

# Strict Omega Categories

Anthony Bordg      Adrián Doña Mateo

June 6, 2026

## Abstract

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

## Contents

<b>1</b>	<b>Background material on extensional functions</b>	<b>1</b>
<b>2</b>	<b>Globular sets</b>	<b>2</b>
2.1	Globular sets . . . . .	2
2.2	Maps between globular sets . . . . .	4
2.3	The terminal globular set . . . . .	5
<b>3</b>	<b>Strict <math>\omega</math>-categories</b>	<b>6</b>
<b>4</b>	<b>The category of pasting diagrams</b>	<b>7</b>
4.1	Rooted trees . . . . .	8
4.2	The strict $\omega$ -category of pasting diagrams . . . . .	12
<b>5</b>	<b>Acknowledgements</b>	<b>15</b>
<code>theory</code>	<code>Globular-Set</code>	

```
imports HOL-Library.FuncSet
```

```
begin
```

## 1 Background material on extensional functions

```
lemma PiE-imp-Pi:  $f \in A \rightarrow_E B \implies f \in A \rightarrow B$  <proof>
```

```
lemma PiE-iff':  $f \in A \rightarrow_E B = (f \in A \rightarrow B \wedge f \in \text{extensional } A)$   
<proof>
```

**abbreviation** *composing* ( $\langle - \circ - \downarrow \rightarrow [60,0,60]59$ )

where  $g \circ f \downarrow D \equiv \text{compose } D \ g \ f$

**lemma** *compose-PiE*:  $f \in A \rightarrow B \implies g \in B \rightarrow C \implies g \circ f \downarrow A \in A \rightarrow_E C$   
 $\langle \text{proof} \rangle$

**lemma** *compose-eq-iff*:  $(g \circ f \downarrow A = k \circ h \downarrow A) = (\forall x \in A. g (f x) = k (h x))$   
 $\langle \text{proof} \rangle$

**lemma** *compose-eq-if*:  $(\bigwedge x. x \in A \implies g (f x) = k (h x)) \implies g \circ f \downarrow A = k \circ h \downarrow A$   
 $\langle \text{proof} \rangle$

**lemma** *compose-compose-eq-iff2*:  $(h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A) =$   
 $(\forall x \in A. h (g (f x)) = h' (g' (f' x)))$   
 $\langle \text{proof} \rangle$

**lemma** *compose-compose-eq-iff1*: **assumes**  $f \in A \rightarrow B \ f' \in A \rightarrow B$   
**shows**  $((h \circ g \downarrow B) \circ f \downarrow A = (h' \circ g' \downarrow B) \circ f' \downarrow A) = (\forall x \in A. h (g (f x)) = h' (g' (f' x)))$   
 $\langle \text{proof} \rangle$

**lemma** *compose-compose-eq-if1*:  $\llbracket f \in A \rightarrow B; f' \in A \rightarrow B; \forall x \in A. h (g (f x)) = h' (g' (f' x)) \rrbracket \implies$   
 $(h \circ g \downarrow B) \circ f \downarrow A = (h' \circ g' \downarrow B) \circ f' \downarrow A$   
 $\langle \text{proof} \rangle$

**lemma** *compose-compose-eq-if2*:  $\forall x \in A. h (g (f x)) = h' (g' (f' x)) \implies$   
 $h \circ (g \circ f \downarrow A) \downarrow A = h' \circ (g' \circ f' \downarrow A) \downarrow A$   
 $\langle \text{proof} \rangle$

**lemma** *compose-restrict-eq1*:  $f \in A \rightarrow B \implies \text{restrict } g \ B \circ f \downarrow A = g \circ f \downarrow A$   
 $\langle \text{proof} \rangle$

**lemma** *compose-restrict-eq2*:  $g \circ (\text{restrict } f \ A) \downarrow A = g \circ f \downarrow A$   
 $\langle \text{proof} \rangle$

