Strict Omega Categories

Anthony Bordg Adrián Doña Mateo

March 17, 2025

Abstract

This theory formalises a definition of strict ω-categories and the strict ω-category of pasting diagrams, following [1]. It is the first step towards a formalisation of weak infinity categories à la Batanin–Leinster.

Contents

theory *Globular-Set*

imports *HOL−Library.FuncSet*

begin

1 Background material on extensional functions

lemma *PiE-imp-Pi*: $f \in A \to_E B \Longrightarrow f \in A \to B$ **by** *fast*

lemma *PiE-iff'*: $f \in A \to_E B = (f \in A \to B \wedge f \in extensional\ A)$
by (*simp add*: *PiE-iff Pi-iff*)

abbreviation *composing* (‹- ∘ - ↓ -› [*60*,*0*,*60*]*59*)
  where $g \circ f \downarrow D \equiv$ *compose* $D\ g\ f$

lemma *compose-PiE*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow g \circ f \downarrow A \in A \to_E C$
  by (*metis funcset-compose compose-extensional PiE-iff′*)

lemma *compose-eq-iff*: $(g \circ f \downarrow A = k \circ h \downarrow A) = (\forall x \in A.\ g\ (f\ x) = k\ (h\ x))$
proof (*safe*)
  fix x **assume** $g \circ f \downarrow A = k \circ h \downarrow A$ $x \in A$
  then show $g\ (f\ x) = k\ (h\ x)$ **by** (*metis compose-eq*)
next
  assume $\forall x \in A.\ g\ (f\ x) = k\ (h\ x)$
  hence $\bigwedge x.\ x \in A \Longrightarrow (g \circ f \downarrow A)\ x = (k \circ h \downarrow A)\ x$ **by** (*metis compose-eq*)
  then show $g \circ f \downarrow A = k \circ h \downarrow A$ **by** (*metis extensionalityI compose-extensional*)
qed

lemma *compose-eq-if*: $(\bigwedge x.\ x \in A \Longrightarrow g\ (f\ x) = k\ (h\ x)) \Longrightarrow g \circ f \downarrow A = k \circ h \downarrow A$
  using *compose-eq-iff* **by** *blast*

lemma *compose-compose-eq-iff2*: $(h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A) =$
  $(\forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x)))$
  by (*simp add*: *compose-eq compose-eq-iff*)

lemma *compose-compose-eq-iff1*: **assumes** $f \in A \to B$ $f' \in A \to B$
  shows $((h \circ g \downarrow B) \circ f \downarrow A = (h' \circ g' \downarrow B) \circ f' \downarrow A) = (\forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x)))$
proof −
  have $(h \circ g \downarrow B) \circ f \downarrow A = h \circ (g \circ f \downarrow A) \downarrow A$ **by** (*metis assms*(*1*) *compose-assoc*)
  moreover have $(h' \circ g' \downarrow B) \circ f' \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A$ **by** (*metis assms*(*2*) *compose-assoc*)
  ultimately have h: $((h \circ g \downarrow B) \circ f \downarrow A = (h' \circ g' \downarrow B) \circ f' \downarrow A) =$
    $(h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A)$ **by** *presburger*
  then show *?thesis* **by** (*simp only*: *h compose-compose-eq-iff2*)
qed

lemma *compose-compose-eq-if1*: $[\![f \in A \to B;\ f' \in A \to B;\ \forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x))]\!] \Longrightarrow$
  $(h \circ g \downarrow B) \circ f \downarrow A = (h' \circ g' \downarrow B) \circ f' \downarrow A$
  using *compose-compose-eq-iff1* **by** *blast*

lemma *compose-compose-eq-if2*: $\forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x)) \Longrightarrow$
  $h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A$
  using *compose-compose-eq-iff2* **by** *blast*

lemma *compose-restrict-eq1*: $f \in A \to B \Longrightarrow$ *restrict* $g\ B \circ f \downarrow A = g \circ f \downarrow A$
  by (*smt* (*verit*) *PiE compose-eq-iff restrict-apply′*)

lemma *compose-restrict-eq2*: $g \circ$ (*restrict* $f\ A) \downarrow A = g \circ f \downarrow A$

by (*metis* (*mono-tags*, *lifting*) *compose-eq-if restrict-apply'*)

lemma *compose-Id-eq-restrict*: $g \circ (\lambda x \in A.\ x) \downarrow A = restrict\ g\ A$
by (*smt* (*verit*) *compose-restrict-eq1 compose-def restrict-apply' restrict-ext*)

2 Globular sets

2.1 Globular sets

We define a locale *globular-set* that encodes the cell data of a strict ω-category [1, Def 1.4.5]. The elements of $X\ n$ are the n-cells, and the maps s and t give the source and target of a cell, respectively.

locale *globular-set* =
fixes $X :: nat \Rightarrow {'a}\ set$ **and** $s :: nat \Rightarrow {'a} \Rightarrow {'a}$ **and** $t :: nat \Rightarrow {'a} \Rightarrow {'a}$
assumes *s-fun*: $s\ n \in X\ (Suc\ n) \to X\ n$
and *t-fun*: $t\ n \in X\ (Suc\ n) \to X\ n$
and *s-comp*: $x \in X\ (Suc\ (Suc\ n)) \Longrightarrow s\ n\ (t\ (Suc\ n)\ x) = s\ n\ (s\ (Suc\ n)\ x)$
and *t-comp*: $x \in X\ (Suc\ (Suc\ n)) \Longrightarrow t\ n\ (s\ (Suc\ n)\ x) = t\ n\ (t\ (Suc\ n)\ x)$
begin

lemma *s-comp'*: $s\ n \circ t\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n)) = s\ n \circ s\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n))$
by (*metis* (*full-types*) *compose-eq-if s-comp*)

lemma *t-comp'*: $t\ n \circ s\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n)) = t\ n \circ t\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n))$
by (*metis* (*full-types*) *compose-eq-if t-comp*)

These are the generalised source and target maps. The arguments are the dimension of the input and output, respectively. They allow notation similar to s^{m-p} in [1].

fun $s' :: nat \Rightarrow nat \Rightarrow {'a} \Rightarrow {'a}$ **where**
$s'\ 0\ 0 = id\ |$
$s'\ 0\ (Suc\ n) = undefined\ |$
$s'\ (Suc\ m)\ n = (if\ Suc\ m < n\ then\ undefined$
$else\ if\ Suc\ m = n\ then\ id$
$else\ s'\ m\ n \circ s\ m)$

fun $t' :: nat \Rightarrow nat \Rightarrow {'a} \Rightarrow {'a}$ **where**
$t'\ 0\ 0 = id\ |$
$t'\ 0\ (Suc\ n) = undefined\ |$
$t'\ (Suc\ m)\ n = (if\ Suc\ m < n\ then\ undefined$
$else\ if\ Suc\ m = n\ then\ id$
$else\ t'\ m\ n \circ t\ m)$

lemma *s'-n-n* [*simp*]: $s'\ n\ n = id$
by (*cases n*, *simp-all*)

lemma *s′-Suc-n-n* [*simp*]: s' (*Suc* n) $n = s\ n$
  by *simp*

lemma *s′-Suc-Suc-n-n* [*simp*]: s' (*Suc* (*Suc* n)) $n = s\ n \circ s$ (*Suc* n)
  by *simp*

lemma *s′-Suc* [*simp*]: $n \leq m \Longrightarrow s'$ (*Suc* m) $n = s'\ m\ n \circ s\ m$
  by *simp*

lemma *s′-Suc′*: $n < m \Longrightarrow s'\ m\ n = s\ n \circ s'\ m$ (*Suc* n)
proof (*induction m arbitrary*: *n*)
  case *0*
  then show *?case* **by** *blast*
next
  case (*Suc m*)
  hence $n \leq m$ **by** *fastforce*
  show *?case* **proof** (*cases* $n = m$, *simp*)
    assume $n \neq m$
    then show s' (*Suc* m) $n = s\ n \circ s'$ (*Suc* m) (*Suc* n) **using** *Suc* **by** *fastforce*
  qed
qed

lemma *t′-n-n* [*simp*]: $t'\ n\ n = id$
  by (*cases n*, *simp-all*)

lemma *t′-Suc-n-n* [*simp*]: t' (*Suc* n) $n = t\ n$
  by *simp*

lemma *t′-Suc-Suc-n-n* [*simp*]: t' (*Suc* (*Suc* n)) $n = t\ n \circ t$ (*Suc* n)
  by *simp*

lemma *t′-Suc* [*simp*]: $n \leq m \Longrightarrow t'$ (*Suc* m) $n = t'\ m\ n \circ t\ m$
  by *simp*

lemma *t′-Suc′*: $n < m \Longrightarrow t'\ m\ n = t\ n \circ t'\ m$ (*Suc* n)
proof (*induction m arbitrary*: *n*)
  case *0*
  then show *?case* **by** *blast*
next
  case (*Suc m*)
  hence $n \leq m$ **by** *fastforce*
  show *?case* **proof** (*cases* $n = m$, *simp*)
    assume $n \neq m$
    then show t' (*Suc* m) $n = t\ n \circ t'$ (*Suc* m) (*Suc* n) **using** *Suc* **by** *fastforce*
  qed
qed

