Overview Code CW-Complexes

Hannah Scholz

April 2024

1 Introduction

This is a place to keep track of all the statements I already have. I will document what still needs to be done, what I am stuck on and why and what I have questions about.

Contents

Todo list

2 General

Sources for types: Theorem proving in lean, Reference manual, Hitchhiker's guide to proof verification (sections 4.6, 13.3, 13.4, 13.5)

Convert CWComplex and Subcomplex to class so that CW-Complex can be an instance. Try if it works.

Provide basic examples in a seperate file. For example spheres (both different types) and intervals.

3 File: auxiliary

- Lemma *aux1*

```
lemma aux1 (l : ℕ) {X : Type*} {s : ℕ →  Type*} (Y : (m : ℕ) → s m → Set X):
    ⋃ m, ⋃ (_ : m < Nat.succ l), ⋃ j, Y m j =
    (⋃ (j : s l), Y l j) ∪ ⋃ m, ⋃ (_ : m < l), ⋃ j, Y m j
```

- Lemma *ENat.coe_lt_top*

```
lemma ENat.coe_lt_top {n : ℕ} : ↑n < (⊤ : ℕ∞)
```

- Lemma *isClosed_inter_singleton*

```
lemma isClosed_inter_singleton {X : Type*} [TopologicalSpace X] [T1Space X] {A : Set X}
{a : X} : IsClosed (A ∩ {a})
```

- Lemma *sphere_zero_dim_empty*

```
lemma sphere_zero_dim_empty {X : Type*} {h : PseudoMetricSpace (Fin 0 → X)}:
  (Metric.sphere ![] 1 : Set (Fin 0 → X)) = ∅
```

- Definition *kification*

```
def kification (X : Type*) := X
```

- Instance *instkification*

```
instance instkification {X : Type*} [t : TopologicalSpace X] :
TopologicalSpace (kification X) where
  IsOpen A := IsOpen A := ∀ (B : t.Compacts), ∃ (C: t.Opens), A ∩ B.1 = C.1 ∩ B.1
  isOpen_univ := sorry
  isOpen_inter := sorry
  isOpen_sUnion := sorry
```

- Definition *tokification*

```
def tokification {X : Type*} : X ≃ kification X :=
⟨fun x ↦ x, fun x ↦ x, fun _ ↦ rfl, fun _ ↦ rfl⟩
```

- Definition *fromkification*

```
def fromkification {X : Type*} : kification X ≃ X :=
⟨fun x ↦ x, fun x ↦ x, fun _ ↦ rfl, fun _ ↦ rfl⟩
```

- Lemma *continuous_fromkification*

```
lemma continuous_fromkification {X : Type*} [t : TopologicalSpace X] :
Continuous (fromkification X)
```

- Lemma *isopenmap_tokification*

```
lemma isopenmap_tokification {X : Type*} [t: TopologicalSpace X] :
IsOpenMap (tokification X)
```

- Definition *EquivFinReal*

```
def EquivFinReal (m n : ℕ) : (Fin m → ℝ) × (Fin n → ℝ) ≃ (Fin (m + n) → ℝ) :=
Equiv.trans (Equiv.sumArrowEquivProdArrow _ _ _).symm
  (Equiv.arrowCongr finSumFinEquiv (Equiv.refl _))
```

- Definition *HomeomorphFinReal1*

```
def HomeomorphFinReal1 (m n : ℕ) : (Fin m → ℝ) × (Fin n → ℝ) ≃ₜ (Fin m ⊕ Fin n → ℝ) where
  toFun := (Equiv.sumArrowEquivProdArrow _ _ _).symm
  invFun := Equiv.sumArrowEquivProdArrow _ _ _
  left_inv := sorry
  right_inv := sorry
  continuous_toFun := sorry
  continuous_invFun := sorry
```

- Definition *HomeomorphFinReal2*

```
def HomeomorphFinReal2 (m n : ℕ) : (Fin m ⊕ Fin n → ℝ) ≃ₜ (Fin (m + n) → ℝ) where
  toFun := Equiv.arrowCongr finSumFinEquiv (Equiv.refl _)
  invFun := (Equiv.arrowCongr finSumFinEquiv (Equiv.refl _)).symm
  left_inv := sorry
  right_inv := sorry
```

