The Classical Seifert–van Kampen Theorem

Arthur F. Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

September 9, 2026

Abstract

This entry formalizes a quotient-oriented proof of the classical Seifert–van Kampen theorem in Isabelle/HOL. For open subsets U and V of a topological space with $x_0 \in U \cap V$ and path-connected intersection, it proves that the fundamental group of $U \cup V$ at x_0 is in bijection with the carrier-based amalgamated free product of the fundamental groups of U and V over the fundamental group of $U \cap V$. The development also provides reusable infrastructure for explicit path homotopies, pushout-style quotients, free-product words, carrier-based amalgamated free products, and the abstract encode/decode interface used to organize the final argument. AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

1 Overview

The main unconditional result of the entry is the theorem `classical_seifert_van_kampen_bij_betw` in `Classical_Seifert_Van_Kampen.thy`. It establishes the classical open-union form of Seifert–van Kampen through a quotient-level bijection between the fundamental group of $U \cup V$ and the carrier-based amalgamated free product assembled from the fundamental groups of U, V, and $U \cap V$.

Supporting theories package the quotient-level encode/decode interface and the topological pushout constructions used by the classical proof. These auxiliary layers are internal to the entry; the completed headline result remains the classical open-union theorem above.

2 Development structure

The theories are organized in three layers:

- *Quotients and pushouts:* `Equivalence_Quotients.thy`, `Pushout_Scaffold.thy`, `Binary_Sum_Topology.thy`, and `Topological_`

`Pushout_Scaffold.thy` isolate the generic equivalence-class and topological-pushout constructions.

- *The algebraic target:* `Free_Product_Words.thy`, `Amalgamated_Free_Product.thy`, `Carrier_Group_Scaffold.thy`, `Carrier_Amalgamated_Free_Product.thy`, and `Carrier_Amalgamated_Free_Product_Eval.thy` provide the word calculus, carrier-group locales, and evaluation map for the carrier-based amalgamated free product.
- *The topological argument:* `Explicit_Path_Homotopy_Scaffold.thy`, `Explicit_Fundamental_Group_Scaffold.thy`, and `Fundamental_Group_Scaffold.thy` develop the quotient view of the fundamental group, while `Seifert_Van_Kampen_Scaffold.thy`, `Topological_Seifert_Van_Kampen.thy`, and `Classical_Seifert_Van_Kampen.thy` package the abstract interface and its classical open-union instantiation.

3 Sources

The informal mathematics follows the classical theorem due to Seifert and van Kampen [3, 4] and standard textbook presentations in algebraic topology, especially Hatcher [2] and Brown [1].

Contents

1 Overview **1**

2 Development structure **1**

3 Sources **2**

4 Equivalence classes as quotient infrastructure **3**

5 Free-product words **4**

6 Amalgamation quotients of free-product words **7**

7 Carrier-based amalgamated free products **9**

8 Groups on carriers **14**

9 Evaluating carrier amalgamated free-product words **15**

10 Explicit-topology paths and homotopies **18**

11 Fundamental-group quotients **24**

12 Explicit-topology fundamental-group quotients **31**

13 Pushout-style quotients of disjoint sums **36**

14 Conditional Seifert–van Kampen interface **38**

15 Classical Seifert–van Kampen for open unions **40**
15.1 Carrier-side setup 41
15.2 Partitions and encoded loop words 45
15.3 Homotopy invariance of the partition encoding 68
15.4 Encoding, decoding, and the final bijection 68

16 Binary coproduct topology **70**

17 Topological pushouts **72**

18 Concrete decode data for topological pushouts **74**

19 Top-level entry point **78**

theory *Equivalence-Quotients*
imports *Main*
begin

4 Equivalence classes as quotient infrastructure

The later pushout and fundamental-group constructions are phrased in terms of explicit equivalence classes and quotient carriers. This small theory keeps that quotient infrastructure elementary and reusable, so subsequent theories can concentrate on the specific relations that matter for Seifert–van Kampen.