**lemma** *compose-Id-eq-restrict*:  $g \circ (\lambda x \in A. x) \downarrow A = \text{restrict } g \ A$   
 $\langle \text{proof} \rangle$

## 2 Globular sets

### 2.1 Globular sets

We define a locale *globular-set* that encodes the cell data of a strict  $\omega$ -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**  $X :: \text{nat} \Rightarrow 'a \text{ set}$  **and**  $s :: \text{nat} \Rightarrow 'a \Rightarrow 'a$  **and**  $t :: \text{nat} \Rightarrow 'a \Rightarrow 'a$   
**assumes**  $s\text{-fun}: s\ n \in X\ (\text{Suc}\ n) \rightarrow X\ n$   
**and**  $t\text{-fun}: t\ n \in X\ (\text{Suc}\ n) \rightarrow X\ n$   
**and**  $s\text{-comp}: x \in X\ (\text{Suc}\ (\text{Suc}\ n)) \Longrightarrow s\ n\ (t\ (\text{Suc}\ n)\ x) = s\ n\ (s\ (\text{Suc}\ n)\ x)$   
**and**  $t\text{-comp}: x \in X\ (\text{Suc}\ (\text{Suc}\ n)) \Longrightarrow t\ n\ (s\ (\text{Suc}\ n)\ x) = t\ n\ (t\ (\text{Suc}\ n)\ x)$   
**begin**

**lemma**  $s\text{-comp}'$ :  $s\ n \circ t\ (\text{Suc}\ n) \downarrow X\ (\text{Suc}\ (\text{Suc}\ n)) = s\ n \circ s\ (\text{Suc}\ n) \downarrow X\ (\text{Suc}\ (\text{Suc}\ n))$   
 $\langle \text{proof} \rangle$

**lemma**  $t\text{-comp}'$ :  $t\ n \circ s\ (\text{Suc}\ n) \downarrow X\ (\text{Suc}\ (\text{Suc}\ n)) = t\ n \circ t\ (\text{Suc}\ n) \downarrow X\ (\text{Suc}\ (\text{Suc}\ n))$   
 $\langle \text{proof} \rangle$

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

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

**lemma**  $s'\text{-n-n}$  [*simp*]:  $s'\ n\ n = \text{id}$   
 $\langle \text{proof} \rangle$

**lemma**  $s'\text{-Suc-n-n}$  [*simp*]:  $s'\ (\text{Suc}\ n)\ n = s\ n$   
 $\langle \text{proof} \rangle$

**lemma**  $s'\text{-Suc-Suc-n-n}$  [*simp*]:  $s'\ (\text{Suc}\ (\text{Suc}\ n))\ n = s\ n \circ s\ (\text{Suc}\ n)$   
 $\langle \text{proof} \rangle$

**lemma**  $s'\text{-Suc}$  [*simp*]:  $n \leq m \Longrightarrow s'\ (\text{Suc}\ m)\ n = s'\ m\ n \circ s\ m$   
 $\langle \text{proof} \rangle$

**lemma**  $s'\text{-Suc}'$ :  $n < m \Longrightarrow s'\ m\ n = s\ n \circ s'\ m\ (\text{Suc}\ n)$   
 $\langle \text{proof} \rangle$

**lemma**  $t'\text{-n-n}$  [*simp*]:  $t'\ n\ n = \text{id}$   
 $\langle \text{proof} \rangle$

**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]*:  $n \leq m \implies t' (Suc\ m)\ n = t' m\ n \circ t\ m$   
 ⟨proof⟩

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

**lemma** *s'-fun*:  $n \leq m \implies s' m\ n \in X\ m \rightarrow X\ n$   
 ⟨proof⟩

**lemma** *t'-fun*:  $n \leq m \implies t' m\ n \in X\ m \rightarrow X\ n$   
 ⟨proof⟩

**lemma** *s'-comp*:  $\llbracket n < m; x \in X\ m \rrbracket \implies s\ n\ (t' m\ (Suc\ n)\ x) = s' m\ n\ x$   
 ⟨proof⟩