- Definition *HomeomorphFinReal*

```
def HomeomorphFinReal (m n : ℕ) : (Fin m → ℝ) × (Fin n → ℝ) ≃ₜ (Fin (m + n) → ℝ) :=
  Homeomorph.trans (HomeomorphFinReal1 _ _) (HomeomorphFinReal2 _ _)
```

- Definition *IsometryEquivFinMap1*

```
def IsometryEquivFinMap1 {X: Type*} [PseudoEMetricSpace X] (m n : ℕ) :
(Fin m → X) × (Fin n → X) ≃ᵢ (Fin m ⊕ Fin n → X) where
  toFun := (Equiv.sumArrowEquivProdArrow _ _ _).symm
  invFun := Equiv.sumArrowEquivProdArrow _ _ _
  left_inv := sorry
  right_inv := sorry
  isometry_toFun := sorry
```

- Definition *IsometryEquivFinMap2*

```
def IsometryEquivFinMap2 {X: Type*} [PseudoEMetricSpace X] (m n : ℕ) :
(Fin m ⊕ Fin n → X) ≃ᵢ (Fin (m + n) → X) where
  toFun := Equiv.arrowCongr finSumFinEquiv (Equiv.refl _)
  invFun := (Equiv.arrowCongr finSumFinEquiv (Equiv.refl _)).symm
  left_inv := sorry
  right_inv := sorry
  isometry_toFun := sorry
```

- Definition *IsometryEquivFinMap*

```
def IsometryEquivFinMap {X : Type*} [MetricSpace X] (m n : ℕ) :
(Fin m → X) × (Fin n → X) ≃ᵢ (Fin (m + n) → X) :=
  IsometryEquiv.trans (IsometryEquivFinMap1 _ _) (IsometryEquivFinMap2 _ _)
```

4 File: Definition

- Assumption `variable {X : Type*} [t : TopologicalSpace X]`
- Structure *CWCcomplex*

```
structure CWComplex.{u} {X : Type u} [TopologicalSpace X] (C : Set X) where
  cell (n : ℕ) : Type u
  map (n : ℕ) (i : cell n) : PartialEquiv (Fin n → ℝ) X
  source_eq (n : ℕ) (i : cell n) : (map n i).source = closedBall 0 1
  cont (n : ℕ) (i : cell n) : ContinuousOn (map n i) (closedBall 0 1)
  cont_symm (n : ℕ) (i : cell n) : ContinuousOn (map n i).symm (map n i).target
  pairwiseDisjoint :
    (univ : Set (Σ n, cell n)).PairwiseDisjoint (fun ni ↦ map ni.1 ni.2 '' ball 0 1)
  mapsto (n : ℕ) (i : cell n) : ∃ I : Π m, Finset (cell m),
    MapsTo (map n i) (sphere 0 1) (⋃ (m < n) (j ∈ I m), map m j '' closedBall 0 1)
  closed (A : Set X) (asubc : A ⊆ ↑C) :
    IsClosed A ↔ ∀ n j, IsClosed (A ∩ map n j '' closedBall 0 1)
  union : ⋃ (n : ℕ) (j : cell n), map n j '' closedBall 0 1 = C
```

- Assumption `variable [T2Space X] {C : Set X} (hC : CWComplex C)`
- Definition *levelaux*

```
def levelaux (n : ℕ∞) : Set X :=
  ⋃ (m : ℕ) (hm : m < n) (j : hC.cell m), hC.map m j '' closedBall 0 1
```

- Definition *level*

```
def level (n : ℕ∞) : Set X :=
  hC.levelaux (n + 1)
```

- Lemma *levelaux_eq_level_sub_one*

```
lemma levelaux_eq_level_sub_one {n : ℕ∞} (npos : n ≠ 0) : hC.levelaux n = hC.level (n - 1)
```

- Lemma *levelaux_zero_eq_empty*

```
lemma levelaux_zero_eq_empty : hC.levelaux 0 = ∅
```

Add CW-Complex of finite type

- Class *Finite*

```
class Finite.{u} {X : Type u} [TopologicalSpace X] (C : Set X) (cwcomplex : CWComplex C) :
Prop where
  finitelevels : ∀ᶠ n in Filter.atTop, IsEmpty (cwcomplex.cell n)
  finitecells (n : ℕ) : Finite (cwcomplex.cell n)
```

- Structure *Subcomplex*

```
structure Subcomplex (E : Set X) where
  I : Π n, Set (hC.cell n)
  closed : IsClosed E
  union : E = ⋃ (n : ℕ) (j : I n), hC.map n j '' ball 0 1
```

