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*: $f \in A \rightarrow_E B \Longrightarrow f \in A \rightarrow B$ ⟨*proof*⟩

lemma *PiE-iff'*: $f \in A \rightarrow_E B = (f \in A \rightarrow B \wedge f \in extensional\ A)$
⟨*proof*⟩

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

lemma *compose-PiE*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow g \circ f \downarrow A \in A \to_E C$
⟨*proof*⟩

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*⟩

lemma *compose-eq-if*: $(\bigwedge x.\ x \in A \Longrightarrow g\ (f\ x) = k\ (h\ x)) \Longrightarrow g \circ f \downarrow A = k \circ h \downarrow A$
⟨*proof*⟩

lemma *compose-compose-eq-iff2*: $(h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A) = (\forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x)))$
⟨*proof*⟩

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*⟩

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

lemma *compose-compose-eq-if2*: $\forall x \in A.\ h\ (g\ (f\ x)) = h'\ (g'\ (f'\ x)) \Longrightarrow$
$h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A$
⟨*proof*⟩

lemma *compose-restrict-eq1*: $f \in A \to B \Longrightarrow$ *restrict* $g\ B \circ f \downarrow A = g \circ f \downarrow A$
⟨*proof*⟩

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

lemma *compose-Id-eq-restrict*: $g \circ (\lambda x \in A.\ x) \downarrow A =$ *restrict* $g\ A$
⟨*proof*⟩

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* ⇒ *'a set* **and** s :: *nat* ⇒ *'a* ⇒ *'a* **and** t :: *nat* ⇒ *'a* ⇒ *'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'*: $s\ n \circ t\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n)) = s\ n \circ s\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n))$
⟨*proof*⟩

lemma *t-comp'*: $t\ n \circ s\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n)) = t\ n \circ t\ (Suc\ n) \downarrow X\ (Suc\ (Suc\ n))$
⟨*proof*⟩

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* ⇒ *nat* ⇒ *'a* ⇒ *'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* ⇒ *nat* ⇒ *'a* ⇒ *'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*]: $s'\ n\ n = id$
⟨*proof*⟩

lemma *s'-Suc-n-n* [*simp*]: $s'\ (Suc\ n)\ n = s\ n$
⟨*proof*⟩

lemma *s'-Suc-Suc-n-n* [*simp*]: $s'\ (Suc\ (Suc\ n))\ n = s\ n \circ s\ (Suc\ n)$
⟨*proof*⟩

lemma *s'-Suc* [*simp*]: $n \le m \Longrightarrow s'\ (Suc\ m)\ n = s'\ m\ n \circ s\ m$
⟨*proof*⟩

lemma *s'-Suc'*: $n < m \Longrightarrow s'\ m\ n = s\ n \circ s'\ m\ (Suc\ n)$
⟨*proof*⟩

lemma *t'-n-n* [*simp*]: $t'\ n\ n = id$
⟨*proof*⟩

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

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

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

lemma *t′-Suc′*: $n < m \Longrightarrow t'\ m\ n = t\ n \circ t'\ m$ (*Suc* n)
⟨*proof*⟩

lemma *s′-fun*: $n \leq m \Longrightarrow s'\ m\ n \in X\ m \to X\ n$
⟨*proof*⟩

lemma *t′-fun*: $n \leq m \Longrightarrow t'\ m\ n \in X\ m \to X\ n$
⟨*proof*⟩

lemma *s′-comp*: $[\![\ n < m;\ x \in X\ m\ ]\!] \Longrightarrow s\ n\ (t'\ m$ (*Suc* n) $x) = s'\ m\ n\ x$
⟨*proof*⟩

lemma *t′-comp*: $[\![\ n < m;\ x \in X\ m\ ]\!] \Longrightarrow t\ n\ (s'\ m$ (*Suc* n) $x) = t'\ m\ n\ x$
⟨*proof*⟩

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

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

[1, p. 44]