lemma *s′-fun*: $n \leq m \Longrightarrow s'\ m\ n \in X\ m \rightarrow X\ n$
proof (*induction m arbitrary*: *n*)

```
  case 0
  thus ?case by force
next
  case (Suc m)
  thus ?case proof (cases n = Suc m)
    case True
    then show ?thesis by auto
  next
    case False
    hence n ≤ m using ‹n ≤ Suc m› by force
    thus ?thesis using Suc.IH s-fun s′-Suc by auto
  qed
qed

lemma t′-fun: n ≤ m ⟹ t′ m n ∈ X m → X n
proof (induction m arbitrary: n)
  case 0
  thus ?case by force
next
  case (Suc m)
  thus ?case proof (cases n = Suc m)
    case True
    then show ?thesis by auto
  next
    case False
    hence n ≤ m using ‹n ≤ Suc m› by force
    thus ?thesis using Suc.IH t-fun t′-Suc by auto
  qed
qed

lemma s′-comp: ⟦ n < m; x ∈ X m ⟧ ⟹ s n (t′ m (Suc n) x) = s′ m n x
proof (induction m − n arbitrary: n)
  case 0
  then show ?case by force
next
  case IH: (Suc k)
  show ?case proof (cases k)
    case 0
    with IH(2) have m = Suc n by fastforce
    then show ?thesis using s′-Suc′ by auto
  next
    case (Suc k′)
    with ‹Suc k = m − n› have hle: Suc (Suc n) ≤ m by simp
    hence Suc n < m by force
    hence Suc (Suc n) ≤ m by fastforce
    have s n (t′ m (Suc n) x)
      = s n (t (Suc n) (t′ m (Suc (Suc n)) x)) using t′-Suc′ ‹Suc n < m› by simp
    also have ... = s n (s (Suc n) (t′ m (Suc (Suc n)) x))
      using t′-fun ‹Suc (Suc n) ≤ m› s-comp IH(4) by blast
```

    also have ... = $s\ n\ (s'\ m\ (Suc\ n)\ x)$
      using *IH Suc-diff-Suc Suc-inject* ‹$Suc\ n < m$› **by** *presburger*
    finally show *?thesis* **using** ‹$n < m$› *s'-Suc'* **by** *simp*
  qed
qed

lemma *t'-comp*: $[\![\ n < m;\ x \in X\ m\ ]\!] \Longrightarrow t\ n\ (s'\ m\ (Suc\ n)\ x) = t'\ m\ n\ x$
proof (*induction* $m - n$ *arbitrary*: n)
  case *0*
  then show *?case* **by** *force*
next
  case *IH*: (*Suc k*)
  show *?case* **proof** (*cases k*)
    case *0*
    with *IH*(*2*) **have** $m = Suc\ n$ **by** *fastforce*
    then show *?thesis* **using** *IH.prems*(*1*) **by** *auto*
  next
    case (*Suc k'*)
    with ‹$Suc\ k = m - n$› **have** *hle*: $Suc\ (Suc\ n) \leq m$ **by** *simp*
    hence $Suc\ n < m$ **by** *force*
    hence $Suc\ (Suc\ n) \leq m$ **by** *fastforce*
    have $t\ n\ (s'\ m\ (Suc\ n)\ x)$
      $= t\ n\ (s\ (Suc\ n)\ (s'\ m\ (Suc\ (Suc\ n))\ x))$ **using** *s'-Suc'* ‹$Suc\ n < m$› **by** *simp*
    also have ... = $t\ n\ (t\ (Suc\ n)\ (s'\ m\ (Suc\ (Suc\ n))\ x))$
      using *s'-fun* ‹$Suc\ (Suc\ n) \leq m$› *t-comp IH*(*4*) **by** *blast*
    also have ... = $t\ n\ (t'\ m\ (Suc\ n)\ x)$
      using *IH Suc-diff-Suc Suc-inject* ‹$Suc\ n < m$› **by** *presburger*
    finally show *?thesis* **using** ‹$n < m$› *t'-Suc'* **by** *simp*
  qed
qed

The following predicates and sets are needed to define composition in an ω-category.

definition *is-parallel-pair* :: $nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
is-parallel-pair $m\ n\ x\ y \equiv n \leq m \wedge x \in X\ m \wedge y \in X\ m \wedge s'\ m\ n\ x = s'\ m\ n\ y \wedge t'\ m\ n\ x = t'\ m\ n\ y$

[1, p. 44]

definition *is-composable-pair* :: $nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
is-composable-pair $m\ n\ y\ x \equiv n < m \wedge y \in X\ m \wedge x \in X\ m \wedge t'\ m\ n\ x = s'\ m\ n\ y$

definition *composable-pairs* :: $nat \Rightarrow nat \Rightarrow ('a \times 'a)\ set$ **where**
composable-pairs $m\ n = \{(y, x).$ *is-composable-pair* $m\ n\ y\ x\}$

lemma *composable-pairs-empty*: $m \leq n \Longrightarrow$ *composable-pairs* $m\ n = \{\}$
  using *is-composable-pair-def composable-pairs-def* **by** *simp*

end

2.2 Maps between globular sets

We define maps between globular sets to be natural transformations of the corresponding functors [1, Def 1.4.5].

locale *globular-map* = *source*: *globular-set* X s_X t_X + *target*: *globular-set* Y s_Y t_Y
for X s_X t_X Y s_Y t_Y +
fixes φ :: *nat* $\Rightarrow$ *'a* $\Rightarrow$ *'b*
assumes *map-fun*: $\varphi\ m \in X\ m \to Y\ m$
and *is-natural-wrt-s*: $x \in X\ (Suc\ m) \Longrightarrow \varphi\ m\ (s_X\ m\ x) = s_Y\ m\ (\varphi\ (Suc\ m)\ x)$
and *is-natural-wrt-t*: $x \in X\ (Suc\ m) \Longrightarrow \varphi\ m\ (t_X\ m\ x) = t_Y\ m\ (\varphi\ (Suc\ m)\ x)$
begin

lemma *is-natural-wrt-s'*: $[\![\ n \leq m;\ x \in X\ m\ ]\!] \Longrightarrow \varphi\ n\ (source.s'\ m\ n\ x) = target.s'\ m\ n\ (\varphi\ m\ x)$
proof (*induction* $m - n$ *arbitrary*: n)
case *0*
hence $m = n$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Suc k*)
hence $n < m$ **by** *force*
hence $Suc\ n \leq m$ **by** *auto*
have $\varphi\ n\ (source.s'\ m\ n\ x) = \varphi\ n\ (s_X\ n\ (source.s'\ m\ (Suc\ n)\ x))$
using *source.s'-Suc'* ‹$n < m$› **by** *simp*
also have $\ldots = s_Y\ n\ (\varphi\ (Suc\ n)\ (source.s'\ m\ (Suc\ n)\ x))$
using *source.s'-fun* ‹$Suc\ n \leq m$› *Suc(1)* *Suc(4)* *is-natural-wrt-s* **by** *blast*
also have $\ldots = s_Y\ n\ (target.s'\ m\ (Suc\ n)\ (\varphi\ m\ x))$
using *Suc* ‹$Suc\ n \leq m$› *Suc-diff-Suc Suc-inject* ‹$n < m$› **by** *presburger*
finally show *?case* **using** *target.s'-Suc'* ‹$n < m$› **by** *simp*
qed

lemma *is-natural-wrt-t'*: $[\![\ n \leq m;\ x \in X\ m\ ]\!] \Longrightarrow \varphi\ n\ (source.t'\ m\ n\ x) = target.t'\ m\ n\ (\varphi\ m\ x)$
proof (*induction* $m - n$ *arbitrary*: n)
case *0*
hence $m = n$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Suc k*)
hence $n < m$ **by** *force*
hence $Suc\ n \leq m$ **by** *auto*
have $\varphi\ n\ (source.t'\ m\ n\ x) = \varphi\ n\ (t_X\ n\ (source.t'\ m\ (Suc\ n)\ x))$
using *source.t'-Suc'* ‹$n < m$› **by** *simp*
also have $\ldots = t_Y\ n\ (\varphi\ (Suc\ n)\ (source.t'\ m\ (Suc\ n)\ x))$
using *source.t'-fun* ‹$Suc\ n \leq m$› *Suc(1)* *Suc(4)* *is-natural-wrt-t* **by** *blast*
also have $\ldots = t_Y\ n\ (target.t'\ m\ (Suc\ n)\ (\varphi\ m\ x))$
using *Suc* ‹$Suc\ n \leq m$› *Suc-diff-Suc Suc-inject* ‹$n < m$› **by** *presburger*

finally show *?case* **using** *target.t′-Suc′* ‹$n < m$› **by** *simp*
qed

end

The composition of two globular maps is itself a globular map. This intermediate locale gathers the data needed for such a statement.

locale *two-globular-maps* = *fst*: *globular-map* X s_X t_X Y s_Y t_Y φ + *snd*: *globular-map* Y s_Y t_Y Z s_Z t_Z ψ
for X s_X t_X Y s_Y t_Y Z s_Z t_Z φ ψ

sublocale *two-globular-maps* ⊆ *comp*: *globular-map* X s_X t_X Z s_Z t_Z $\lambda m.\ \psi\ m \circ \varphi\ m$
proof (*unfold-locales*)
fix m
show $\psi\ m \circ \varphi\ m \in X\ m \to Z\ m$ **using** *fst.map-fun snd.map-fun* **by** *fastforce*
next
fix x m **assume** $x \in X\ (Suc\ m)$
then show $(\psi\ m \circ \varphi\ m)\ (s_X\ m\ x) = s_Z\ m\ ((\psi\ (Suc\ m) \circ \varphi\ (Suc\ m))\ x)$
using *fst.is-natural-wrt-s snd.is-natural-wrt-s comp-apply fst.map-fun* **by** *fastforce*
next
fix x m **assume** $x \in X\ (Suc\ m)$
then show $(\psi\ m \circ \varphi\ m)\ (t_X\ m\ x) = t_Z\ m\ ((\psi\ (Suc\ m) \circ \varphi\ (Suc\ m))\ x)$
using *fst.is-natural-wrt-t snd.is-natural-wrt-t comp-apply fst.map-fun* **by** *fastforce*
qed

sublocale *two-globular-maps* ⊆ *compose*: *globular-map* X s_X t_X Z s_Z t_Z $\lambda m.\ \psi\ m \circ \varphi\ m \downarrow X\ m$
proof (*unfold-locales*)
fix m
show $\psi\ m \circ \varphi\ m \downarrow X\ m \in X\ m \to Z\ m$ **using** *funcset-compose fst.map-fun snd.map-fun* **by** *fast*
next
fix x m **assume** $x \in X\ (Suc\ m)$
then show $(\psi\ m \circ \varphi\ m \downarrow X\ m)\ (s_X\ m\ x) = s_Z\ m\ ((\psi\ (Suc\ m) \circ \varphi\ (Suc\ m) \downarrow X\ (Suc\ m))\ x)$
by (*metis PiE fst.is-natural-wrt-s snd.is-natural-wrt-s fst.map-fun compose-eq fst.source.s-fun*)
next
fix x m **assume** $x \in X\ (Suc\ m)$
then show $(\psi\ m \circ \varphi\ m \downarrow X\ m)\ (t_X\ m\ x) = t_Z\ m\ ((\psi\ (Suc\ m) \circ \varphi\ (Suc\ m) \downarrow X\ (Suc\ m))\ x)$
by (*metis PiE fst.is-natural-wrt-t snd.is-natural-wrt-t fst.map-fun compose-eq fst.source.t-fun*)
qed