**lemma** *t'-comp*:  $\llbracket n < m; x \in X\ m \rrbracket \implies t\ n\ (s' m\ (Suc\ n)\ x) = t' m\ n\ x$   
 ⟨proof⟩

The following predicates and sets are needed to define composition in an  $\omega$ -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*  $m\ n = \{(y, x). \text{is-composable-pair } m\ n\ y\ x\}$

**lemma** *composable-pairs-empty*:  $m \leq n \implies \text{composable-pairs } m\ n = \{\}$   
 ⟨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  $\varphi :: \text{nat} \Rightarrow 'a \Rightarrow 'b$ 
assumes  $\text{map-fun}: \varphi m \in X m \rightarrow Y m$ 
  and  $\text{is-natural-wrt-s}: x \in X (\text{Suc } m) \Longrightarrow \varphi m (s_X m x) = s_Y m (\varphi (\text{Suc } m) x)$ 
  and  $\text{is-natural-wrt-t}: x \in X (\text{Suc } m) \Longrightarrow \varphi m (t_X m x) = t_Y m (\varphi (\text{Suc } m) x)$ 
begin

```

```

lemma  $\text{is-natural-wrt-s}': \llbracket n \leq m; x \in X m \rrbracket \Longrightarrow \varphi n (\text{source.s}' m n x) = \text{target.s}' m n (\varphi m x)$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma  $\text{is-natural-wrt-t}': \llbracket n \leq m; x \in X m \rrbracket \Longrightarrow \varphi n (\text{source.t}' m n x) = \text{target.t}' m n (\varphi m x)$ 
 $\langle \text{proof} \rangle$ 

```

**end**

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

```

locale  $\text{two-globular-maps} = \text{fst}: \text{globular-map } X s_X t_X Y s_Y t_Y \varphi + \text{snd}: \text{globular-map } Y s_Y t_Y Z s_Z t_Z \psi$ 
  for  $X s_X t_X Y s_Y t_Y Z s_Z t_Z \varphi \psi$ 

```

```

sublocale  $\text{two-globular-maps} \subseteq \text{comp}: \text{globular-map } X s_X t_X Z s_Z t_Z \lambda m. \psi m \circ \varphi m$ 
 $\langle \text{proof} \rangle$ 

```

```

sublocale  $\text{two-globular-maps} \subseteq \text{compose}: \text{globular-map } X s_X t_X Z s_Z t_Z \lambda m. \psi m \circ \varphi m \downarrow X m$ 
 $\langle \text{proof} \rangle$ 

```

### 2.3 The terminal globular set

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

```

interpretation  $\text{final-glob}: \text{globular-set } \lambda m. \{()\} \lambda m. \text{id } \lambda m. \text{id}$ 
 $\langle \text{proof} \rangle$ 

```

```

context  $\text{globular-set}$ 
begin

```

[1, p. 272]

```

interpretation  $\text{map-to-final-glob}: \text{globular-map } X s t$ 
 $\lambda m. \{()\} \lambda m. \text{id } \lambda m. \text{id}$ 
 $\lambda m. (\lambda x. ())$ 
 $\langle \text{proof} \rangle$ 

```

**end**

