$
using *assms unity-agreement assoc-agreement MC.runit-char*(*2*) *runit-char-eqn ide-one*
by (*metis* (*no-types*, *lifting*) *MC.runit-eqI fst-conv runit-in-hom snd-conv*)

lemma *lunit-char-eqn*:
assumes *ide a*
shows $\mathbf{1} \otimes \mathrm{l}[a] = (\iota \otimes a) \cdot \mathrm{a}^{-1}[\mathbf{1}, \mathbf{1}, a]$
proof (*intro pr-joint-monic* [*of* **1** *a* $\mathbf{1} \otimes \mathrm{l}[a]$ $(\iota \otimes a) \cdot \mathrm{a}^{-1}[\mathbf{1}, \mathbf{1}, a]$])
show *ide a* **by** *fact*
show *ide* **1**
using *ide-one* **by** *simp*
show *seq* $\mathrm{l}[a]$ $(\mathbf{1} \otimes \mathrm{l}[a])$
using *assms ide-one* **by** *simp*
show $\mathrm{l}[a] \cdot (\mathbf{1} \otimes \mathrm{l}[a]) = \mathrm{l}[a] \cdot (\iota \otimes a) \cdot \mathrm{a}^{-1}[\mathbf{1}, \mathbf{1}, a]$
using *assms ide-one assoc'-def comp-assoc prod-tuple comp-cod-arr* **by** *simp*
show $\mathfrak{p}_1[\mathbf{1}, a] \cdot \mathit{prod}\ \mathbf{1}\ (\mathit{lunit}\ a) = \mathfrak{p}_1[\mathbf{1}, a] \cdot \mathit{prod}\ \iota\ a \cdot \mathit{assoc'}\ \mathbf{1}\ \mathbf{1}\ a$
using *assms ide-one assoc'-def comp-cod-arr prod-tuple pr-naturality*
apply *simp*
by (*metis* (*full-types*) *cod-pr0 cod-pr1 elementary-category-with-binary-products.ide-prod elementary-category-with-binary-products-axioms pr-simps*(*1*−*2*,*4*−*5*) *trm-naturality trm-one*)
qed

lemma *lunit-agreement*:
assumes *ide a*
shows *MC.lunit a* = $\mathrm{l}[a]$
by (*metis* (*no-types*, *lifting*) *MC.lunit-eqI assms assoc'-agreement fst-conv ide-one lunit-char-eqn lunit-in-hom snd-conv unity-agreement*)

interpretation *CMC*: *cartesian-monoidal-category C Prod* α ι
proof
show *terminal MC.unity*
by (*simp add*: *terminal-one unity-agreement*)
fix *a b* t_a t_b
assume *a*: *ide a* **and** *b*: *ide b*
and t_a: «$t_a : a \to \mathit{MC.unity}$» **and** t_b: «$t_b : b \to \mathit{MC.unity}$»
have *0*: $\mathfrak{p}_0[a, b] = \mathit{MC.lunit}\ b \cdot \mathit{MC.tensor}\ \mathrm{t}[a]\ b$
by (*metis* (*no-types*, *lifting*) *a b ide-char cod-pr0 comp-cod-arr lunit-agreement pr-naturality*(*1*) *pr-simps*(*1*) *prod.sel*(*1*−*2*) *trm-simps*(*1*−*3*))
have *1*: $\mathfrak{p}_1[a, b] = \mathit{MC.runit}\ a \cdot \mathit{MC.tensor}\ a\ \mathrm{t}[b]$
by (*metis* (*no-types*, *lifting*) *a b cod-pr1 comp-cod-arr ide-char pr-naturality*(*2*) *pr-simps*(*4*) *prod.sel*(*1*−*2*) *runit-agreement trm-simps*(*1*−*3*))

have *2*: $\mathrm{t}[a] = t_a \wedge \mathrm{t}[b] = t_b$
using *a b* t_a t_b *terminal-arr-unique trm-eqI unity-agreement* **by** *metis*
show *has-as-binary-product a b* (*MC.runit a* · *MC.tensor a* t_b) (*MC.lunit b* · *MC.tensor* t_a *b*)
using *a b 0 1 2 has-as-binary-product* **by** *force*
qed

lemma *extends-to-cartesian-monoidal-category*$_{ECC}$:
shows *cartesian-monoidal-category C Prod* α ι
..