2.3 The terminal globular set

The terminal globular set, with a unique m-cell for each m [1, p. 264].

interpretation *final-glob*: *globular-set* $\lambda m.\ \{()\}\ \lambda m.\ id\ \lambda m.\ id$
  by (*unfold-locales*, *auto*)

context *globular-set*
begin

[1, p. 272]

interpretation *map-to-final-glob*: *globular-map* $X\ s\ t$
  $\lambda m.\ \{()\}\ \lambda m.\ id\ \lambda m.\ id$
  $\lambda m.\ (\lambda x.\ ())$
  by (*unfold-locales*, *simp-all*)

end

end
theory *Strict-Omega-Category*
imports *Globular-Set*

begin

3 Strict ω-categories

First, we define a locale *pre-strict-omega-category* that holds the data of a strict ω-category without the associativity, unity and exchange axioms [1, Def 1.4.8 (a) - (b)]. We do this in order to set up convenient notation before we state the remaining axioms.

locale *pre-strict-omega-category* = *globular-set* +
  fixes $comp :: nat \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
    and $i :: nat \Rightarrow 'a \Rightarrow 'a$
  assumes *comp-fun*: $is\text{-}composable\text{-}pair\ m\ n\ x'\ x \Longrightarrow comp\ m\ n\ x'\ x \in X\ m$
    and *i-fun*: $i\ n \in X\ n \to X\ (Suc\ n)$
    and *s-comp-Suc*: $is\text{-}composable\text{-}pair\ (Suc\ m)\ m\ x'\ x \Longrightarrow s\ m\ (comp\ (Suc\ m)\ m\ x'\ x) = s\ m\ x$
    and *t-comp-Suc*: $is\text{-}composable\text{-}pair\ (Suc\ m)\ m\ x'\ x \Longrightarrow t\ m\ (comp\ (Suc\ m)\ m\ x'\ x) = t\ m\ x'$
    and *s-comp*: $[\![is\text{-}composable\text{-}pair\ (Suc\ m)\ n\ x'\ x;\ n < m]\!] \Longrightarrow$
        $s\ m\ (comp\ (Suc\ m)\ n\ x'\ x) = comp\ m\ n\ (s\ m\ x')\ (s\ m\ x)$
    and *t-comp*: $[\![is\text{-}composable\text{-}pair\ (Suc\ m)\ n\ x'\ x;\ n < m]\!] \Longrightarrow$
        $t\ m\ (comp\ (Suc\ m)\ n\ x'\ x) = comp\ m\ n\ (t\ m\ x')\ (s\ m\ x)$
    and *s-i*: $x \in X\ n \Longrightarrow s\ n\ (i\ n\ x) = x$
    and *t-i*: $x \in X\ n \Longrightarrow t\ n\ (i\ n\ x) = x$
begin

Similar to the generalised source and target maps in *globular-set*, we defined a generalised identity map. The first argument gives the dimension of the resulting identity cell, while the second gives the dimension of the input cell.

```
fun i′ :: nat ⇒ nat ⇒ ′a ⇒ ′a where
i′ 0 0 = id |
i′ 0 (Suc n) = undefined |
i′ (Suc m) n = (if Suc m < n then undefined
  else if Suc m = n then id
  else i m ∘ i′ m n)

lemma i′-n-n [simp]: i′ n n = id
  by (metis i′.elims i′.simps(1) less-irrefl-nat)

lemma i′-Suc-n-n [simp]: i′ (Suc n) n = i n
  by simp

lemma i′-Suc [simp]: n ≤ m ⟹ i′ (Suc m) n = i m ∘ i′ m n
  by fastforce

lemma i′-Suc′: n < m ⟹ i′ m n = i′ m (Suc n) ∘ i n
proof (induction m arbitrary: n)
  case 0
  then show ?case by blast
next
  case (Suc m)
  then show ?case by force
qed

lemma i′-fun: n ≤ m ⟹ i′ m n ∈ X n → X m
proof (induction m arbitrary: n)
  case 0
  then show ?case by fastforce
next
  case (Suc m)
  thus ?case proof (cases n = Suc m)
    case True
    then show ?thesis by auto
  next
    case False
    hence n ≤ m using ‹n ≤ Suc m› by force
    thus ?thesis using Suc.IH i-fun by auto
  qed
qed

end
```

Now we may define a strict ω-category including the composition, unity and exchange axioms [1, Def 1.4.8 (c) - (f)].

```
locale strict-omega-category = pre-strict-omega-category +
  assumes comp-assoc: ⟦is-composable-pair m n x′ x; is-composable-pair m n x″
```

x⟧ $\Longrightarrow$
$comp\ m\ n\ (comp\ m\ n\ x''\ x')\ x = comp\ m\ n\ x''\ (comp\ m\ n\ x'\ x)$
and *i-comp*: $[\![n < m;\ x \in X\ m]\!] \Longrightarrow comp\ m\ n\ (i'\ m\ n\ (t'\ m\ n\ x))\ x = x$
and *comp-i*: $[\![n < m;\ x \in X\ m]\!] \Longrightarrow comp\ m\ n\ x\ (i'\ m\ n\ (s'\ m\ n\ x)) = x$
and *bin-interchange*: $[\![q < p;\ p < m;$
is-composable-pair $m\ p\ y'\ y$; *is-composable-pair* $m\ p\ x'\ x$;
is-composable-pair $m\ q\ y'\ x'$; *is-composable-pair* $m\ q\ y\ x]\!] \Longrightarrow$
$comp\ m\ q\ (comp\ m\ p\ y'\ y)\ (comp\ m\ p\ x'\ x) = comp\ m\ p\ (comp\ m\ q\ y'\ x')$
$(comp\ m\ q\ y\ x)$
and *null-interchange*: $[\![q < p$; *is-composable-pair* $p\ q\ x'\ x]\!] \Longrightarrow$
$comp\ (Suc\ p)\ q\ (i\ p\ x')\ (i\ p\ x) = i\ p\ (comp\ p\ q\ x'\ x)$

locale *strict-omega-functor* = *globular-map* +
source: *strict-omega-category* $X\ s_X\ t_X\ comp_X\ i_X$ +
target: *strict-omega-category* $Y\ s_Y\ t_Y\ comp_Y\ i_Y$
for $comp_X\ i_X\ comp_Y\ i_Y$ +
assumes *commute-with-comp*: *is-composable-pair* $m\ n\ x'\ x \Longrightarrow$
$\varphi\ m\ (comp_X\ m\ n\ x'\ x) = comp_Y\ m\ n\ (\varphi\ m\ x')\ (\varphi\ m\ x)$
and *commute-with-id*: $x \in X\ n \Longrightarrow \varphi\ (Suc\ n)\ (i_X\ n\ x) = i_Y\ n\ (\varphi\ n\ x)$

end
theory *Pasting-Diagram*
imports *Strict-Omega-Category*

begin

4 The category of pasting diagrams

We define the strict ω-category of pasting diagrams, 'pd'. We encode its cells as rooted trees. First we develop some basic theory of trees.

4.1 Rooted trees

datatype *tree* = *Node* (*subtrees*: *tree list*) — [1, p. 268]

abbreviation *Leaf* :: *tree* **where**
Leaf ≡ *Node* []

fun *subtree* :: *tree* ⇒ *nat list* ⇒ *tree* (‹- !t -› [59,60]59) **where**
t !*t* [] = *t* |
t !*t* (*i*#*xs*) = *subtrees* (*t* !*t* *xs*) ! *i*

value *Leaf* !*t* []
value *Node* [*Node* [*Leaf*, *Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]] !*t* [0]
value *Node* [*Node* [*Leaf*, *Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]] !*t* [2,0]
value *Node* [*Node* [*Leaf*, *Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]] !*t* [1]
value *Node* [*Node* [*Leaf*, *Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]] !*t* [0,2]

lemma *subtrees-Leaf*: $(t = \mathit{Leaf}) = (\mathit{subtrees}\ t = [])$
  by $(\mathit{metis\ tree.collapse\ tree.sel})$

fun *is-subtree-index* $:: \mathit{tree} \Rightarrow \mathit{nat\ list} \Rightarrow\ \mathit{bool}$ **where**
$\mathit{is\text{-}subtree\text{-}index}\ t\ [] = \mathit{True}\ |$
$\mathit{is\text{-}subtree\text{-}index}\ t\ (i\#xs) = (\mathit{is\text{-}subtree\text{-}index}\ t\ xs \wedge i < \mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs)))$

lemma *subtree-append*: $ts\ !\ i\ !t\ xs = \mathit{Node}\ ts\ !t\ xs\ @\ [i]$
  by $(\mathit{induction}\ xs,\ \mathit{auto})$