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

definition *composable-pairs* :: *nat* ⇒ *nat* ⇒ (*′a* × *′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* $m\ n = \{\}$
⟨*proof*⟩

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* ⇒ *'a* ⇒ *'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*⟩

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*⟩

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* $X\ s_X\ \ t_X\ \ Y\ s_Y\ \ t_Y\ \ \varphi$ + *snd*: *globular-map* $Y\ s_Y\ \ t_Y\ \ Z\ s_Z\ \ t_Z\ \ \psi$
for $X\ s_X\ \ t_X\ \ Y\ s_Y\ \ t_Y\ \ Z\ s_Z\ \ t_Z\ \ \varphi\ \psi$

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*⟩

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*⟩

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* $\lambda m.\ \{()\}\ \lambda m.\ id\ \lambda m.\ id$
⟨*proof*⟩

context *globular-set*
begin

[1, p. 272]

interpretation *map-to-final-glob*: *globular-map* $X\ s\ t$
$\lambda m.\ \{()\}\ \lambda m.\ id\ \lambda m.\ id$
$\lambda m.\ (\lambda x.\ ())$
⟨*proof*⟩

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* ⇒ *nat* ⇒ *'a* ⇒ *'a* ⇒ *'a*
    and *i* :: *nat* ⇒ *'a* ⇒ *'a*
  assumes *comp-fun*: *is-composable-pair m n x' x* ⟹ *comp m n x' x* ∈ *X m*
    and *i-fun*: *i n* ∈ *X n* → *X* (*Suc n*)
    and *s-comp-Suc*: *is-composable-pair* (*Suc m*) *m x' x* ⟹ *s m* (*comp* (*Suc m*) *m x' x*) = *s m x*
    and *t-comp-Suc*: *is-composable-pair* (*Suc m*) *m x' x* ⟹ *t m* (*comp* (*Suc m*) *m x' x*) = *t m x'*
    and *s-comp*: ⟦*is-composable-pair* (*Suc m*) *n x' x*; *n* < *m*⟧ ⟹
        s m (*comp* (*Suc m*) *n x' x*) = *comp m n* (*s m x'*) (*s m x*)
    and *t-comp*: ⟦*is-composable-pair* (*Suc m*) *n x' x*; *n* < *m*⟧ ⟹
        t m (*comp* (*Suc m*) *n x' x*) = *comp m n* (*t m x'*) (*s m x*)
    and *s-i*: *x* ∈ *X n* ⟹ *s n* (*i n x*) = *x*
    and *t-i*: *x* ∈ *X n* ⟹ *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*
  ⟨*proof*⟩

lemma *i'-Suc-n-n* [*simp*]: *i'* (*Suc n*) *n* = *i n*
  ⟨*proof*⟩

lemma *i′-Suc* [*simp*]: $n \le m \Longrightarrow i'\ (Suc\ m)\ n = i\ m \circ i'\ m\ n$
⟨*proof*⟩

lemma *i′-Suc′*: $n < m \Longrightarrow i'\ m\ n = i'\ m\ (Suc\ n) \circ i\ n$
⟨*proof*⟩

lemma *i′-fun*: $n \le m \Longrightarrow i'\ m\ n \in X\ n \to X\ m$
⟨*proof*⟩

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* = *Leaf*) = (*subtrees* *t* = [])
⟨*proof*⟩

fun *is-subtree-index* :: *tree* ⇒ *nat list* ⇒ *bool* **where**
is-subtree-index *t* [] = *True* |
is-subtree-index *t* (*i*#*xs*) = (*is-subtree-index* *t* *xs* ∧ *i* < *length* (*subtrees* (*t* !*t* *xs*)))

lemma *subtree-append*: *ts* ! *i* !*t* *xs* = *Node* *ts* !*t* *xs* @ [*i*]
⟨*proof*⟩