- Class *Subcomplex.Finite*

```
class Subcomplex.Finite (E : Set X) (subcomplex: hC.Subcomplex E) : Prop where
  finitelevels : ∀? n in Filter.atTop, IsEmpty (hC.cell n)
  finitecells (n : ℕ) : _root_.Finite (hC.cell n)
```

- Lemma *levelaux_top*

```
@[simp] lemma levelaux_top : hC.levelaux ⊤ = C
```

- Lemma *level_top*

```
@[simp] lemma level_top : hC.level ⊤ = C
```

- Lemma *iUnion_map_sphere_subset_levelaux*

```
lemma iUnion_map_sphere_subset_levelaux (l : ℕ) :
  ⋃ (j : hC.cell l), ↑(hC.map l j) '' sphere 0 1 ⊆ hC.levelaux l
```

- Lemma *iUnion_map_sphere_subset_level*

```
lemma iUnion_map_sphere_subset_level (l : ℕ) :
  ⋃ (j : hC.cell l), ↑(hC.map l j) '' sphere 0 1 ⊆ hC.levelaux l
```

- Lemma *levelaux_subset_levelaux_of_le*

```
lemma levelaux_subset_levelaux_of_le {n m : ℕ∞} (h : m ≤ n) :
  hC.levelaux m ⊆ hC.levelaux n
```

- Lemma *level_subset_level_of_le*

```
lemma level_subset_level_of_le {n m : ℕ∞} (h : m ≤ n) : hC.level m ⊆ hC.level n
```

- Lemma *iUnion_levelaux_eq_levelaux*

```
lemma iUnion_levelaux_eq_levelaux (n : ℕ∞) :
  ⋃ (m : ℕ) (hm : m < n + 1), hC.levelaux m = hC.levelaux n
```

- Lemma *iUnion_level_eq_level*

```
lemma iUnion_level_eq_level (n : ℕ∞) : ⋃ (m : ℕ) (hm : m < n + 1), hC.level m = hC.level n
```

- Lemma *iUnion_ball_eq_levelaux*

```
lemma iUnion_ball_eq_levelaux (n : ℕ∞) :
  ⋃ (m : ℕ) (hm : m < n) (j : hC.cell m), hC.map m j '' ball 0 1 = hC.levelaux n
```

- Lemma *iUnion_ball_eq_level*

```
lemma iUnion_ball_eq_level (n : ℕ∞) :
  ⋃ (m : ℕ) (hm : m < n + 1) (j : hC.cell m), hC.map m j '' ball 0 1 = hC.level n
```

- Lemma *mapsto_sphere_levelaux*

```
lemma mapsto_sphere_levelaux (n : ℕ) (j : hC.cell n) (nnezero : n ≠ 0) :
  MapsTo (hC.map n j) (sphere 0 1) (hC.levelaux  n)
```

- Lemma *mapsto_sphere_level*

```
lemma mapsto_sphere_level (n : ℕ) (j : hC.cell n) (nnezero : n ≠ 0) :
  MapsTo (hC.map n j) (sphere 0 1) (hC.level (Nat.pred n))
```

- Lemma *exists_mem_ball_of_mem_levelaux*

```
lemma exists_mem_ball_of_mem_levelaux {n : ℕ∞} {x : X} (xmemlvl : x ∈ hC.levelaux n) :
  ∃ (m : ℕ) (_ : m < n) (j : hC.cell m), x ∈ ↑(hC.map m j) '' ball 0 1
```

- Lemma *exists_mem_ball_of_mem_level*

```
lemma exists_mem_ball_of_mem_level {n : ℕ∞} {x : X} (xmemlvl : x ∈ hC.level n) :
  ∃ (m : ℕ) (_ : m ≤ n) (j : hC.cell m), x ∈ ↑(hC.map m j) '' ball 0 1
```

- Lemma *levelaux_inter_image_closedBall_eq_levelaux_inter_image_sphere*

```
lemma levelaux_inter_image_closedBall_eq_levelaux_inter_image_sphere
{n : ℕ∞} {m : ℕ}{j : hC.cell m} (nlem : n ≤ m) :
  hC.levelaux n ∩ ↑(hC.map m j) '' closedBall 0 1 =
  hC.levelaux n ∩ ↑(hC.map m j) '' sphere 0 1
```