lemma *is-subtree-index-append* $[\mathit{iff}]$: $\mathit{is\text{-}subtree\text{-}index}\ (\mathit{Node}\ ts)\ (xs\ @\ [i]) =$
  $(i < \mathit{length}\ ts \wedge \mathit{is\text{-}subtree\text{-}index}\ (ts!i)\ xs)$
proof
  show $\mathit{is\text{-}subtree\text{-}index}\ (\mathit{Node}\ ts)\ (xs\ @\ [i]) \Longrightarrow i < \mathit{length}\ ts \wedge \mathit{is\text{-}subtree\text{-}index}$
$(ts\ !\ i)\ xs$
  by $(\mathit{induction}\ xs,\ \mathit{auto\ simp}: \mathit{subtree\text{-}append})$
next
  show $i < \mathit{length}\ ts \wedge \mathit{is\text{-}subtree\text{-}index}\ (ts\ !\ i)\ xs \Longrightarrow \mathit{is\text{-}subtree\text{-}index}\ (\mathit{Node}\ ts)$
$(xs\ @\ [i])$
  by $(\mathit{induction}\ xs,\ \mathit{auto\ simp}: \mathit{subtree\text{-}append})$
qed

lemma *is-subtree-index-append′* $[\mathit{iff}]$: $\mathit{is\text{-}subtree\text{-}index}\ t\ (xs\ @\ [i]) =$
  $(\mathit{is\text{-}subtree\text{-}index}\ t\ [i] \wedge \mathit{is\text{-}subtree\text{-}index}\ (t\ !t\ [i])\ xs)$
 by $(\mathit{metis\ is\text{-}subtree\text{-}index\text{-}append\ is\text{-}subtree\text{-}index.simps\ subtree.simps\ tree.collapse})$

lemma *max-set-upt* $[\mathit{simp}]$: $\mathit{Max}\ \{0..<\mathit{Suc}\ n\} = n$
  by $(\mathit{simp\ add}: \mathit{Max\text{-}eq\text{-}iff})$

lemma *length-subtrees-eq-Max*: **assumes** $\mathit{is\text{-}subtree\text{-}index}\ t\ xs\ \mathit{subtrees}\ (t\ !t\ xs) \neq$
$[]$
  shows $\mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs)) = \mathit{Suc}\ (\mathit{Max}\ \{i.\ \mathit{is\text{-}subtree\text{-}index}\ t\ (i\ \#\ xs)\})$
proof −
  have $\bigwedge i.\ \mathit{is\text{-}subtree\text{-}index}\ t\ (i\ \#\ xs) = (i < \mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs)))$ **using**
$\mathit{assms}(1)$ **by** *simp*
  hence $\{i.\ \mathit{is\text{-}subtree\text{-}index}\ t\ (i\ \#\ xs)\} = \{0..<\mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs))\}$ **by**
fastforce
  moreover have $\mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs)) > 0$ **using** $\mathit{assms}(2)$ **by** *simp*
  ultimately show $\mathit{length}\ (\mathit{subtrees}\ (t\ !t\ xs)) = \mathit{Suc}\ (\mathit{Max}\ \{i.\ \mathit{is\text{-}subtree\text{-}index}\ t\ (i$
$\#\ xs)\})$
    by $(\mathit{metis\ max\text{-}set\text{-}upt\ gr0\text{-}implies\text{-}Suc})$
qed

lemma *tree-eq-iff-subtree-eq*: $(t = u) = (\mathit{length}\ (\mathit{subtrees}\ t) = \mathit{length}\ (\mathit{subtrees}\ u)$
$\wedge$
  $(\forall i < \mathit{length}\ (\mathit{subtrees}\ t).\ t\ !t\ [i] = u\ !t\ [i]))$
  by $(\mathit{cases}\ t,\ \mathit{cases}\ u,\ \mathit{auto\ simp\ add}: \mathit{list\text{-}eq\text{-}iff\text{-}nth\text{-}eq})$

We define the height of a rooted tree. A tree with only one node has height 0. The trees of height at most n encode the n-cells in 'pd'.

fun *height* :: *tree* ⇒ *nat* **where**
height Leaf = *0* |
height (*Node ts*) = *Suc* (*fold* (*max* ∘ *height*) *ts 0*)

value *height Leaf*
value *height* (*Node* [*Leaf*, *Leaf*])
value *height* (*Node* [*Node* [*Leaf*, *Leaf*], *Leaf*])
value *height* (*Node* [*Node* [*Leaf*, *Node* [*Leaf*]]])

lemma *height-Node* [*simp*]: *ts* ≠ [] ⟹ *height* (*Node ts*) = *Suc* (*fold* (*max* ∘ *height*) *ts 0*)
  by (*metis height.simps*(*2*) *neq-Nil-conv*)

lemma *fold-eq-Max* [*simp*]: *ts* ≠ [] ⟹ *fold* (*max* ∘ *height*) *ts 0* = *Max* (*set* (*map height ts*))
  using *Max.set-eq-fold fold-map list.exhaust*
  by (*metis* (*no-types*, *lifting*) *fold-simps*(*2*) *map-is-Nil-conv max-nat.right-neutral*)

lemma *height-Node-Max*: *ts* ≠ [] ⟹ *height* (*Node ts*) = *Suc* (*Max* (*set* (*map height ts*)))
  by *simp*

lemma *height-Node-pos* : *ts* ≠ [] ⟹ *0* < *height* (*Node ts*)
proof (*induction Node ts rule*: *height.induct*)
  case *1*
  then show *?case* **by** *blast*
next
  case (*2 t ts′*)
  then show *?case* **by** *fastforce*
qed

lemma *height-exists*:
  assumes *height* (*Node ts*) = *Suc n*
  shows ∃ *t*. *t* ∈ *set ts* ∧ *height t* = *n*
proof (*cases ts* = [])
  case *True*
  then show *?thesis* **using** *assms* **by** *simp*
next
  case *False*
  hence *n* = *Max* (*set* (*map height ts*)) **using** *assms height-Node-Max* **by** *force*
  hence *n* ∈ *set* (*map height ts*) **using** *Max-in* ‹*ts* ≠ []› **by** *auto*
  then show *?thesis* **by** *auto*
qed

lemma *height-lt*: **assumes** *t* ∈ *set ts* **shows** *height t* < *height* (*Node ts*)
proof −
  from *assms* **have** *nemp*: *ts* ≠ [] **by** *fastforce*
  have *height t* ≤ *Max* (*set* (*map height ts*)) **using** *assms* **by** *fastforce*
  also have . . . = *fold* (*max* ∘ *height*) *ts 0* **using** *nemp fold-eq-Max* **by** *simp*

finally show *?thesis* **using** *nemp* **by** *simp*
qed

lemma *height-le-imp-le-Suc*:
  assumes $\forall t \in set\ ts.\ height\ t \leq n$
  shows $height\ (Node\ ts) \leq Suc\ n$
proof $(cases\ ts = [])$
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  hence $height\ (Node\ ts) = Suc\ (Max\ (set\ (map\ height\ ts)))$ **using** *height-Node-Max* **by** *blast*
  also have $\ldots \leq Suc\ (Max\ (height\ `\ set\ ts))$ **using** *set-map* **by** *fastforce*
  finally show *?thesis* **using** $\langle ts \neq [] \rangle$ *assms* **by** *simp*
qed

lemma *height-zero* [*simp*]: $height\ t = 0 \Longrightarrow t = Leaf$
  by (*metis height.cases height-Node-pos less-nat-zero-code*)

lemma *is-subtree-index-length-le*: $is\text{-}subtree\text{-}index\ t\ xs \Longrightarrow length\ xs \leq height\ t$
proof (*induction xs arbitrary*: *t rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *force*
next
  case (*snoc i xs*)
  hence *hi*: $i < length\ (subtrees\ t)$ **by** (*metis is-subtree-index-append tree.exhaust-sel*)
  hence $length\ xs \leq height\ (subtrees\ t\ !\ i)$
    by (*metis snoc is-subtree-index-append tree.exhaust-sel*)
  moreover have $subtrees\ t\ !\ i \in set\ (subtrees\ t)$ **using** *hi* **by** *simp*
  ultimately show *?case* **using** *height-lt* **by** *fastforce*
qed

lemma *height-subtree*: $is\text{-}subtree\text{-}index\ t\ xs \Longrightarrow height\ (t\ !t\ xs) \leq height\ t - length\ xs$
proof (*induction xs arbitrary*: *t rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*snoc i xs*)
  hence $is\text{-}subtree\text{-}index\ (t\ !t\ [i])\ xs$ **using** *is-subtree-index-append'* **by** *fastforce*
  hence $height\ (t\ !t\ [i]\ !t\ xs) \leq height\ (t\ !t\ [i]) - length\ xs$ **using** *snoc.IH* **by** *blast*
  moreover have $height\ (t\ !t\ [i]) < height\ t$
    by (*metis height-lt is-subtree-index.simps*(*2*) *is-subtree-index-append' nth-mem snoc.prems*
      subtree.simps tree.collapse)
  moreover have $t\ !t\ [i]\ !t\ xs = t\ !t\ xs\ @\ [i]$ **using** *subtree-append* **by** *simp*
  ultimately show *?case* **by** *auto*
qed

lemma *height-induct*: $(\bigwedge t.\ \forall u.\ height\ u < height\ t \longrightarrow P\ u \Longrightarrow P\ t) \Longrightarrow P\ t$
  by (*metis Nat.measure-induct*)

lemma *subtree-index-induct* [*case-names Index Step*]:
  assumes
    is-subtree-index t xs
    $\bigwedge xs.\ [\![is\text{-}subtree\text{-}index\ t\ xs;\ \forall i < length\ (subtrees\ (t\ !t\ xs)).\ P\ (i\#xs)]\!] \Longrightarrow P\ xs$
  shows *P xs*
proof −
  have *hl*: *length xs* ≤ *height t* **by** (*simp add*: *assms*(*1*) *is-subtree-index-length-le*)
  then show *P xs* **using** *assms*
  proof (*induction height t* − *length xs arbitrary*: *xs*)
    case *0*
    hence *height* (*t* !*t xs*) = *0* **using** *height-subtree* **by** *fastforce*
    hence ∀ *i* < *length* (*subtrees* (*t* !*t xs*)). *P* (*i* # *xs*)
      by (*metis height-zero length-0-conv less-nat-zero-code tree.sel*)
    then show *?case* **using** *0.prems* **by** *blast*
  next
    case (*Suc n*)
    have ∀ *i* < *length* (*subtrees* (*t* !*t xs*)). *P* (*i* # *xs*)
    proof (*safe*)
      fix *i* **assume** *i* < *length* (*subtrees* (*t* !*t xs*))
      hence *is-subtree-index t* (*i* # *xs*) **using** *Suc*(*4*) **by** *simp*
      moreover hence *length* (*i* # *xs*) ≤ *height t* **using** *is-subtree-index-length-le*
by *blast*
      moreover have *n* = *height t* − *length* (*i* # *xs*) **using** *Suc*(*2*) **by** *simp*
      ultimately show *P* (*i* # *xs*) **using** *Suc*(*1*) *Suc*(*5*) **by** *blast*
    qed
    then show *?case* **using** *Suc.prems* **by** *blast*
  qed
qed