definition *equiv-class* :: (*'a* => *'a* => *bool*) => *'a* => *'a set* **where**
equiv-class R x = {*y*. *R y x*}

definition *quotient-space* :: (*'a* => *'a* => *bool*) => *'a set set* **where**
quotient-space R = *equiv-class R* ‘ *UNIV*

lemma *quotient-spaceI*:
equiv-class R x ∈ *quotient-space R*
⟨*proof*⟩

lemma *equiv-class-iff* [*simp*]:
y ∈ *equiv-class R x* ⟷ *R y x*
⟨*proof*⟩

locale *equivalence-relation* =
fixes *R* :: *'a* => *'a* => *bool*
assumes *refl* [*intro*!, *simp*]: *R x x*

and *sym*: $R\ x\ y \Longrightarrow R\ y\ x$
and *transitive*: $R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
begin

lemma *equiv-class-eqI*:
assumes $R\ x\ y$
shows *equiv-class* $R\ x$ = *equiv-class* $R\ y$
⟨*proof*⟩

lemma *equiv-class-eq-iff*:
equiv-class $R\ x$ = *equiv-class* $R\ y \longleftrightarrow R\ x\ y$
⟨*proof*⟩

lemma *quotient-spaceE*:
assumes $Q \in$ *quotient-space* R
obtains x **where** Q = *equiv-class* $R\ x$
⟨*proof*⟩

end

end
theory *Free-Product-Words*
imports *Main*
begin

5 Free-product words

Encodings of loops in the Seifert–van Kampen argument are expressed as words that alternate between generators from the left and right factors. This theory provides the basic word combinators, reversal operations, and reduction-oriented bookkeeping used by the later amalgamation quotients.

datatype ('*a*, '*b*) *free-product-word* =
WordNil
| *WordLeft* '*a* ('*a*, '*b*) *free-product-word*
| *WordRight* '*b* ('*a*, '*b*) *free-product-word*

primrec *fpw-concat* ::
('*a*, '*b*) *free-product-word* => ('*a*, '*b*) *free-product-word* => ('*a*, '*b*) *free-product-word*
where
fpw-concat WordNil w2 = *w2*
| *fpw-concat* (*WordLeft a rest*) *w2* = *WordLeft a* (*fpw-concat rest w2*)
| *fpw-concat* (*WordRight b rest*) *w2* = *WordRight b* (*fpw-concat rest w2*)

primrec *fpw-length* :: ('*a*, '*b*) *free-product-word* => *nat* **where**
fpw-length WordNil = *0*
| *fpw-length* (*WordLeft a rest*) = *Suc* (*fpw-length rest*)
| *fpw-length* (*WordRight b rest*) = *Suc* (*fpw-length rest*)

primrec *fpw-reverse* :: ('*a*, '*b*) *free-product-word* => ('*a*, '*b*) *free-product-word*
where
  fpw-reverse WordNil = *WordNil*
| *fpw-reverse* (*WordLeft a rest*) = *fpw-concat* (*fpw-reverse rest*) (*WordLeft a WordNil*)
| *fpw-reverse* (*WordRight b rest*) = *fpw-concat* (*fpw-reverse rest*) (*WordRight b WordNil*)

primrec *fpw-map* ::
  ('*a* => '*c*) => ('*b* => '*d*) => ('*a*, '*b*) *free-product-word* => ('*c*, '*d*) *free-product-word*
where
  fpw-map fl fr WordNil = *WordNil*
| *fpw-map fl fr* (*WordLeft a rest*) = *WordLeft* (*fl a*) (*fpw-map fl fr rest*)
| *fpw-map fl fr* (*WordRight b rest*) = *WordRight* (*fr b*) (*fpw-map fl fr rest*)

lemma *fpw-concat-nil-right* [*simp*]:
  fpw-concat w WordNil = *w*
  ⟨*proof*⟩

lemma *fpw-concat-assoc* [*simp*]:
  fpw-concat (*fpw-concat u v*) *w* = *fpw-concat u* (*fpw-concat v w*)
  ⟨*proof*⟩