- Lemma *level_inter_image_closedBall_eq_level_inter_image_sphere*

```
lemma level_inter_image_closedBall_eq_level_inter_image_sphere
{n : ℕ∞} {m : ℕ}{j : hC.cell m} (nltm : n < m) :
  hC.level n ∩ ↑(hC.map m j) '' closedBall 0 1 =
  hC.level n ∩ ↑(hC.map m j) '' sphere 0 1
```

- Lemma *isClosed_map_sphere*

```
lemma isClosed_map_sphere {n : ℕ} {i : hC.cell n} : IsClosed (hC.map n i '' sphere 0 1)
```

- Lemma *isClosed_inter_sphere_succ_of_le_isClosed_inter_closedBall*

```
lemma isClosed_inter_sphere_succ_of_le_isClosed_inter_closedBall
{A : Set X} {n : ℕ}
(hn : ∀ m ≤ n, ∀ (j : hC.cell m), IsClosed (A ∩ ↑(hC.map m j) '' closedBall 0 1)) :
∀ (j : hC.cell (n + 1)), IsClosed (A ∩ ↑(hC.map (n + 1) j) '' sphere 0 1)
```

- Lemma *isClosed_map_closedBall*

```
lemma isClosed_map_closedBall (n : ℕ) (i : hC.cell n) :
  IsClosed (hC.map n i '' closedBall 0 1)
```

- Lemma *isClosed*

```
lemma isClosed : IsClosed C
```

- Lemma *levelaux_succ_eq_levelaux_union_iUnion*

```
lemma levelaux_succ_eq_levelaux_union_iUnion (n : ℕ) :
  hC.levelaux (↑n + 1) = hC.levelaux ↑n ∪ ⋃ (j : hC.cell ↑n), hC.map ↑n j '' closedBall 0 1
```

- Lemma *level_succ_eq_level_union_iUnion*

```
lemma level_succ_eq_level_union_iUnion (n : ℕ) :
  hC.level (↑n + 1) =
  hC.level ↑n ∪ ⋃ (j : hC.cell (↑n + 1)), hC.map (↑n + 1) j '' closedBall 0 1
```

- Lemma *map_closedBall_subset_levelaux*

```
lemma map_closedBall_subset_levelaux (n : ℕ) (j : hC.cell n) :
  (hC.map n j) '' closedBall 0 1 ⊆ hC.levelaux (n + 1)
```

- Lemma *map_closedBall_subset_level*

```
lemma map_closedBall_subset_level (n : ℕ) (j : hC.cell n) :
  (hC.map n j) '' closedBall 0 1 ⊆ hC.level n
```

- Lemma *map_ball_subset_levelaux*

```
lemma map_ball_subset_levelaux (n : ℕ) (j : hC.cell n) :
  (hC.map n j) '' ball 0 1 ⊆ hC.levelaux (n + 1)
```

- Lemma *map_ball_subset_level*

```
lemma map_ball_subset_level (n : ℕ) (j : hC.cell n) :
  (hC.map n j) '' ball 0 1 ⊆ hC.level n
```

- Lemma *map_ball_subset_complex*

```
lemma map_ball_subset_complex (n : ℕ) (j : hC.cell n) :
  (hC.map n j) '' ball 0 1 ⊆ C
```

- Lemma *map_ball_subset_map_closedball*

```
lemma map_ball_subset_map_closedball {n : ℕ} {j : hC.cell n} :
  hC.map n j '' ball 0 1 ⊆ hC.map n j '' closedBall 0 1
```

- Lemma *closure_map_ball_eq_map_closedball*

```
lemma closure_map_ball_eq_map_closedball {n : ℕ} {j : hC.cell n} :
  closure (hC.map n j '' ball 0 1) = hC.map n j '' closedBall 0 1
```

- Lemma *not_disjoint_equal*

```
lemma not_disjoint_equal {n : ℕ} {j : hC.cell n} {m : ℕ} {i : hC.cell m}
(notdisjoint: ¬ Disjoint (↑(hC.map n j) '' ball 0 1) (↑(hC.map m i) '' ball 0 1)) :
  (⟨n, j⟩ : (Σ n, hC.cell n)) = ⟨m, i⟩
```

- Lemma *map_closedBall_inter_map_closedBall_eq_map_ball_inter_map_ball_of_le*

Do the proof.

```
lemma map_closedBall_inter_map_closedBall_eq_map_ball_inter_map_ball_of_le
{n : ℕ} {j : hC.cell n} {m : ℕ} {i : hC.cell m} (ne : (⟨n, j⟩ : (Σ n, hC.cell n)) ≠ ⟨m, i⟩)
(mlen : m ≤ n) :
hC.map n j '' closedBall 0 1 ∩ hC.map m i '' closedBall 0 1 = hC.map n j '' sphere 0 1 ∩ hC.map
```

could this proof be simplified using ‘exists_mem_ball_of_mem_level‘?