The function *trim* keeps the first n layers of a tree and removes the remaining ones.

fun *trim* :: *nat* ⇒ *tree* ⇒ *tree* **where**
trim 0 t = *Leaf* |
trim (*Suc n*) (*Node ts*) = *Node* (*map* (*trim n*) *ts*)

lemma *trim-Leaf* [*simp*]: *trim n Leaf* = *Leaf*
  by (*metis list.simps*(*8*) *trim.elims trim.simps*(*2*))

lemma *height-trim-le*: *height* (*trim n t*) ≤ *n*
proof (*induction n t rule*: *trim.induct*)
  case (*1 t*)
  then show *?case* **by** *auto*
next
  case (*2 n ts*)
  hence $\forall t' \in set\ (map\ (trim\ n)\ ts).\ height\ t' \le n$ **by** *auto*

then show *?case* **using** *height-le-imp-le-Suc trim.simps*(*2*) **by** *presburger*
qed

lemma *trim-const*: $height\ t \leq n \Longrightarrow trim\ n\ t = t$
proof (*induction n t rule*: *trim.induct*)
case (*1 t*)
then show *?case* **using** *height-zero trim-Leaf* **by** *blast*
next
case (*2 n ts*)
hence $\bigwedge t.\ t \in set\ ts \Longrightarrow trim\ n\ t = t$ **using** *height-lt* **by** *fastforce*
hence *map* (*trim n*) *ts* = *ts* **using** *map-idI* **by** *blast*
then show *?case* **by** *fastforce*
qed

lemma *height-trim-le'*: $n \leq height\ t \Longrightarrow height\ (trim\ n\ t) = n$
proof (*induction n t rule*: *trim.induct*)
case (*1 t*)
then show *?case* **by** *fastforce*
next
case (*2 n ts*)
hence $\exists\, m.\ height\ (Node\ ts) = Suc\ m$ **by** *presburger*
then obtain *m* **where** *hm*: $height\ (Node\ ts) = Suc\ m$ **by** *presburger*
then obtain *t* **where** *ht*: $t \in set\ ts \land height\ t = m$ **using** *height-exists* **by** *meson*
have $n \leq m$ **using** *2 hm* **by** *fastforce*
hence *hn*: $height\ (trim\ n\ t) = n$ **using** *2 ht* **by** *blast*
have $trim\ n\ t \in set\ (subtrees\ (trim\ (Suc\ n)\ (Node\ ts)))$ **using** *ht* **by** *simp*
then show *?case* **using** *hn height-lt* **by** (*metis height-trim-le leD le-SucE tree.collapse*)
qed

lemma *height-trim*: $height\ (trim\ n\ t) = (if\ n \leq height\ t\ then\ n\ else\ height\ t)$
using *height-trim-le' trim-const* **by** *auto*

value *trim 1 Leaf*
value *trim 1* (*Node* [*Leaf*, *Leaf*])
value *trim 2* (*Node* [*Node* [*Leaf*, *Leaf*], *Leaf*])
value *trim 1* (*Node* [*Node* [*Leaf*, *Node* [*Leaf*]], *Node* [*Leaf*]])

lemma *trim-trim'* [*simp*]: $trim\ n \circ trim\ n = trim\ n$
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **apply** (*simp add*: *fun-eq-iff*) **proof**
fix *t*
show $trim\ (Suc\ n)\ (trim\ (Suc\ n)\ t) = trim\ (Suc\ n)\ t$
using *Suc* **by** (*metis list.map-comp tree.exhaust trim.simps*(*2*))
qed
qed

lemma *trim-trim-Suc* [*simp*]: *trim n* ∘ *trim* (*Suc n*) = *trim n*
proof (*induction n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then show *?case* **apply** (*simp add*: *fun-eq-iff*) **proof**
    fix *t*
    show *trim* (*Suc n*) (*trim* (*Suc* (*Suc n*)) *t*) = *trim* (*Suc n*) *t*
      using *Suc* **by** (*metis list.map-comp tree.exhaust trim.simps*(*2*))
  qed
qed

lemma *trim-trim* [*simp*]: $n \leq m \Longrightarrow$ *trim n* ∘ *trim m* = *trim n*
proof (*induction m arbitrary*: *n*)
  case *0*
  then show *?case* **by** *force*
next
  case (*Suc m*)
  then show *?case* **proof** (*cases n* = *Suc m*)
    case *True*
    then show *?thesis* **by** *auto*
  next
    case *False*
    hence $n \leq m$ **using** *Suc.prems* **by** *auto*
    hence *ih*: *trim n* = *trim n* ∘ *trim m* **using** *Suc* **by** *presburger*
    hence *trim n* ∘ *trim* (*Suc m*) = (*trim n* ∘ *trim m*) ∘ *trim* (*Suc m*) **by** *simp*
    also have ... = *trim n* ∘ *trim m* **by** (*metis fun.map-comp trim-trim-Suc*)
    finally show *?thesis* **using** *ih* **by** *auto*
  qed
qed

lemma *trim-eq-imp-trim-eq* [*simp*]: ⟦$n \leq m$; *trim m t* = *trim m u*⟧ $\Longrightarrow$ *trim n t* = *trim n u*
  by (*metis trim-trim comp-apply*)

lemma *trim-1-eq*: **assumes** *trim 1* (*Node ts*) = *trim 1* (*Node us*) **shows** *length ts* = *length us*
proof −
  have ⋀*vs*. *trim 1* (*Node vs*) = *Node* (*map* (λ*x*. *Leaf*) *vs*) **by** *force*
  then show *?thesis* **using** *assms map-eq-imp-length-eq* **by** *auto*
qed

lemma *length-subtrees-trim-Suc*: *length* (*subtrees* (*trim* (*Suc n*) *t*)) = *length* (*subtrees* *t*)
  by (*induction t*, *simp*)

lemma *trim-eq-Leaf*: *trim n t* = *Leaf* $\Longrightarrow$ *n* = *0* ∨ *t* = *Leaf*

by (*induction n t rule*: *trim.induct*, *simp-all*)

lemma *map-eq-imp-pairs-eq*: *map f xs* = *map g ys* $\Longrightarrow$ ($\bigwedge$*x y*. (*x*, *y*) $\in$ *set* (*zip xs ys*) $\Longrightarrow$ *f x* = *g y*)
by (*metis fst-eqD in-set-zip nth-map snd-eqD*)

lemma *trim-eq-subtree-eq*:
assumes *trim* (*Suc n*) (*Node ts*) = *trim* (*Suc n*) (*Node us*)
shows $\bigwedge$*t u*. (*t*, *u*) $\in$ *set* (*zip ts us*) $\Longrightarrow$ *trim n t* = *trim n u*
proof −
fix *t u* **assume** (*t*, *u*) $\in$ *set* (*zip ts us*)
moreover from *assms* **have** *map* (*trim n*) *ts* = *map* (*trim n*) *us* **by** *fastforce*
ultimately show *trim n t* = *trim n u* **using** *map-eq-imp-pairs-eq* **by** *fast*
qed

lemma *pairs-eq-imp-map-eq*:
assumes *length xs* = *length ys* $\forall$(*x*, *y*) $\in$ *set* (*zip xs ys*). *f x* = *g y*
shows *map f xs* = *map g ys*
proof −
have $\bigwedge$*x y*. (*x*, *y*) $\in$ *set* (*zip* (*map f xs*) (*map g ys*)) $\Longrightarrow$ *x* = *y* **proof** −
fix *x y* **assume** *h*: (*x*, *y*) $\in$ *set* (*zip* (*map f xs*) (*map g ys*))
hence $\exists$*n*. (*map f xs*)!*n* = *x* $\wedge$ (*map g ys*)!*n* = *y* $\wedge$ *n* < *length xs* $\wedge$ *n* < *length ys*
by (*metis in-set-zip fst-conv length-map snd-conv*)
then obtain *n* **where** *hn*: (*map f xs*)!*n* = *x* (*map g ys*)!*n* = *y* *n* < *length xs* *n* < *length ys*
by *blast*
hence (*xs*!*n*, *ys*!*n*) $\in$ *set* (*zip xs ys*) **using** *in-set-zip* **by** *fastforce*
with *hn assms*(*2*) **show** *x* = *y* **by** *auto*
qed
hence $\forall$(*x*, *y*) $\in$ *set* (*zip* (*map f xs*) (*map g ys*)). *x* = *y* **by** *force*
with *assms*(*1*) *list-eq-iff-zip-eq* **show** *map f xs* = *map g ys* **by** *fastforce*
qed

lemma *map-eq-iff-pairs-eq*: (*map f xs* = *map g ys*) =
(*length xs* = *length ys* $\wedge$ ($\forall$(*x*, *y*) $\in$ *set* (*zip xs ys*). *f x* = *g y*))
proof −
have *map f xs* = *map g ys* $\Longrightarrow$ $\forall$(*x*, *y*) $\in$ *set* (*zip xs ys*). *f x* = *g y* **using** *map-eq-imp-pairs-eq*
by *fast*
thus *?thesis* **by** (*metis pairs-eq-imp-map-eq length-map*)
qed

lemma *subtree-eq-trim-eq*:
assumes *length ts* = *length us* $\forall$(*t*, *u*) $\in$ *set* (*zip ts us*). *trim n t* = *trim n u*
shows *trim* (*Suc n*) (*Node ts*) = *trim* (*Suc n*) (*Node us*)
by (*auto simp add*: *assms map-eq-iff-pairs-eq*)

lemma *subtree-trim-1*: *is-subtree-index t* [*i*] $\Longrightarrow$ *trim* (*Suc n*) *t* !*t* [*i*] = *trim n* (*t*