```

end
theory Strict-Omega-Category
imports Globular-Set

```

```
begin
```

### 3 Strict $\omega$ -categories

First, we define a locale *pre-strict-omega-category* that holds the data of a strict  $\omega$ -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-composable-pair m n x' x  $\Longrightarrow$  comp m n x' x  $\in$  X m
    and i-fun: i n  $\in$  X n  $\rightarrow$  X (Suc n)
    and s-comp-Suc: is-composable-pair (Suc m) m x' x  $\Longrightarrow$  s m (comp (Suc m)
m x' x) = s m x
    and t-comp-Suc: is-composable-pair (Suc m) m x' x  $\Longrightarrow$  t m (comp (Suc m) m
x' x) = t m x'
    and s-comp:  $\llbracket$ is-composable-pair (Suc m) n x' x; n < m $\rrbracket$   $\Longrightarrow$ 
      s m (comp (Suc m) n x' x) = comp m n (s m x') (s m x)
    and t-comp:  $\llbracket$ is-composable-pair (Suc m) n x' x; n < m $\rrbracket$   $\Longrightarrow$ 
      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  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  '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  $\circ$  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*]:  $n \leq m \implies i' (Suc\ m)\ n = i\ m \circ i'\ m\ n$   
 ⟨*proof*⟩

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

**lemma** *i'-fun*:  $n \leq m \implies i'\ m\ n \in X\ n \rightarrow X\ m$   
 ⟨*proof*⟩

**end**

Now we may define a strict  $\omega$ -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*:  $\llbracket is\text{-composable-pair}\ m\ n\ x'\ x;\ is\text{-composable-pair}\ m\ n\ x''\ x' \rrbracket \implies$

$comp\ m\ n\ (comp\ m\ n\ x''\ x')\ x = comp\ m\ n\ x''\ (comp\ m\ n\ x'\ x)$

**and** *i-comp*:  $\llbracket n < m;\ x \in X\ m \rrbracket \implies comp\ m\ n\ (i'\ m\ n\ (t'\ m\ n\ x))\ x = x$

**and** *comp-i*:  $\llbracket n < m;\ x \in X\ m \rrbracket \implies comp\ m\ n\ x\ (i'\ m\ n\ (s'\ m\ n\ x)) = x$

**and** *bin-interchange*:  $\llbracket q < p;\ p < m \rrbracket$

$is\text{-composable-pair}\ m\ p\ y'\ y;\ is\text{-composable-pair}\ m\ p\ x'\ x;$

$is\text{-composable-pair}\ m\ q\ y'\ x';\ is\text{-composable-pair}\ m\ q\ y\ x \rrbracket \implies$

$comp\ m\ q\ (comp\ m\ p\ y'\ y)\ (comp\ m\ p\ x'\ x) = comp\ m\ p\ (comp\ m\ q\ y'\ x')$

( $comp\ m\ q\ y\ x$ )

**and** *null-interchange*:  $\llbracket q < p;\ is\text{-composable-pair}\ p\ q\ x'\ x \rrbracket \implies$

$comp\ (Suc\ p)\ q\ (i\ p\ x')\ (i\ p\ x) = i\ p\ (comp\ p\ q\ x'\ x)$

**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*:  $is\text{-composable-pair}\ m\ n\ x'\ x \implies$

$\varphi\ m\ (comp_X\ m\ n\ x'\ x) = comp_Y\ m\ n\ (\varphi\ m\ x')\ (\varphi\ m\ x)$

**and** *commute-with-id*:  $x \in X\ n \implies \varphi\ (Suc\ n)\ (i_X\ n\ x) = i_Y\ n\ (\varphi\ n\ x)$

**end**

**theory** *Pasting-Diagram*

**imports** *Strict-Omega-Category*

**begin**

## 4 The category of pasting diagrams

We define the strict  $\omega$ -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  $\Rightarrow$  nat where
  height Leaf = 0 |
  height (Node ts) = Suc (fold (max  $\circ$  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  $\neq$  []  $\implies$  height (Node ts) = Suc (fold (max  $\circ$  height)
  ts 0)
  <proof>

lemma fold-eq-Max [simp]: ts  $\neq$  []  $\implies$  fold (max  $\circ$  height) ts 0 = Max (set (map
  height ts))
  <proof>

lemma height-Node-Max: ts  $\neq$  []  $\implies$  height (Node ts) = Suc (Max (set (map
  height ts)))
  <proof>

lemma height-Node-pos : ts  $\neq$  []  $\implies$  0 < height (Node ts)
  <proof>

lemma height-exists:
  assumes height (Node ts) = Suc n
  shows  $\exists t. t \in \text{set } ts \wedge \text{height } t = n$ 
  <proof>

lemma height-lt: assumes t  $\in$  set ts shows height t < height (Node ts)
  <proof>

lemma height-le-imp-le-Suc:
  assumes  $\forall t \in \text{set } ts. \text{height } t \leq n$ 
  shows height (Node ts)  $\leq$  Suc n
  <proof>

lemma height-zero [simp]: height t = 0  $\implies$  t = Leaf
  <proof>

lemma is-subtree-index-length-le: is-subtree-index t xs  $\implies$  length xs  $\leq$  height t
  <proof>

lemma height-subtree: is-subtree-index t xs  $\implies$  height (t !t xs)  $\leq$  height t - length
  xs
  <proof>

lemma height-induct: ( $\bigwedge t. \forall u. \text{height } u < \text{height } t \longrightarrow P u \implies P t$ )  $\implies$  P t
  <proof>