lemma *is-subtree-index-append* [*iff*]: *is-subtree-index* (*Node* *ts*) (*xs* @ [*i*]) =
(*i* < *length* *ts* ∧ *is-subtree-index* (*ts*!*i*) *xs*)
⟨*proof*⟩

lemma *is-subtree-index-append′* [*iff*]: *is-subtree-index* *t* (*xs* @ [*i*]) =
(*is-subtree-index* *t* [*i*] ∧ *is-subtree-index* (*t* !*t* [*i*]) *xs*)
⟨*proof*⟩

lemma *max-set-upt* [*simp*]: *Max* {0..<*Suc* *n*} = *n*
⟨*proof*⟩

lemma *length-subtrees-eq-Max*: **assumes** *is-subtree-index* *t* *xs* *subtrees* (*t* !*t* *xs*) ≠
[]
shows *length* (*subtrees* (*t* !*t* *xs*)) = *Suc* (*Max* {*i*. *is-subtree-index* *t* (*i* # *xs*)})
⟨*proof*⟩

lemma *tree-eq-iff-subtree-eq*: (*t* = *u*) = (*length* (*subtrees* *t*) = *length* (*subtrees* *u*)
∧
(∀ *i* < *length* (*subtrees* *t*). *t* !*t* [*i*] = *u* !*t* [*i*]))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fold-eq-Max* [*simp*]: *ts* ≠ [] ⟹ *fold* (*max* ∘ *height*) *ts 0* = *Max* (*set* (*map height ts*))
⟨*proof*⟩

lemma *height-Node-Max*: *ts* ≠ [] ⟹ *height* (*Node ts*) = *Suc* (*Max* (*set* (*map height ts*)))
⟨*proof*⟩

lemma *height-Node-pos* : *ts* ≠ [] ⟹ *0* < *height* (*Node ts*)
⟨*proof*⟩

lemma *height-exists*:
assumes *height* (*Node ts*) = *Suc n*
shows ∃ *t*. *t* ∈ *set ts* ∧ *height t* = *n*
⟨*proof*⟩

lemma *height-lt*: **assumes** *t* ∈ *set ts* **shows** *height t* < *height* (*Node ts*)
⟨*proof*⟩

lemma *height-le-imp-le-Suc*:
assumes ∀ *t* ∈ *set ts*. *height t* ≤ *n*
shows *height* (*Node ts*) ≤ *Suc n*
⟨*proof*⟩

lemma *height-zero* [*simp*]: *height t* = *0* ⟹ *t* = *Leaf*
⟨*proof*⟩

lemma *is-subtree-index-length-le*: *is-subtree-index t xs* ⟹ *length xs* ≤ *height t*
⟨*proof*⟩

lemma *height-subtree*: *is-subtree-index t xs* ⟹ *height* (*t* !*t xs*) ≤ *height t* − *length xs*
⟨*proof*⟩

lemma *height-induct*: (⋀*t*. ∀ *u*. *height u* < *height t* ⟶ *P u* ⟹ *P t*) ⟹ *P t*
⟨*proof*⟩

lemma *subtree-index-induct* [*case-names Index Step*]:
assumes
is-subtree-index t xs
$\bigwedge xs$. ⟦*is-subtree-index t xs*; ∀ *i* < *length* (*subtrees* (*t* !*t* *xs*)). *P* (*i*#*xs*)⟧ ⟹ *P xs*
shows *P xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *height-trim-le*: *height* (*trim n t*) ≤ *n*
⟨*proof*⟩

lemma *trim-const*: *height t* ≤ *n* ⟹ *trim n t* = *t*
⟨*proof*⟩

lemma *height-trim-le'*: *n* ≤ *height t* ⟹ *height* (*trim n t*) = *n*
⟨*proof*⟩

lemma *height-trim*: *height* (*trim n t*) = (*if n* ≤ *height t then n else height t*)
⟨*proof*⟩