lemma *fpw-reverse-concat* [*simp*]:
  fpw-reverse (*fpw-concat u v*) = *fpw-concat* (*fpw-reverse v*) (*fpw-reverse u*)
  ⟨*proof*⟩

lemma *fpw-reverse-reverse* [*simp*]:
  fpw-reverse (*fpw-reverse w*) = *w*
  ⟨*proof*⟩

fun *fpw-reduced* :: ('*a*, '*b*) *free-product-word* => *bool* **where**
  fpw-reduced WordNil = *True*
| *fpw-reduced* (*WordLeft a WordNil*) = *True*
| *fpw-reduced* (*WordLeft a* (*WordLeft b rest*)) = *False*
| *fpw-reduced* (*WordLeft a* (*WordRight b rest*)) = *fpw-reduced* (*WordRight b rest*)
| *fpw-reduced* (*WordRight b WordNil*) = *True*
| *fpw-reduced* (*WordRight b* (*WordRight c rest*)) = *False*
| *fpw-reduced* (*WordRight b* (*WordLeft a rest*)) = *fpw-reduced* (*WordLeft a rest*)

fun *fpw-inverse* ::
  ('*a*::*group-add*, '*b*::*group-add*) *free-product-word* =>
    ('*a*, '*b*) *free-product-word*
where
  fpw-inverse WordNil = *WordNil*
| *fpw-inverse* (*WordLeft a rest*) =
    fpw-concat (*fpw-inverse rest*) (*WordLeft* (− *a*) *WordNil*)

```
| fpw-inverse (WordRight b rest) =
    fpw-concat (fpw-inverse rest) (WordRight (- b) WordNil)

lemma fpw-inverse-concat:
  fpw-inverse (fpw-concat u v) = fpw-concat (fpw-inverse v) (fpw-inverse u)
  ⟨proof⟩

lemma fpw-inverse-inverse [simp]:
  fpw-inverse (fpw-inverse w) = w
  ⟨proof⟩

inductive fpw-reduction-step ::
  ('a::group-add, 'b::group-add) free-product-word =>
    ('a, 'b) free-product-word => bool
where
  combine-left:
    fpw-reduction-step
      (WordLeft a (WordLeft b rest))
      (WordLeft (a + b) rest)
| combine-right:
    fpw-reduction-step
      (WordRight a (WordRight b rest))
      (WordRight (a + b) rest)
| remove-left-zero:
    fpw-reduction-step (WordLeft 0 rest) rest
| remove-right-zero:
    fpw-reduction-step (WordRight 0 rest) rest
| context-left:
    fpw-reduction-step u v ==>
      fpw-reduction-step (WordLeft a u) (WordLeft a v)
| context-right:
    fpw-reduction-step u v ==>
      fpw-reduction-step (WordRight b u) (WordRight b v)

inductive fpw-reduction ::
  ('a::group-add, 'b::group-add) free-product-word =>
    ('a, 'b) free-product-word => bool
where
  refl [intro!, simp]: fpw-reduction w w
| sym: fpw-reduction u v ==> fpw-reduction v u
| trans: fpw-reduction u v ==> fpw-reduction v w ==> fpw-reduction u w
| step: fpw-reduction-step u v ==> fpw-reduction u v

lemma fpw-reduction-left-context:
  assumes fpw-reduction u v
  shows fpw-reduction (WordLeft a u) (WordLeft a v)
  ⟨proof⟩

lemma fpw-reduction-right-context:
```

assumes *fpw-reduction u v*
shows *fpw-reduction* (*WordRight b u*) (*WordRight b v*)
⟨*proof*⟩

end
theory *Amalgamated-Free-Product*
imports *Equivalence-Quotients Free-Product-Words*
begin

6 Amalgamation quotients of free-product words

This is the purely algebraic target of the later topological theorem. Starting from free-product words, the theory quotients by the relations induced by the common interface and packages the resulting equivalence classes as the abstract amalgamated free product.