```

**lemma** *subtree-index-induct* [*case-names Index Step*]:

**assumes**

*is-subtree-index t xs*

$\bigwedge xs. \llbracket is-subtree-index\ t\ xs; \forall i < length\ (subtrees\ (t\ !t\ xs)).\ P\ (i\#xs) \rrbracket \implies P\ xs$

**shows** *P xs*

*<proof>*

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

**fun** *trim* :: *nat*  $\Rightarrow$  *tree*  $\Rightarrow$  *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*)  $\leq$  *n*

*<proof>*

**lemma** *trim-const*: *height* *t*  $\leq$  *n*  $\implies$  *trim* *n* *t* = *t*

*<proof>*

**lemma** *height-trim-le'*: *n*  $\leq$  *height* *t*  $\implies$  *height* (*trim* *n* *t*) = *n*

*<proof>*

**lemma** *height-trim*: *height* (*trim* *n* *t*) = (*if* *n*  $\leq$  *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*  $\circ$  *trim* *n* = *trim* *n*

*<proof>*

**lemma** *trim-trim-Suc* [*simp*]: *trim* *n*  $\circ$  *trim* (*Suc* *n*) = *trim* *n*

*<proof>*

**lemma** *trim-trim* [*simp*]: *n*  $\leq$  *m*  $\implies$  *trim* *n*  $\circ$  *trim* *m* = *trim* *n*

*<proof>*

**lemma** *trim-eq-imp-trim-eq* [*simp*]:  $\llbracket n \leq m; trim\ m\ t = trim\ m\ u \rrbracket \implies trim\ n\ t = trim\ n\ u$

*<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*:  $\text{length} (\text{subtrees} (\text{trim} (\text{Suc } n) t)) = \text{length} (\text{subtrees } t)$

*<proof>*

**lemma** *trim-eq-Leaf*:  $\text{trim } n t = \text{Leaf} \implies n = 0 \vee t = \text{Leaf}$

*<proof>*

**lemma** *map-eq-imp-pairs-eq*:  $\text{map } f xs = \text{map } g ys \implies (\bigwedge x y. (x, y) \in \text{set} (\text{zip } xs ys) \implies f x = g y)$

*<proof>*

**lemma** *trim-eq-subtree-eq*:

**assumes**  $\text{trim} (\text{Suc } n) (\text{Node } ts) = \text{trim} (\text{Suc } n) (\text{Node } us)$

**shows**  $\bigwedge t u. (t, u) \in \text{set} (\text{zip } ts us) \implies \text{trim } n t = \text{trim } n u$

*<proof>*

**lemma** *pairs-eq-imp-map-eq*:

**assumes**  $\text{length } xs = \text{length } ys \wedge \forall (x, y) \in \text{set} (\text{zip } xs ys). f x = g y$

**shows**  $\text{map } f xs = \text{map } g ys$

*<proof>*

**lemma** *map-eq-iff-pairs-eq*:  $(\text{map } f xs = \text{map } g ys) =$

$(\text{length } xs = \text{length } ys \wedge (\forall (x, y) \in \text{set} (\text{zip } xs ys). f x = g y))$