- Lemma *mapsto'*

```
lemma mapsto' (n : ℕ) (i : hC.cell n) : ∃ I : Π m, Finset (hC.cell m),
  MapsTo (hC.map n i) (sphere 0 1) (⋃ (m < n) (j ∈ I m), hC.map m j '' ball 0 1)
```

- Lemma *mapsto"*

Do the proof.

```
lemma mapsto'' (n :  ℕ) (i : hC.cell n) : _root_.Finite (Σ (m : ℕ),
{j : hC.cell m // ¬ Disjoint (↑(hC.map n i) '' sphere 0 1 ) (↑(hC.map m j) '' ball 0 1)} )
```

5 File: Constructions

- Assumption:

```
variable {X : Type*} [t : TopologicalSpace X] [T2Space X] {C : Set X} (hC : CWComplex C)
```

- Definition *CWComplex_level*

```
def CWComplex_level (n : ℕ∞) : CWComplex (hC.level n) where
  cell l := {x : hC.cell l // l < n + 1}
  map l i := hC.map l i
  source_eq l i := sorry
  cont l i := sorry
  cont_symm l i := sorry
  pairwiseDisjoint := sorry
  mapsto l i := sorry
  closed A := sorry
  union := sorry
```

- Assumption `variable {D : Set X} (hD : CWComplex D)`

- Definition *CWComplex_disjointUnion*

```
def CWComplex_disjointUnion (disjoint : Disjoint C D) : CWComplex (C ∪ D) where
  cell n := Sum (hC.cell n) (hD.cell n)
  map n i :=
    match i with
    | Sum.inl x => hC.map n x
    | Sum.inr x => hD.map n x
  source_eq n i := sorry
  cont n i := sorry
  cont_symm n i := sorry
  pairwiseDisjoint := sorry
  mapsto n i := sorry
  closed A := sorry
  union := sorry
```

- Definition *CWComplex_subcomplex*

```
def CWComplex_subcomplex (E : Set X) (subcomplex: Subcomplex hC E) : CWComplex E where
  cell n := subcomplex.I n
  map n i := hC.map n i
  source_eq n i := sorry
  cont n i := sorry
  cont_symm n i := sorry
  pairwiseDisjoint := sorry
  mapsto n i := sorry
  closed A := sorry
  union := sorry
```

- Assumption

```
variable {X : Type*} {Y : Type*} [t1 : TopologicalSpace X] [t2 : TopologicalSpace Y]
[T2Space X] [T2Space Y] {C : Set X} {D : Set Y} (hC : @CWComplex X t1 C)
(hD : @CWComplex Y t2 D)
```

- Definition *Prodkification*

```
def Prodkification X Y := kification (X × Y)
```

- Notation ×ˢ

```
infixr:35 " ×ˢ "  => Prodkification
```

Do a composition of to maps, first an equivalence of (Fin m1 -> R) x (Fin m2 -> R) (in auxiliary) and then do Prodmap

- Definition *prodmap*

```
def prodmap (m1 : ℕ) (m2 : ℕ) (c1 : hC.cell m1) (c2 : hD.cell m2) :
(Fin (m1 + m2) → ℝ) → (X ×ˢ Y) :=
fun x ↦ ⟨hC.map m1 c1 (fun y ↦ x ⟨y.1, lt_of_lt_of_le y.2 (Nat.le_add_right m1 m2)⟩),
hD.map m2 c2 (fun y ↦ x ⟨m1 +y.1, add_lt_add_left y.2 m1⟩)⟩
```

- Definition *prodinvmap*

```
def prodinvmap (m1 : ℕ) (m2 : ℕ) (c1 : hC.cell m1) (c2 : hD.cell m2) :
(X ×ˢ Y) → (Fin (m1 + m2) → ℝ) :=
fun ⟨x, y⟩ ↦ (fun l ↦ if llt : l < m1 then (hC.map m1 c1).invFun x ⟨l, llt⟩
else (hD.map m2 c2).invFun y ⟨l - m1, Nat.sub_lt_left_of_lt_add (not_lt.1 llt) l.2⟩)
```