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* ∘ *trim n* = *trim n*
⟨*proof*⟩

lemma *trim-trim-Suc* [*simp*]: *trim n* ∘ *trim* (*Suc n*) = *trim n*
⟨*proof*⟩

lemma *trim-trim* [*simp*]: *n* ≤ *m* ⟹ *trim n* ∘ *trim m* = *trim n*
⟨*proof*⟩

lemma *trim-eq-imp-trim-eq* [*simp*]: ⟦*n* ≤ *m*; *trim m t* = *trim m u*⟧ ⟹ *trim n t* = *trim n u*
⟨*proof*⟩

lemma *trim-1-eq*: **assumes** *trim 1* (*Node ts*) = *trim 1* (*Node us*) **shows** *length ts* = *length us*

⟨*proof*⟩

lemma *length-subtrees-trim-Suc*: *length* (*subtrees* (*trim* (*Suc n*) *t*)) = *length* (*subtrees* *t*)
⟨*proof*⟩

lemma *trim-eq-Leaf*: *trim n t* = *Leaf* ⟹ *n* = *0* ∨ *t* = *Leaf*
⟨*proof*⟩

lemma *map-eq-imp-pairs-eq*: *map f xs* = *map g ys* ⟹ (⋀*x y*. (*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *f x* = *g y*)
⟨*proof*⟩

lemma *trim-eq-subtree-eq*:
assumes *trim* (*Suc n*) (*Node ts*) = *trim* (*Suc n*) (*Node us*)
shows ⋀*t u*. (*t*, *u*) ∈ *set* (*zip ts us*) ⟹ *trim n t* = *trim n u*
⟨*proof*⟩

lemma *pairs-eq-imp-map-eq*:
assumes *length xs* = *length ys* ∀ (*x*, *y*) ∈ *set* (*zip xs ys*). *f x* = *g y*
shows *map f xs* = *map g ys*
⟨*proof*⟩

lemma *map-eq-iff-pairs-eq*: (*map f xs* = *map g ys*) =
(*length xs* = *length ys* ∧ (∀ (*x*, *y*) ∈ *set* (*zip xs ys*). *f x* = *g y*))
⟨*proof*⟩

lemma *subtree-eq-trim-eq*:
assumes *length ts* = *length us* ∀ (*t*, *u*) ∈ *set* (*zip ts us*). *trim n t* = *trim n u*
shows *trim* (*Suc n*) (*Node ts*) = *trim* (*Suc n*) (*Node us*)
⟨*proof*⟩

lemma *subtree-trim-1*: *is-subtree-index t* [*i*] ⟹ *trim* (*Suc n*) *t* !*t* [*i*] = *trim n* (*t* !*t* [*i*])
⟨*proof*⟩

lemma *is-subtree-index-trim*:
is-subtree-index (*trim n t*) *xs* = (*is-subtree-index t xs* ∧ *length xs* ≤ *n*)
⟨*proof*⟩

lemma *subtree-trim*: ⟦*is-subtree-index t xs*; *length xs* ≤ *n*⟧ ⟹
trim n t !*t xs* = *trim* (*n* − *length xs*) (*t* !*t xs*)
⟨*proof*⟩

lemma *length-subtrees-trim*: ⟦*is-subtree-index t xs*; *length xs* < *n*⟧ ⟹
length (*subtrees* (*trim n t* !*t xs*)) = *length* (*subtrees* (*t* !*t xs*))
⟨*proof*⟩

lemma *subtree-trim-Leaf*: **assumes** *is-subtree-index* (*trim n t*) *xs t* !*t xs* = *Leaf*

shows *trim n t* !*t xs* = *Leaf*
⟨*proof*⟩

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**
δ ≡ *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* ⇒ *tree set* **where**
PD n ≡ {*t*. *height t* ≤ *n*}

interpretation *pd*: *globular-set PD* δ δ
⟨*proof*⟩

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

lemma *s′-eq-trim*: **assumes** *n* ≤ *m height t* ≤ *m* **shows** *pd.s′ m n t* = *trim n t*
⟨*proof*⟩