inductive *amalgam-step* ::
(*'h => 'a*) => (*'h => 'b*) =>
(*'a*, *'b*) *free-product-word => ('a*, *'b*) *free-product-word => bool*
for *i1* :: *'h => 'a* **and** *i2* :: *'h => 'b*
where
identify:
amalgam-step i1 i2 (*WordLeft* (*i1 h*) *rest*) (*WordRight* (*i2 h*) *rest*)

inductive *amalgam-equiv* ::
(*'h => 'a*) => (*'h => 'b*) =>
(*'a*, *'b*) *free-product-word => ('a*, *'b*) *free-product-word => bool*
for *i1* :: *'h => 'a* **and** *i2* :: *'h => 'b*
where
refl [*intro*!, *simp*]: *amalgam-equiv i1 i2 w w*
| *sym*: *amalgam-equiv i1 i2 u v ==> amalgam-equiv i1 i2 v u*
| *trans*: *amalgam-equiv i1 i2 u v ==> amalgam-equiv i1 i2 v w ==> amalgam-equiv i1 i2 u w*
| *step*: *amalgam-step i1 i2 u v ==> amalgam-equiv i1 i2 u v*
| *left-context*:
amalgam-equiv i1 i2 u v ==> amalgam-equiv i1 i2 (*WordLeft a u*) (*WordLeft a v*)
| *right-context*:
amalgam-equiv i1 i2 u v ==> amalgam-equiv i1 i2 (*WordRight b u*) (*WordRight b v*)

interpretation *amalgam-equiv-equiv*: *equivalence-relation amalgam-equiv i1 i2*
⟨*proof*⟩

definition *amalgam-class* ::
(*'h => 'a*) => (*'h => 'b*) =>
(*'a*, *'b*) *free-product-word* => ((*'a*, *'b*) *free-product-word*) *set*
where
amalgam-class i1 i2 w = equiv-class (*amalgam-equiv i1 i2*) *w*

definition *amalgamated-free-product-space* ::
  $('h \Rightarrow 'a) \Rightarrow ('h \Rightarrow 'b) \Rightarrow$
    $(('a, 'b)$ *free-product-word*$)$ *set set*
where
  amalgamated-free-product-space i1 i2 = *quotient-space* (*amalgam-equiv i1 i2*)

lemma *amalgam-class-eq-iff*:
  amalgam-class i1 i2 u = *amalgam-class i1 i2 v* $\longleftrightarrow$ *amalgam-equiv i1 i2 u v*
  ⟨*proof*⟩

inductive *full-amalg-equiv* ::
  $('h \Rightarrow 'a$::*group-add*$) \Rightarrow ('h \Rightarrow 'b$::*group-add*$) \Rightarrow$
    $('a, 'b)$ *free-product-word* $\Rightarrow$ $('a, 'b)$ *free-product-word* $\Rightarrow$ *bool*
  for *i1* :: $'h \Rightarrow 'a$ **and** *i2* :: $'h \Rightarrow 'b$
where
  refl [*intro*!, *simp*]: *full-amalg-equiv i1 i2 w w*
| *sym*: *full-amalg-equiv i1 i2 u v* ⟹ *full-amalg-equiv i1 i2 v u*
| *trans*:
    full-amalg-equiv i1 i2 u v ⟹ *full-amalg-equiv i1 i2 v w* ⟹ *full-amalg-equiv i1 i2 u w*
| *of-amalg*:
    amalgam-equiv i1 i2 u v ⟹ *full-amalg-equiv i1 i2 u v*
| *of-reduction*:
    fpw-reduction u v ⟹ *full-amalg-equiv i1 i2 u v*

interpretation *full-amalg-equiv-equiv*: *equivalence-relation full-amalg-equiv i1 i2*
⟨*proof*⟩