Finish this

- Definition *mapprodkification*

```
def mapprodkification (m1 : ℕ) (m2 : ℕ) (c1 : hC.cell m1) (c2 : hD.cell m2):
  PartialEquiv (Fin (m1 + m2) → ℝ) (X ×ˢ Y) where
    toFun :=  prodmap hC hD m1 m2 c1 c2
    invFun := prodinvmap hC hD m1 m2 c1 c2
    source := closedBall 0 1
    target := (prodmap hC hD m1 m2 c1 c2) '' closedBall 0 1
    map_source' := sorry
    map_target' := sorry
    left_inv' := sorry
    right_inv' := sorry
```

Make this work and do the proofs!

- Definition *CWComplex_product*

```
instance CWComplex_product : @CWComplex (X ×ˢ Y) instprodkification (C ×ˢ D) where
  cell n := (Σ' (m : ℕ) (l : ℕ) (hml : m + l = n), hC.cell m × hD.cell l)
  map n i := sorry
  source_eq n i := sorry
  cont n i := sorry
  cont_symm := sorry
  pairwiseDisjoint := sorry
  mapsto n i := sorry
  closed A := sorry
  union := sorry
```

Define Quotients.

6 File: Lemmas

- Assumption

```
variable {X : Type*} [t : TopologicalSpace X] [T2Space X] {C : Set X} (hC : CWComplex C)
```

- Lemma *isClosed_level*

```
lemma isClosed_level (n : ℕ∞) : IsClosed (hC.level n)
```

- Lemma *isClosed_levelaux*

```
lemma isClosed_levelaux (n : ℕ∞) : IsClosed (hC.levelaux n)
```

- Lemma *closed_iff_inter_levelaux_closed*

```
lemma closed_iff_inter_levelaux_closed (A : Set X) (asubc : A ⊆ C) :
IsClosed A ↔ ∀ (n : ℕ), IsClosed (A ∩ hC.levelaux n)
```

- Lemma *inter_levelaux_succ_closed_iff_inter_levelaux_closed_and_inter_closedBall_closed*

```
lemma inter_levelaux_succ_closed_iff_inter_levelaux_closed_and_inter_closedBall_closed
(A : Set X) : IsClosed (A ∩ hC.levelaux (Nat.succ n)) ↔
IsClosed (A ∩ hC.levelaux n) ∧ ∀ (j : hC.cell n), IsClosed (A ∩ hC.map n j '' closedBall 0 1)
```

- Lemma *isDiscrete_level_zero*

```
lemma isDiscrete_level_zero {A : Set X} : IsClosed (A ∩ hC.level 0)
```

- Lemma *compact_inter_finite*

```
lemma compact_inter_finite (A : t.Compacts) :
  _root_.Finite (Σ (m : ℕ), {j : hC.cell m // ¬ Disjoint A.1 (↑(hC.map m j) '' ball 0 1)})
```

Make the following lemmata work with new definitions

Use CW-Complex_ subcomplex

- Lemma *iUnion_subcomplex*

```
lemma iUnion_subcomplex (J : Type u) (I : J → Π n, Set (hC.cell n))
(cw : ∀ (l : J), CWComplex (⋃ (n : ℕ) (j : I l n), hC.map n j '' ball 0 1))
  : CWComplex (⋃ (l : J) (n : ℕ) (j : I l n), hC.map n j '' ball 0 1)
```

Do the proofs.

- Lemma *finite_iUnion_finitesubcomplex*

```
lemma finite_iUnion_finitesubcomplex (m : ℕ) (I : Fin m → Π n, Set (hC.cell n))
(fincw : ∀ (l : Fin m), FiniteCWComplex (⋃ (n : ℕ) (j : I l n), hC.map n j '' ball 0 1)) :
FiniteCWComplex (⋃ (l : Fin m) (n : ℕ) (j : I l n), hC.map n j '' ball 0 1) where
  cwcomplex := sorry
  finitelevels := sorry
  finitecells := sorry
```

See Hatcher p. 522. I don't really want to do that know so I'll just leave it here for now.

Make this work.

- Definition *open_neighbourhood_aux*

```
def open_neighbourhood_aux (ε : (n : ℕ) → (hC.cell n) → {x : ℝ // x > 0})
(A : Set X) (AsubC : A ⊆ C) (n : ℕ): Set X :=
  match n with
  | 0 => A ∩ hC.level 0
  | Nat.succ m => sorry
```