!*t* [*i*])
by (*smt* (*verit*) *Suc-inject is-subtree-index.simps*(*2*) *list.distinct*(*1*) *nat.distinct*(*1*) *nth-map*
subtree.elims subtree.simps(*2*) *tree.sel trim.elims*)

lemma *is-subtree-index-trim*:
is-subtree-index (*trim n t*) *xs* = (*is-subtree-index t xs* ∧ *length xs* ≤ *n*)
proof (*induction n t arbitrary*: *xs rule*: *trim.induct*)
case (*1 t*)
then show *?case* **using** *is-subtree-index-length-le* **by** *fastforce*
next
case (*2 n ts*)
then show *?case* **proof** (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case* **by** *fastforce*
qed
qed

lemma *subtree-trim*: ⟦*is-subtree-index t xs*; *length xs* ≤ *n*⟧ ⟹
trim n t !*t xs* = *trim* (*n* − *length xs*) (*t* !*t xs*)
proof (*induction n t arbitrary*: *xs rule*: *trim.induct*)
case (*1 t*)
then show *?case* **by** *simp*
next
case (*2 n ts*)
then show *?case* **proof** (*cases length xs* = *Suc n*)
case *True*
hence *is-subtree-index* (*trim* (*Suc n*) (*Node ts*)) *xs* **using** *is-subtree-index-trim* *2* **by** *blast*
hence *height* (*trim* (*Suc n*) (*Node ts*) !*t xs*) ≤ *0*
by (*metis height-subtree height-trim-le True diff-is-0-eq′*)
then show *?thesis* **using** *True height-zero* **by** *fastforce*
next
case *False*
then show *?thesis* **proof** (*cases xs rule*: *rev-cases*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*snoc ys i*)
have *hi*: *ts* ! *i* ∈ *set ts is-subtree-index* (*ts* ! *i*) *ys* **using** *snoc 2*(*2*) **by** *simp-all*
have *hl*: *length ys* ≤ *n* **using** *snoc 2*(*3*) **by** *simp*
have *Node* (*map* (*trim n*) *ts*) !*t ys* @ [*i*] = *trim n* (*ts* ! *i*) !*t ys*
by (*metis 2.prems*(*1*) *is-subtree-index-append nth-map snoc subtree-append*)
also have . . . = *trim* (*n* − *length ys*) (*ts* ! *i* !*t ys*) **using** *2*(*1*) *hi hl* **by** *blast*
finally show *trim* (*Suc n*) (*Node ts*) !*t xs* = *trim* (*Suc n* − *length xs*) (*Node ts* !*t xs*)

by (*simp add*: *snoc subtree-append*)
qed
qed
qed

lemma *length-subtrees-trim*: $[\![$*is-subtree-index t xs*; *length xs* $<$ *n*$]\!] \Longrightarrow$
length (*subtrees* (*trim n t* !*t xs*)) = *length* (*subtrees* (*t* !*t xs*))
by (*metis subtree-trim length-subtrees-trim-Suc Suc-diff-Suc less-imp-le-nat*)

lemma *subtree-trim-Leaf*: **assumes** *is-subtree-index* (*trim n t*) *xs t* !*t xs* = *Leaf*
shows *trim n t* !*t xs* = *Leaf*
proof (*cases length xs* $<$ *n*)
case *True*
then show *?thesis*
using *length-subtrees-trim assms is-subtree-index-trim subtrees-Leaf* **by** *fastforce*
next
case *False*
hence *length xs* = *n* **using** *assms*(*1*) **by** (*simp add*: *is-subtree-index-trim*)
then show *?thesis* **using** *assms*(*1*) *is-subtree-index-trim subtree-trim* **by** *auto*
qed

4.2 The strict ω-category of pasting diagrams

The function δ acts as both the source and target map in the globular set of pasting diagrams. It is denoted ∂ in Leinster [1, p. 264].

abbreviation δ **where**
$\delta \equiv$ *trim*

value δ *1* (*Node* [*Node* [*Leaf*, *Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]])
value δ *2* (*Node* [*Node* [*Node* [*Leaf*, *Leaf*]], *Node* [*Leaf*, *Leaf*]])

abbreviation *PD* :: *nat* $\Rightarrow$ *tree set* **where**
PD n $\equiv$ {*t*. *height t* $\leq$ *n*}

interpretation *pd*: *globular-set PD* δ δ
by (*unfold-locales*, *auto simp add*: *height-trim-le*)

The generalised source and target maps have simple interpretations in terms of *trim*.

lemma *s′-eq-trim*: **assumes** *n* $\leq$ *m height t* $\leq$ *m* **shows** *pd.s′ m n t* = *trim n t*
using *assms*
proof (*induction m arbitrary*: *t*)
case *0*
moreover hence *n* = *0* **by** *force*
ultimately show *?case* **using** *pd.s′-n-n trim-const* **by** *simp*
next
case (*Suc m*)
then show *?case* **proof** (*cases n* = *Suc m*)
case *True*

then show *?thesis* **using** *pd.s′-n-n Suc*(*3*) *trim-const* **by** *simp*
next
case *False*
with *Suc*(*2*) **have** $n \leq m$ **by** *simp*
hence *pd.s′* (*Suc m*) *n t* = *pd.s′ m n* (δ *m t*) **using** *Suc*(*3*) **by** *force*
also have ... = δ *n* (δ *m t*) **using** *Suc.IH height-trim-le* ‹$n \leq m$› **by** *blast*
finally show *?thesis* **by** (*metis trim-trim* ‹$n \leq m$› *comp-apply*)
qed
qed

lemma *s′-eq-t′*: *pd.s′* = *pd.t′*
proof (*clarsimp simp add*: *fun-eq-iff*)
fix *m n t*
show *pd.s′ m n t* = *pd.t′ m n t* **proof** (*induction m arbitrary*: *n t*)
case *0*
then show *?case*
using *pd.s′-n-n pd.t′-n-n pd.s′.simps*(*2*) *pd.t′.simps*(*2*) **by** (*cases n*, *presburger+*)
next
case (*Suc m*)
then show *?case* **by** (*cases Suc m rule*: *linorder-cases*, *simp-all*)
qed
qed

lemma *t′-eq-trim*: **assumes** $n \leq m$ *height t* $\leq m$ **shows** *pd.t′ m n t* = *trim n t*
by (*metis* (*mono-tags*, *lifting*) *assms s′-eq-trim s′-eq-t′*)

Next we define identities and composition [1, p. 266]. The identity of a tree with height at most n is the same tree seen as a tree of height at most $n + 1$.

fun *tree-comp* :: *nat* ⇒ *tree* ⇒ *tree* ⇒ *tree* **where**
tree-comp 0 (*Node ts*) (*Node us*) = *Node* (*ts* @ *us*) |
tree-comp (*Suc n*) (*Node ts*) (*Node us*) = *Node* (*map2* (*tree-comp n*) *ts us*)

value *tree-comp 1*
(*Node* [*Node* [*Leaf*, *Leaf*], *Leaf*, *Node* [*Leaf*]])
(*Node* [*Leaf*, *Leaf*, *Node* [*Leaf*, *Leaf*]])

value *tree-comp 0*
(*Node* [*Node* [*Node* [*Leaf*, *Leaf*]]])
(*Node* [*Node* [*Leaf*, *Leaf*]])

value *tree-comp 0*
(*tree-comp 0*
(*tree-comp 1*

    (*Node* [*Leaf*, *Leaf*])
    (*Node* [*Node* [*Leaf*], *Node* [*Leaf*, *Leaf*, *Leaf*]]))
  (*Node* [*Leaf*, *Node* [*Leaf*, *Leaf*]]))
 (*Node* [*Leaf*, *Leaf*, *Leaf*])

lemma *tree-comp-0-Leaf1* [*simp*]: *tree-comp 0 Leaf t* = *t*
  by (*metis eq-Nil-appendI tree.exhaust tree-comp.simps*(*1*))

lemma *tree-comp-0-Leaf2* [*simp*]: *tree-comp 0 t Leaf* = *t*
  by (*metis append-Nil2 tree.exhaust tree-comp.simps*(*1*))

lemma *tree-comp-Suc-Leaf1* [*simp*]: *tree-comp* (*Suc n*) *Leaf t* = *Leaf*
  by (*cases t*, *simp*)

lemma *tree-comp-Suc-Leaf2* [*simp*]: *tree-comp* (*Suc n*) *t Leaf* = *Leaf*
  by (*cases t*, *simp*)

lemma *height-tree-comp-0* [*simp*]: *height* (*tree-comp 0 t u*) = *max* (*height t*) (*height u*)
proof (*cases t* = *Leaf* ∨ *u* = *Leaf*)
  case *True*
  then show *?thesis* **by** *auto*
next
  case *False*
  hence *nempt*: *subtrees t* ≠ [] ∧ *subtrees u* ≠ [] **by** (*metis tree.exhaust-sel*)
  have *height* (*tree-comp 0 t u*) = *height* (*Node* (*subtrees t* @ *subtrees u*))
    by (*metis tree.collapse tree-comp.simps*(*1*))
  also have ... = *Suc* (*Max* (*set* (*map height* (*subtrees t* @ *subtrees u*))))
    using *nempt height-Node-Max* **by** *blast*
  also have ... = *Suc* (*Max* (*set* (*map height* (*subtrees t*)) ∪ *set* (*map height* (*subtrees u*))))
    by *simp*
  also have ... = *Suc* (*max* (*Max* (*set* (*map height* (*subtrees t*))))
                (*Max* (*set* (*map height* (*subtrees u*)))))
    using *nempt Max-Un* **by** (*metis List.finite-set map-is-Nil-conv set-empty2*)
  also have ... = *max* (*Suc* (*Max* (*set* (*map height* (*subtrees t*)))))
              (*Suc* (*Max* (*set* (*map height* (*subtrees u*)))))
    by *linarith*
  finally show *height* (*tree-comp 0 t u*) = *max* (*height t*) (*height u*)
    using *nempt height-Node-Max* **by** (*metis tree.collapse*)
qed