*<proof>*

**lemma** *subtree-eq-trim-eq*:

**assumes**  $\text{length } ts = \text{length } us \wedge \forall (t, u) \in \text{set} (\text{zip } ts us). \text{trim } n t = \text{trim } n u$

**shows**  $\text{trim} (\text{Suc } n) (\text{Node } ts) = \text{trim} (\text{Suc } n) (\text{Node } us)$

*<proof>*

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

*<proof>*

**lemma** *is-subtree-index-trim*:

$\text{is-subtree-index} (\text{trim } n t) xs = (\text{is-subtree-index } t xs \wedge \text{length } xs \leq n)$

*<proof>*

**lemma** *subtree-trim*:  $\llbracket \text{is-subtree-index } t xs; \text{length } xs \leq n \rrbracket \implies$

$\text{trim } n t !t xs = \text{trim} (n - \text{length } xs) (t !t xs)$

*<proof>*

**lemma** *length-subtrees-trim*:  $\llbracket \text{is-subtree-index } t xs; \text{length } xs < n \rrbracket \implies$

$\text{length} (\text{subtrees} (\text{trim } n t !t xs)) = \text{length} (\text{subtrees} (t !t xs))$

*<proof>*

**lemma** *subtree-trim-Leaf*: **assumes**  $\text{is-subtree-index} (\text{trim } n t) xs \wedge t !t xs = \text{Leaf}$

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

## 4.2 The strict $\omega$ -category of pasting diagrams

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

**abbreviation**  $\delta$  **where**

$\delta \equiv trim$

**value**  $\delta\ 1$  ( $Node\ [Node\ [Leaf,\ Leaf,\ Leaf],\ Leaf,\ Node\ [Leaf]]$ )

**value**  $\delta\ 2$  ( $Node\ [Node\ [Node\ [Leaf,\ Leaf]],\ Node\ [Leaf,\ Leaf]]$ )

**abbreviation**  $PD :: nat \Rightarrow tree\ set$  **where**

$PD\ n \equiv \{t.\ height\ t \leq n\}$

**interpretation**  $pd: globular\ set\ PD\ \delta\ \delta$

⟨*proof*⟩

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**  $pd.s'\ m\ n\ t = trim\ n\ t$

⟨*proof*⟩

**lemma**  $s'$ -*eq-t'*:  $pd.s' = pd.t'$

⟨*proof*⟩

**lemma**  $t'$ -*eq-trim*: **assumes**  $n \leq m$   $height\ t \leq m$  **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**  $tree\ comp :: nat \Rightarrow tree \Rightarrow tree \Rightarrow tree$  **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*:  $pd.is-composable-pair\ m\ n\ t\ u =$   
 $(n < m \wedge height\ t \leq m \wedge height\ u \leq m \wedge composable-tree\ n\ t\ u)$   
 $\langle proof \rangle$

**lemma** *composable-tree-imp-composable-tree-subtrees*:  
 $composable-tree\ (Suc\ n)\ (Node\ ts)\ (Node\ us) \implies \forall (t, u) \in set\ (zip\ ts\ us). com-$   
 $posable-tree\ n\ t\ u$   
 $\langle proof \rangle$

**lemma** *composable-tree-nth-subtrees*:  
 $\llbracket composable-tree\ (Suc\ n)\ (Node\ ts)\ (Node\ us); i < length\ ts \rrbracket \implies composable-tree$   
 $n\ (ts!i)\ (us!i)$   
 $\langle proof \rangle$