definition *full-amalg-class* ::
  $('h \Rightarrow 'a$::*group-add*$) \Rightarrow ('h \Rightarrow 'b$::*group-add*$) \Rightarrow$
    $('a, 'b)$ *free-product-word* $\Rightarrow$ $(('a, 'b)$ *free-product-word*$)$ *set*
where
  full-amalg-class i1 i2 w = *equiv-class* (*full-amalg-equiv i1 i2*) *w*

definition *full-amalgamated-free-product-space* ::
  $('h \Rightarrow 'a$::*group-add*$) \Rightarrow ('h \Rightarrow 'b$::*group-add*$) \Rightarrow$
    $(('a, 'b)$ *free-product-word*$)$ *set set*
where
  full-amalgamated-free-product-space i1 i2 =
    quotient-space (*full-amalg-equiv i1 i2*)

lemma *full-amalg-class-eq-iff*:
  full-amalg-class i1 i2 u = *full-amalg-class i1 i2 v* $\longleftrightarrow$ *full-amalg-equiv i1 i2 u v*
  ⟨*proof*⟩

lemma *full-amalg-class-in-space* [*intro*]:
  full-amalg-class i1 i2 w ∈ *full-amalgamated-free-product-space i1 i2*
  ⟨*proof*⟩

definition *full-amalg-one* ::
  (*'h* => *'a*::*group-add*) => (*'h* => *'b*::*group-add*) =>
    ((*'a*, *'b*) *free-product-word*) *set*
where
  full-amalg-one i1 i2 = *full-amalg-class i1 i2 WordNil*

end
theory *Carrier-Amalgamated-Free-Product*
  imports *Amalgamated-Free-Product*
begin

7 Carrier-based amalgamated free products

The abstract amalgamated free product is refined here to words whose letters lie in fixed carrier sets. That carrier-level control matches the way loop representatives are produced on the topological side and is what eventually lets the Seifert–van Kampen theorem talk about concrete fundamental-group carriers.

fun *fpw-in-space* :: *'a set* => *'b set* => (*'a*, *'b*) *free-product-word* => *bool* **where**
  fpw-in-space G H WordNil = *True*
| *fpw-in-space G H* (*WordLeft a rest*) = (*a* ∈ *G* ∧ *fpw-in-space G H rest*)
| *fpw-in-space G H* (*WordRight b rest*) = (*b* ∈ *H* ∧ *fpw-in-space G H rest*)

definition *carrier-fpw-space* :: *'a set* => *'b set* => ((*'a*, *'b*) *free-product-word*) *set*
where
  carrier-fpw-space G H = {*w*. *fpw-in-space G H w*}

lemma *carrier-fpw-space-iff* [*simp*]:
  w ∈ *carrier-fpw-space G H* ⟷ *fpw-in-space G H w*
  ⟨*proof*⟩

inductive *carrier-amalgam-step* ::
  'h set => (*'h* => *'a*) => (*'h* => *'b*) =>
    (*'a*, *'b*) *free-product-word* => (*'a*, *'b*) *free-product-word* => *bool*
  for *K* :: *'h set* **and** *i1* :: *'h* => *'a* **and** *i2* :: *'h* => *'b*
where
  identify:
    h ∈ *K* ⟹
      carrier-amalgam-step K i1 i2
        (*WordLeft* (*i1 h*) *rest*)
        (*WordRight* (*i2 h*) *rest*)