An alternative description of being composable for trees. Defined so that *tree-comp n t u* is defined if and only if *composable-tree n t u*.

fun *composable-tree* :: *nat* ⇒ *tree* ⇒ *tree* ⇒ *bool* **where**
composable-tree 0 (*Node ts*) (*Node us*) = *True* |
composable-tree (*Suc n*) (*Node ts*) (*Node us*) = (*length ts* = *length us* ∧

$(\forall\, i < \textit{length ts. composable-tree n } (ts!i) \ (us!i)))$

lemma *sym-composable-tree*: *composable-tree n t u* = *composable-tree n u t*
by (*induction n t u rule*: *composable-tree.induct*, *simp*, *fastforce*)

lemma *is-composable-pair-imp-composable-tree*: *pd.is-composable-pair m n t u* $\Longrightarrow$ *composable-tree n t u*
proof (*induction n t u rule*: *composable-tree.induct*)
case (*1 ts us*)
then show *?case* **by** *fastforce*
next
case (*2 n ts us*)
with *pd.is-composable-pair-def* **have** *h*: *Suc n* < *m height* (*Node ts*) $\leq$ *m height* (*Node us*) $\leq$ *m*
pd.t′ m (*Suc n*) (*Node us*) = *pd.s′ m* (*Suc n*) (*Node ts*) **by** *blast+*
moreover hence *Suc n* $\leq$ *m* **by** *linarith*
ultimately have *htrim*: *trim* (*Suc n*) (*Node ts*) = *trim* (*Suc n*) (*Node us*)
by (*metis* (*mono-tags*, *lifting*) *s′-eq-trim t′-eq-trim*)
hence *trim 1* (*Node ts*) = *trim 1* (*Node us*)
by (*metis One-nat-def Suc-le-mono le0 trim-eq-imp-trim-eq*)
with *trim-1-eq* **have** *hl*: *length ts* = *length us* **by** *blast*
moreover have $\forall$ *i* < *length ts. composable-tree n* (*ts!i*) (*us!i*) **proof** (*safe*)
fix *i* **assume** *hi*: *i* < *length ts*
hence *height* (*ts!i*) $\leq$ *m* **using** *h*(*2*) *height-lt nth-mem* **by** *fastforce*
moreover have *height* (*us!i*) $\leq$ *m* **using** *hi h*(*3*) *height-lt nth-mem hl* **by** *fastforce*
moreover have *n* < *m* **using** *h*(*1*) **by** *simp*
moreover have *trim n* (*ts!i*) = *trim n* (*us!i*) **proof** −
have *map* (*trim n*) *ts* = *map* (*trim n*) *us* **using** *htrim* **by** *auto*
thus *trim n* (*ts!i*) = *trim n* (*us!i*) **using** *nth-map hi hl* **by** *metis*
qed
ultimately have *pd.t′ m n* (*us!i*) = *pd.s′ m n* (*ts!i*)
using *s′-eq-trim t′-eq-trim order-less-imp-le*[*of n m*] **by** *presburger*
hence *pd.is-composable-pair m n* (*ts!i*) (*us!i*)
using *pd.is-composable-pair-def* ‹*n* < *m*› ‹*height* (*ts!i*) $\leq$ *m*› ‹*height* (*us!i*) $\leq$ *m*› **by** *blast*
with *2*(*1*) *hi* **show** *composable-tree n* (*ts!i*) (*us!i*) **by** *fast*
qed
ultimately show *?case* **by** *fastforce*
qed

lemma *composable-tree-imp-trim-eq*: *composable-tree n t u* $\Longrightarrow$ *trim n t* = *trim n u*
proof (*induction n t u rule*: *composable-tree.induct*)
case (*1 ts us*)
then show *?case* **by** *simp*
next
case (*2 n ts us*)
then show *?case*

by (*metis* (*mono-tags*, *lifting*) *nth-map trim.simps*(*2*) *length-map nth-equalityI composable-tree.simps*(*2*))
qed

lemma *composable-tree-imp-is-composable-pair*:
assumes $n < m$ *height* $t \leq m$ *height* $u \leq m$ *composable-tree* $n\ t\ u$
shows *pd.is-composable-pair* $m\ n\ t\ u$
using *assms*
proof (*induction* m *arbitrary*: $n\ t\ u$)
case *0*
then show *?case* **by** *blast*
next
case (*Suc* m)
hence *trim* $n\ u$ = *trim* $n\ t$ **using** *composable-tree-imp-trim-eq* **by** *presburger*
hence $pd.t'$ (*Suc* m) $n\ u$ = $pd.s'$ (*Suc* m) $n\ t$
using *Suc*(*2*−*4*) s'*-eq-trim* t'*-eq-trim less-imp-le-nat* **by** *presburger*
with *Suc*(*2*−*4*) *pd.is-composable-pair-def* **show** *?case* **by** *blast*
qed

lemma *is-composable-pair-iff-composable-tree*: *pd.is-composable-pair* $m\ n\ t\ u$ =
($n < m \wedge$ *height* $t \leq m \wedge$ *height* $u \leq m \wedge$ *composable-tree* $n\ t\ u$)
by (*metis* (*mono-tags*, *lifting*) *composable-tree-imp-is-composable-pair is-composable-pair-imp-composable-tree mem-Collect-eq pd.is-composable-pair-def*)

lemma *composable-tree-imp-composable-tree-subtrees*:
composable-tree (*Suc* n) (*Node ts*) (*Node us*) $\Longrightarrow \forall (t, u) \in$ *set* (*zip ts us*). *composable-tree* $n\ t\ u$
by (*metis in-set-zip case-prod-beta composable-tree.simps*(*2*))

lemma *composable-tree-nth-subtrees*:
$[\![$*composable-tree* (*Suc* n) (*Node ts*) (*Node us*); $i <$ *length ts*$]\!] \Longrightarrow$ *composable-tree* n (*ts*!i) (*us*!i)
by *fastforce*

lemma *is-composable-pair-imp-is-composable-pair-subtrees*:
assumes *pd.is-composable-pair* (*Suc* m) (*Suc* n) (*Node ts*) (*Node us*)
shows $\forall (t, u) \in$ *set* (*zip ts us*). *pd.is-composable-pair* $m\ n\ t\ u$
proof
have *pd.is-composable-pair* $m\ n$ (*fst* p) (*snd* p) **if** *hp*: $p \in$ *set* (*zip ts us*) **for** p
proof −
have *composable-tree* (*Suc* n) (*Node ts*) (*Node us*)
using *is-composable-pair-iff-composable-tree assms* **by** *blast*
hence h: *composable-tree* n (*fst* p) (*snd* p)
using *hp composable-tree-imp-composable-tree-subtrees* **by** *fastforce*
have *fst* $p \in$ *set ts snd* $p \in$ *set us* **by** (*metis hp in-set-zipE prod.exhaust-sel*)+
hence *height* (*fst* p) $\leq m$ *height* (*snd* p) $\leq m$
by (*meson hp height-lt assms less-Suc-eq-le order-less-le-trans is-composable-pair-iff-composable-tree*)+
with h *is-composable-pair-iff-composable-tree assms*

show *pd.is-composable-pair m n* (*fst p*) (*snd p*) **by** *force*
qed
then show $\bigwedge x$. $x \in$ *set* (*zip ts us*) $\Longrightarrow$ *case x of* (*t*, *u*) $\Rightarrow$ *pd.is-composable-pair m n t u*
by *force*
qed

lemma *in-set-map2*: ($z \in$ *set* (*map2 f xs ys*)) = ($\exists$ (*x*, *y*) $\in$ *set* (*zip xs ys*). $z = f$ *x y*)
by *auto*

lemma *height-tree-comp-le*: ⟦*height t* $\le$ *m*; *height u* $\le$ *m*⟧ $\Longrightarrow$ *height* (*tree-comp n t u*) $\le$ *m*
proof (*induction n t u arbitrary*: *m rule*: *tree-comp.induct*)
case (*1 ts us*)
then show *?case* **using** *height-tree-comp-0* **by** *presburger*
next
case (*2 n ts us*)
show *?case* **proof** (*cases ts* $\neq$ [] $\wedge$ *us* $\neq$ [])
case *True*
hence $\exists m'$. $m = Suc\ m'$ **using** *height-zero 2.prems*(*1*) *not0-implies-Suc* **by** *auto*
then obtain m' **where** $m = Suc\ m'$ **by** *blast*
hence $\forall t \in$ *set ts*. *height t* $\le m'$ $\forall u \in$ *set us*. *height u* $\le m'$
using *True 2.prems* **by** *simp+*
hence $\forall$ (*t*, *u*) $\in$ *set* (*zip ts us*). *height* (*tree-comp n t u*) $\le m'$
by (*metis* (*no-types*, *lifting*) *2.IH case-prodI2 set-zip-leftD set-zip-rightD*)
then show *?thesis* **using** *True* ‹$m = Suc\ m'$› **by** *auto*
next
case *False*
then show *?thesis* **by** *force*
qed
qed