lemma *s′-eq-t′*: *pd.s′* = *pd.t′*
⟨*proof*⟩

lemma *t′-eq-trim*: **assumes** *n* ≤ *m height t* ≤ *m* **shows** *pd.t′ m n t* = *trim n t*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *tree-comp-0-Leaf2* [*simp*]: *tree-comp 0 t Leaf* = *t*
  ⟨*proof*⟩

lemma *tree-comp-Suc-Leaf1* [*simp*]: *tree-comp* (*Suc n*) *Leaf t* = *Leaf*
  ⟨*proof*⟩

lemma *tree-comp-Suc-Leaf2* [*simp*]: *tree-comp* (*Suc n*) *t Leaf* = *Leaf*
  ⟨*proof*⟩

lemma *height-tree-comp-0* [*simp*]: *height* (*tree-comp 0 t u*) = *max* (*height t*) (*height u*)
⟨*proof*⟩

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* ∧
  (∀ *i* < *length ts*. *composable-tree n* (*ts*!*i*) (*us*!*i*)))

lemma *sym-composable-tree*: *composable-tree n t u* = *composable-tree n u t*
  ⟨*proof*⟩

lemma *is-composable-pair-imp-composable-tree*: *pd.is-composable-pair m n t u* ⟹
  composable-tree n t u
⟨*proof*⟩

lemma *composable-tree-imp-trim-eq*: *composable-tree n t u* ⟹ *trim n t* = *trim n u*
⟨*proof*⟩

lemma *composable-tree-imp-is-composable-pair*:
  assumes *n* < *m height t* ≤ *m height u* ≤ *m composable-tree n t u*
  shows *pd.is-composable-pair m n t u*
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *composable-tree-nth-subtrees*:
⟦*composable-tree* (*Suc n*) (*Node ts*) (*Node us*); $i <$ *length ts*⟧ $\Longrightarrow$ *composable-tree n* (*ts*!*i*) (*us*!*i*)
⟨*proof*⟩

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*⟩

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

lemma *height-tree-comp-le*: ⟦*height* $t \leq m$; *height* $u \leq m$⟧ $\Longrightarrow$ *height* (*tree-comp n t u*) $\leq m$
⟨*proof*⟩

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

lemma *trim-tree-comp1*: *composable-tree n t u* $\Longrightarrow$ *trim n* (*tree-comp n t u*) = *trim n t*
⟨*proof*⟩

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

lemma *map2-map-map′*: *map2 f* (*map g xs*) (*map h ys*) = *map* ($\lambda(x, y)$. *f* (*g x*) (*h y*)) (*zip xs ys*)
⟨*proof*⟩

lemma *trim-tree-comp-commute*: *trim m* (*tree-comp n t u*) = *tree-comp n* (*trim m t*) (*trim m u*)
⟨*proof*⟩

interpretation *pd-pre-cat*: *pre-strict-omega-category PD* δ δ λ *m*. *tree-comp* λ *n*.

id
⟨*proof*⟩

lemma *tree-comp-assoc*: *tree-comp n* (*tree-comp n t u*) *v* = *tree-comp n t* (*tree-comp n u v*)
⟨*proof*⟩

lemma *i′-eq-id*: $n \leq m \Longrightarrow$ *pd-pre-cat.i′ m n* = *id*
⟨*proof*⟩

lemma *composable-tree-trim1*: $n \leq m \Longrightarrow$ *composable-tree n* (*trim m t*) *t*
⟨*proof*⟩

lemma *composable-tree-trim2*: $n \leq m \Longrightarrow$ *composable-tree n t* (*trim m t*)
⟨*proof*⟩

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

lemma *tree-comp-trim2*: *tree-comp n t* (*trim n t*) = *t*
⟨*proof*⟩

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*⟧ ⟹
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*⟩

interpretation *pd-cat′*: *strict-omega-category PD δ δ λ m. tree-comp λ n. id*
⟨*proof*⟩

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.