inductive *carrier-amalgam-equiv* ::
  'h set => (*'h* => *'a*) => (*'h* => *'b*) =>
    (*'a*, *'b*) *free-product-word* => (*'a*, *'b*) *free-product-word* => *bool*
  for *K* :: *'h set* **and** *i1* :: *'h* => *'a* **and** *i2* :: *'h* => *'b*
where

```
  refl [intro!, simp]: carrier-amalgam-equiv K i1 i2 w w
| sym:
    carrier-amalgam-equiv K i1 i2 u v ⟹
      carrier-amalgam-equiv K i1 i2 v u
| trans:
    carrier-amalgam-equiv K i1 i2 u v ⟹
      carrier-amalgam-equiv K i1 i2 v w ⟹
      carrier-amalgam-equiv K i1 i2 u w
| step:
    carrier-amalgam-step K i1 i2 u v ⟹
      carrier-amalgam-equiv K i1 i2 u v
| left-context:
    carrier-amalgam-equiv K i1 i2 u v ⟹
      carrier-amalgam-equiv K i1 i2 (WordLeft a u) (WordLeft a v)
| right-context:
    carrier-amalgam-equiv K i1 i2 u v ⟹
      carrier-amalgam-equiv K i1 i2 (WordRight b u) (WordRight b v)

interpretation carrier-amalgam-equiv-equiv:
  equivalence-relation carrier-amalgam-equiv K i1 i2
⟨proof⟩

definition carrier-amalgam-class ::
  'h set => ('h => 'a) => ('h => 'b) =>
    ('a, 'b) free-product-word => (('a, 'b) free-product-word) set
where
  carrier-amalgam-class K i1 i2 w =
    equiv-class (carrier-amalgam-equiv K i1 i2) w

lemma carrier-amalgam-class-eq-iff:
  carrier-amalgam-class K i1 i2 u = carrier-amalgam-class K i1 i2 v
    ⟷ carrier-amalgam-equiv K i1 i2 u v
  ⟨proof⟩

inductive carrier-fpw-reduction-step ::
  'a set => 'b set =>
    ('a => 'a => 'a) => 'a => ('b => 'b => 'b) => 'b =>
    ('a, 'b) free-product-word => ('a, 'b) free-product-word => bool
  for G1 :: 'a set and G2 :: 'b set
    and mult1 :: 'a => 'a => 'a and one1 :: 'a
    and mult2 :: 'b => 'b => 'b and one2 :: 'b
where
  combine-left:
    ⟦a ∈ G1; b ∈ G1; mult1 a b ∈ G1; fpw-in-space G1 G2 rest⟧ ⟹
      carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
      (WordLeft a (WordLeft b rest))
      (WordLeft (mult1 a b) rest)
| combine-right:
    ⟦a ∈ G2; b ∈ G2; mult2 a b ∈ G2; fpw-in-space G1 G2 rest⟧ ⟹
```

carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
(*WordRight a* (*WordRight b rest*))
(*WordRight* (*mult2 a b*) *rest*)
| *remove-left-one*:
⟦*one1* ∈ *G1*; *fpw-in-space G1 G2 rest*⟧ ⟹
carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
(*WordLeft one1 rest*) *rest*
| *remove-right-one*:
⟦*one2* ∈ *G2*; *fpw-in-space G1 G2 rest*⟧ ⟹
carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
(*WordRight one2 rest*) *rest*
| *context-left*:
⟦*a* ∈ *G1*; *carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v*⟧ ⟹
carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
(*WordLeft a u*) (*WordLeft a v*)
| *context-right*:
⟦*b* ∈ *G2*; *carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v*⟧ ⟹
carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2
(*WordRight b u*) (*WordRight b v*)

inductive *carrier-fpw-reduction* ::
'a set => *'b set* =>
(*'a* => *'a* => *'a*) => *'a* => (*'b* => *'b* => *'b*) => *'b* =>
(*'a*, *'b*) *free-product-word* => (*'a*, *'b*) *free-product-word* => *bool*
for *G1* :: *'a set* **and** *G2* :: *'b set*
and *mult1* :: *'a* => *'a* => *'a* **and** *one1* :: *'a*
and *mult2* :: *'b* => *'b* => *'b* **and** *one2* :: *'b*
where
refl [*intro*!, *simp*]: *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 w w*
| *sym*:
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v ⟹
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 v u
| *trans*:
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v ⟹
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 v w ⟹
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u w
| *step*:
carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v ⟹
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v

lemma *carrier-fpw-reduction-left-context*:
assumes *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v*
and *a* ∈ *G1*
shows *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2* (*WordLeft a u*)
(*WordLeft a v*)
⟨*proof*⟩