lemma *nth-map2* [*simp*]: ⟦$n <$ *length xs*; $n <$ *length ys*⟧ $\Longrightarrow$ *map2 f xs ys* ! $n = f$ (*xs* ! *n*) (*ys* ! *n*)
by *fastforce*

lemma *trim-tree-comp1*: *composable-tree n t u* $\Longrightarrow$ *trim n* (*tree-comp n t u*) = *trim n t*
proof (*induction n t u rule*: *composable-tree.induct*)
case (*1 ts us*)
then show *?case* **by** *fastforce*
next
case (*2 n ts us*)
then show *?case* **by** (*simp add*: *list-eq-iff-nth-eq*)
qed

lemma *trim-tree-comp2*: *composable-tree n t u* $\Longrightarrow$ *trim n* (*tree-comp n t u*) =

trim n u
  using *trim-tree-comp1 composable-tree-imp-trim-eq* **by** *presburger*

lemma *map2-map-map'*: *map2 f* (*map g xs*) (*map h ys*) = *map* (λ(*x*, *y*). *f* (*g x*) (*h y*)) (*zip xs ys*)
proof (*induction xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **proof** (*induction ys*)
    case *Nil*
    then show *?case* **by** *simp*
  next
    case (*Cons a ys*)
    then show *?case* **by** *auto*
  qed
qed

lemma *trim-tree-comp-commute*: *trim m* (*tree-comp n t u*) = *tree-comp n* (*trim m t*) (*trim m u*)
proof (*induction m arbitrary*: *n t u*)
  case *0*
  then show *?case* **by** (*cases n*, *simp-all*)
next
  case (*Suc m*)
  then show *?case*
    by (*induction n t u rule*: *composable-tree.induct*, *simp-all add*: *list-eq-iff-nth-eq*)
qed

interpretation *pd-pre-cat*: *pre-strict-omega-category PD* δ δ λ *m*. *tree-comp* λ *n*. *id*
proof (*unfold-locales*)
  fix *m n x' x* **assume** *pd.is-composable-pair m n x' x*
  then show *tree-comp n x' x* ∈ *PD m*
    using *is-composable-pair-iff-composable-tree height-tree-comp-le* **by** *auto*
next
  fix *n* **show** *id* ∈ *PD n* → *PD* (*Suc n*) **by** *simp*
next
  fix *m x' x* **assume** *pd.is-composable-pair* (*Suc m*) *m x' x*
  then show δ *m* (*tree-comp m x' x*) = δ *m x*
   by (*simp add*: *is-composable-pair-iff-composable-tree trim-tree-comp2 height-tree-comp-le*)
next
  fix *m x' x* **assume** *pd.is-composable-pair* (*Suc m*) *m x' x*
  then show δ *m* (*tree-comp m x' x*) = δ *m x'*
   by (*simp add*: *is-composable-pair-iff-composable-tree trim-tree-comp1 height-tree-comp-le*)
next
  fix *m n x' x* **assume** *pd.is-composable-pair* (*Suc m*) *n x' x n* < *m*
  then show δ *m* (*tree-comp n x' x*) = *tree-comp n* (δ *m x'*) (δ *m x*)

by (*simp add*: *is-composable-pair-iff-composable-tree trim-tree-comp-commute height-tree-comp-le*)
next
fix x n **assume** $x \in PD\ n$
then show $\delta\ n\ (id\ x) = x$ **using** *trim-const* **by** *auto*
qed

lemma *tree-comp-assoc*: *tree-comp n* (*tree-comp n t u*) v = *tree-comp n t* (*tree-comp n u v*)
proof (*induction n t u arbitrary*: *v rule*: *composable-tree.induct*)
case (*1 ts us*)
then show *?case* **by** (*metis append-assoc tree-comp.simps*(*1*) *tree.exhaust*)
next
case (*2 n ts us*)
define *vs* **where** *vs* = *subtrees v* **hence** *hv*: v = *Node vs* **by** *force*
let *?k* = *min* (*length ts*) (*min* (*length us*) (*length vs*))
have $\forall\, i <$ *?k*. *tree-comp n* (*tree-comp n* (*ts*!*i*) (*us*!*i*)) (*vs*!*i*) =
tree-comp n (*ts*!*i*) (*tree-comp n* (*us*!*i*) (*vs*!*i*)) **using** *2.IH* **by** *auto*
hence *map2* (*tree-comp n*) (*map2* (*tree-comp n*) *ts us*) *vs* =
map2 (*tree-comp n*) *ts* (*map2* (*tree-comp n*) *us vs*) **by** (*simp add*: *list-eq-iff-nth-eq*)
then show *?case* **using** *hv* **by** *auto*
qed

lemma *i′-eq-id*: $n \le m \Longrightarrow$ *pd-pre-cat.i′ m n* = *id*
proof (*induction m*)
case *0*
then show *?case* **using** *pd-pre-cat.i′.simps*(*1*) **by** *blast*
next
case (*Suc m*)
then show *?case* **by** (*metis pd-pre-cat.i′-Suc id-comp le-Suc-eq pd-pre-cat.i′-n-n*)
qed

lemma *composable-tree-trim1*: $n \le m \Longrightarrow$ *composable-tree n* (*trim m t*) *t*
proof (*induction n t arbitrary*: *m rule*: *trim.induct*)
case (*1 t*)
then show *?case* **by** (*metis composable-tree.simps*(*1*) *tree.exhaust*)
next
case (*2 n ts*)
hence $\exists\, m'.\ m$ = *Suc* m' **by** *presburger*
then obtain m' **where** *hm*: m = *Suc* m' $n \le m'$ **using** *2*(*2*) **by** *blast*
moreover hence $\forall\, i <$ *length ts*. *composable-tree n* ($\delta\ m'$ (*ts*!*i*)) (*ts*!*i*) **using** *2*(*1*) **by** *simp*
ultimately show *?case* **by** *force*
qed

lemma *composable-tree-trim2*: $n \le m \Longrightarrow$ *composable-tree n t* (*trim m t*)
using *sym-composable-tree composable-tree-trim1* **by** *presburger*

lemma *tree-comp-trim1*: *tree-comp n* (*trim n t*) *t* = *t*

by (*induction n t rule*: *trim.induct*, *simp add*: *tree.exhaust*, *simp add*: *list-eq-iff-nth-eq*)

lemma *tree-comp-trim2*: *tree-comp n t* (*trim n t*) = *t*
by (*induction n t rule*: *trim.induct*, *simp add*: *tree.exhaust*, *simp add*: *list-eq-iff-nth-eq*)

lemma *tree-comp-exchange*:
$[\![q < p$; *composable-tree p* y' *y*; *composable-tree p* x' *x*;
composable-tree q y' x'; *composable-tree q y x*$]\!] \Longrightarrow$
tree-comp q (*tree-comp p* y' *y*) (*tree-comp p* x' *x*) =
tree-comp p (*tree-comp q* y' x') (*tree-comp q y x*)
proof (*induction p* y' *y arbitrary*: *q* x' *x rule*: *composable-tree.induct*)
case (*1 ys' ys*)
then show *?case* **proof** (*induction q* x' *x rule*: *composable-tree.induct*)
case (*1 xs' xs*)
then show *?case* **by** *blast*
next
case (*2 q xs' xs*)
then show *?case* **by** *force*
qed
next
case (*2 p ys' ys*)
then show *?case* **proof** (*induction q* x' *x rule*: *composable-tree.induct*)
case (*1 ts us*)
then show *?case* **by** *force*
next
case (*2 n ts us*)
then show *?case* **by** (*simp add*: *list-eq-iff-nth-eq*)
qed
qed

interpretation *pd-cat'*: *strict-omega-category PD* δ δ λ *m*. *tree-comp* λ *n*. *id*
proof (*unfold-locales*)
fix *m n* x' *x* x'' **assume** *pd.is-composable-pair m n* x' *x* *pd.is-composable-pair m n* x'' x'
then show *tree-comp n* (*tree-comp n* x'' x') *x* = *tree-comp n* x'' (*tree-comp n* x' *x*)
using *tree-comp-assoc is-composable-pair-iff-composable-tree* **by** *force*
next
fix *n m x* **assume** $n < m$ $x \in PD\ m$
moreover hence *height* $x \le m$ **by** *simp*
ultimately show *tree-comp n* (*pd-pre-cat.i'* *m n* (*pd.t'* *m n x*)) *x* = *x*
by (*metis* (*no-types*, *lifting*) *i'-eq-id t'-eq-trim tree-comp-trim1 id-apply nat-less-le*)
next
fix *n m x* **assume** $n < m$ $x \in PD\ m$
moreover hence *height* $x \le m$ **by** *simp*
ultimately show *tree-comp n x* (*pd-pre-cat.i'* *m n* (*pd.s'* *m n x*)) = *x*
by (*metis* (*no-types*, *lifting*) *i'-eq-id s'-eq-trim tree-comp-trim2 id-apply nat-less-le*)
next
fix *q p m* y' *y* x' *x* **assume** $q < p$ $p < m$

pd.is-composable-pair m p y′ y pd.is-composable-pair m p x′ x
pd.is-composable-pair m q y′ x′ pd.is-composable-pair m q y x
then show *tree-comp q* (*tree-comp p y′ y*) (*tree-comp p x′ x*) =
tree-comp p (*tree-comp q y′ x′*) (*tree-comp q y x*)
using *is-composable-pair-iff-composable-tree tree-comp-exchange* **by** *meson*
qed (*simp*)

end

5 Acknowledgements

The work has been jointly supported by the Cambridge Mathematics Placements (CMP) Programme and the ERC Advanced Grant ALEXANDRIA (Project GA 742178).

References

[1] T. Leinster. *Higher operads, higher categories.* Number 298. Cambridge University Press, 2004.