**lemma** *is-composable-pair-imp-is-composable-pair-subtrees*:  
**assumes**  $pd.is-composable-pair\ (Suc\ m)\ (Suc\ n)\ (Node\ ts)\ (Node\ us)$   
**shows**  $\forall (t, u) \in set\ (zip\ ts\ us). pd.is-composable-pair\ m\ n\ t\ u$   
 $\langle proof \rangle$

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

**lemma** *height-tree-comp-le*:  $\llbracket height\ t \leq m; height\ u \leq m \rrbracket \implies height\ (tree-comp\ n$   
 $t\ u) \leq m$   
 $\langle proof \rangle$

**lemma** *nth-map2 [simp]*:  $\llbracket n < length\ xs; n < length\ ys \rrbracket \implies map2\ f\ xs\ ys\ !\ n = f$   
 $(xs\ !\ n)\ (ys\ !\ n)$   
 $\langle proof \rangle$

**lemma** *trim-tree-comp1*:  $composable-tree\ n\ t\ u \implies trim\ n\ (tree-comp\ n\ t\ u) =$   
 $trim\ n\ t$   
 $\langle proof \rangle$

**lemma** *trim-tree-comp2*:  $composable-tree\ n\ t\ u \implies trim\ n\ (tree-comp\ n\ t\ u) =$   
 $trim\ n\ u$   
 $\langle proof \rangle$

**lemma** *map2-map-map'*:  $map2\ f\ (map\ g\ xs)\ (map\ h\ ys) = map\ (\lambda(x, y). f\ (g\ x)$   
 $(h\ y))\ (zip\ xs\ ys)$   
 $\langle proof \rangle$

**lemma** *trim-tree-comp-commute*:  $trim\ m\ (tree-comp\ n\ t\ u) = tree-comp\ n\ (trim\ m$   
 $t)\ (trim\ m\ u)$   
 $\langle proof \rangle$

**interpretation** *pd-pre-cat*: *pre-strict-omega-category*  $PD\ \delta\ \delta\ \lambda\ m. tree-comp\ \lambda\ n.$

*id*  
*<proof>*

**lemma** *tree-comp-assoc*:  $tree\text{-}comp\ n\ (tree\text{-}comp\ n\ t\ u)\ v = tree\text{-}comp\ n\ t\ (tree\text{-}comp\ n\ u\ v)$   
*<proof>*

**lemma** *i'-eq-id*:  $n \leq m \implies pd\text{-}pre\text{-}cat.i'\ m\ n = id$   
*<proof>*

**lemma** *composable-tree-trim1*:  $n \leq m \implies composable\text{-}tree\ n\ (trim\ m\ t)\ t$   
*<proof>*

**lemma** *composable-tree-trim2*:  $n \leq m \implies composable\text{-}tree\ n\ t\ (trim\ m\ t)$   
*<proof>*

**lemma** *tree-comp-trim1*:  $tree\text{-}comp\ n\ (trim\ n\ t)\ t = t$   
*<proof>*

**lemma** *tree-comp-trim2*:  $tree\text{-}comp\ n\ t\ (trim\ n\ t) = t$   
*<proof>*

**lemma** *tree-comp-exchange*:  
 $\llbracket q < p; composable\text{-}tree\ p\ y'\ y; composable\text{-}tree\ p\ x'\ x; composable\text{-}tree\ q\ y'\ x'; composable\text{-}tree\ q\ y\ x \rrbracket \implies$   
 $tree\text{-}comp\ q\ (tree\text{-}comp\ p\ y'\ y)\ (tree\text{-}comp\ p\ x'\ x) =$   
 $tree\text{-}comp\ p\ (tree\text{-}comp\ q\ y'\ x')\ (tree\text{-}comp\ q\ y\ x)$   
*<proof>*

**interpretation** *pd-cat'*: *strict-omega-category PD  $\delta\ \delta\ \lambda\ m$ . tree-comp  $\lambda\ n$ . id*  
*<proof>*

**end**

## 5 Acknowledgements

The work has been jointly supported by the Cambridge Mathematics Place-ments (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.