lemma *carrier-fpw-reduction-right-context*:
assumes *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v*

and $b \in G2$
shows *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2* (*WordRight b u*) (*WordRight b v*)
⟨*proof*⟩

lemma *carrier-fpw-reduction-step-preserves-space*:
assumes *step*: *carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v*
shows *fpw-in-space G1 G2 u* **and** *fpw-in-space G1 G2 v*
⟨*proof*⟩

lemma *carrier-fpw-reduction-step-preserves-space-iff*:
assumes *step*: *carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v*
shows *fpw-in-space G1 G2 u* ⟷ *fpw-in-space G1 G2 v*
⟨*proof*⟩

lemma *carrier-fpw-reduction-preserves-space-iff*:
assumes *rel*: *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v*
shows *fpw-in-space G1 G2 u* ⟷ *fpw-in-space G1 G2 v*
⟨*proof*⟩

inductive *carrier-full-amalg-equiv* ::
'a set => 'b set => 'h set => ('h => 'a) => ('h => 'b) =>
('a => 'a => 'a) => 'a => ('b => 'b => 'b) => 'b =>
('a, 'b) free-product-word => ('a, 'b) free-product-word => bool
for *G1* :: *'a set* **and** *G2* :: *'b set*
and *K* :: *'h set* **and** *i1* :: *'h => 'a* **and** *i2* :: *'h => 'b*
and *mult1* :: *'a => 'a => 'a* **and** *one1* :: *'a*
and *mult2* :: *'b => 'b => 'b* **and** *one2* :: *'b*
where
refl [*intro*!, *simp*]:
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 w w
| *sym*:
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u v ⟹
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 v u
| *trans*:
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u v ⟹
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 v w ⟹
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u w
| *of-amalg*:
carrier-amalgam-equiv K i1 i2 u v ⟹
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u v
| *of-reduction*:
carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v ⟹
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u v

interpretation *carrier-full-amalg-equiv-equiv*:
equivalence-relation
carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2
⟨*proof*⟩

definition *carrier-full-amalg-class* ::
  'a set => 'b set => 'h set => ('h => 'a) => ('h => 'b) =>
    ('a => 'a => 'a) => 'a => ('b => 'b => 'b) => 'b =>
    ('a, 'b) free-product-word => (('a, 'b) free-product-word) set
where
  carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 w =
    equiv-class (carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2) w

definition *carrier-full-amalgamated-free-product-space* ::
  'a set => 'b set => 'h set => ('h => 'a) => ('h => 'b) =>
    ('a => 'a => 'a) => 'a => ('b => 'b => 'b) => 'b =>
    (('a, 'b) free-product-word) set set
where
  carrier-full-amalgamated-free-product-space G1 G2 K i1 i2 mult1 one1 mult2 one2
=
   carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 ‘ carrier-fpw-space
G1 G2

lemma *carrier-full-amalg-class-eq-iff*:
  carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 u =
      carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 v
    ⟷ carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u v
  ⟨*proof*⟩

lemma *carrier-full-amalg-class-in-space* [*intro*]:
  assumes *fpw-in-space G1 G2 w*
  shows
    carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 w ∈
      carrier-full-amalgamated-free-product-space G1 G2 K i1 i2 mult1 one1 mult2
one2
  ⟨*proof*⟩

definition *carrier-full-amalg-one* ::
  'a set => 'b set => 'h set => ('h => 'a) => ('h => 'b) =>
    ('a => 'a => 'a) => 'a => ('b => 'b => 'b) => 'b =>
    (('a, 'b) free-product-word) set
where
  carrier-full-amalg-one G1 G2 K i1 i2 mult1 one1 mult2 one2 =
    carrier-full-amalg-class G1 G2 K i1 i2 mult1 one1 mult2 one2 WordNil

end
theory *Carrier-Group-Scaffold*
  imports *Main*
begin

8 Groups on carriers

The main Seifert–van Kampen statement is formulated on concrete carrier sets rather than on type-class groups. This locale package isolates the carrier algebra laws and homomorphism notions that the amalgamation and fundamental-group constructions need later on.