lemma *trm-agreement*:
assumes *ide a*
shows *CMC.the-trm a* = $\mathrm{t}[a]$
by (*metis assms CMC.extends-to-elementary-category-with-terminal-object*$_{CMC}$ *elementary-category-with-terminal-object.trm-eqI trm-in-hom unity-agreement*)

lemma *pr-agreement*:
assumes *ide a* **and** *ide b*
shows *CMC.pr*$_0$ *a b* = $\mathfrak{p}_0[a, b]$ **and** *CMC.pr*$_1$ *a b* = $\mathfrak{p}_1[a, b]$
proof −
show *CMC.pr*$_0$ *a b* = $\mathfrak{p}_0[a, b]$
unfolding *CMC.pr*$_0$*-def*
using *assms*(*1*−*2*) *lunit-agreement pr-expansion*(*1*) *trm-agreement* **by** *auto*
show *CMC.pr*$_1$ *a b* = $\mathfrak{p}_1[a, b]$
unfolding *CMC.pr*$_1$*-def*
using *assms*(*1*−*2*) *pr-expansion*(*2*) *runit-agreement trm-agreement* **by** *force*
qed

lemma *dup-agreement*:
assumes *ide a*
shows *CMC.dup a* = $\mathrm{d}[a]$
by (*metis* (*no-types*, *lifting*) *CMC.ECC.tuple-eqI assms ideD*(*1*) *pr-agreement*(*1*−*2*) *pr-dup*(*1*−*2*))

end

5.5 Cartesian Monoidal Category from Elementary Cartesian Category

context *elementary-cartesian-category*
begin

interpretation *MC*: *monoidal-category C Prod* α ι
using *induces-monoidal-category*$_{ECC}$ **by** *blast*

lemma *triangle*:

assumes *ide a* **and** *ide b*
shows $(a \otimes \mathrm{l}[b]) \cdot \mathrm{a}[a, \mathbf{1}, b] = \mathrm{r}[a] \otimes b$
using *assms MC.triangle* [*of a b*] *assoc-agreement ide-one lunit-agreement runit-agreement unity-agreement fst-conv snd-conv*
by (*metis* (*no-types*, *lifting*))

lemma *induces-elementary-cartesian-monoidal-category*$_{ECC}$:
shows *elementary-cartesian-monoidal-category* (·) *prod* **1** *lunit runit assoc trm dup*
using *ide-one inverse-arrows-lunit inverse-arrows-runit inverse-arrows-assoc interchange lunit-naturality runit-naturality assoc-naturality triangle pentagon comp-assoc trm-one trm-naturality in-homI prod-tuple isoI arr-dom MC.tensor-in-homI comp-arr-dom comp-cod-arr*
apply *unfold-locales*
apply *simp-all*
apply *blast*
apply *blast*
by *meson*

end

context *cartesian-category*
begin

interpretation *ECC*: *elementary-cartesian-category C some-pr0 some-pr1 some-terminal some-terminator*
using *extends-to-elementary-cartesian-category* **by** *simp*

lemma *extends-to-cartesian-monoidal-category*$_{CC}$:
shows *cartesian-monoidal-category C ECC.Prod ECC.α ECC.ι*
using *ECC.extends-to-cartesian-monoidal-category*$_{ECC}$ **by** *blast*

end

end

Bibliography

[1] J. Bénabou. Catégories avec multiplication. *C. R. Acad. Sci. Paris*, 258:1887 – 1890, 1963.

[2] P. Etingof, S. Gelaki, D. Nikshych, and V. Ostrik. *Tensor Categories*, volume 205 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2015.

[3] G. M. Kelly. On MacLane's conditions for coherence of natural associativities, commutativities, *etc. Journal of Algebra*, 1:397 – 402, 1964.

[4] S. MacLane. Natural associativity and commutativity. *Rice. Univ. Stud.*, 49:28 – 46, 1963.

[5] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

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