locale *carrier-group* =
  fixes G :: $'a$ *set*
    and *mult* :: $'a$ => $'a$ => $'a$
    and *one* :: $'a$
    and *inv* :: $'a$ => $'a$
  assumes *one-closed* [*intro*, *simp*]: $one \in G$
    and *mult-closed* [*intro*]: $[\![x \in G;\ y \in G]\!] \Longrightarrow mult\ x\ y \in G$
    and *inv-closed* [*intro*]: $x \in G \Longrightarrow inv\ x \in G$
    and *mult-assoc*: $[\![x \in G;\ y \in G;\ z \in G]\!] \Longrightarrow mult\ (mult\ x\ y)\ z = mult\ x\ (mult\ y\ z)$
    and *mult-one-left*: $x \in G \Longrightarrow mult\ one\ x = x$
    and *mult-one-right*: $x \in G \Longrightarrow mult\ x\ one = x$
    and *mult-inv-left*: $x \in G \Longrightarrow mult\ (inv\ x)\ x = one$
    and *mult-inv-right*: $x \in G \Longrightarrow mult\ x\ (inv\ x) = one$
begin

lemma *left-cancel*:
  assumes *x*: $x \in G$
    and *y*: $y \in G$
    and *z*: $z \in G$
    and *eq*: $mult\ x\ y = mult\ x\ z$
  shows $y = z$
⟨*proof*⟩

lemma *right-cancel*:
  assumes *x*: $x \in G$
    and *y*: $y \in G$
    and *z*: $z \in G$
    and *eq*: $mult\ y\ x = mult\ z\ x$
  shows $y = z$
⟨*proof*⟩

lemma *left-inverse-unique*:
  assumes *x*: $x \in G$
    and *y*: $y \in G$
    and *eq*: $mult\ y\ x = one$
  shows $y = inv\ x$
⟨*proof*⟩

lemma *right-inverse-unique*:
  assumes *x*: $x \in G$
    and *y*: $y \in G$
    and *eq*: $mult\ x\ y = one$

shows $y = inv\ x$
⟨*proof*⟩

end

locale *carrier-group-hom* =
  G: *carrier-group G mult one inv* +
  H: *carrier-group H mult′ one′ inv′*
  for $G :: 'a\ set$ **and** $mult :: 'a => 'a => 'a$ **and** $one :: 'a$ **and** $inv :: 'a => 'a$
    and $H :: 'b\ set$ **and** $mult' :: 'b => 'b => 'b$ **and** $one' :: 'b$ **and** $inv' :: 'b => 'b$
    and $h :: 'a => 'b$ +
  assumes *map-closed*: $x \in G \Longrightarrow h\ x \in H$
    and *map-mult*: $[\![x \in G;\ y \in G]\!] \Longrightarrow h\ (mult\ x\ y) = mult'\ (h\ x)\ (h\ y)$
begin

lemma *map-one*:
  $h\ one = one'$
⟨*proof*⟩

lemma *map-inv*:
  assumes x: $x \in G$
  shows $h\ (inv\ x) = inv'\ (h\ x)$
⟨*proof*⟩

end

end
theory *Carrier-Amalgamated-Free-Product-Eval*
  imports *Carrier-Amalgamated-Free-Product Carrier-Group-Scaffold*
begin

9 Evaluating carrier amalgamated free-product words

Once the carrier-side amalgamated words have been defined, one still needs a way to evaluate them in a target carrier group. The evaluation locale proved here is the algebraic engine behind the later decoding maps in both the topological and classical Seifert–van Kampen statements.

locale *carrier-full-amalg-word-eval* =
  Grp1: *carrier-group G1 mult1 one1 inv1* +
  Grp2: *carrier-group G2 mult2 one2 inv2* +
  Cod: *carrier-group K multK oneK invK* +
  J1: *carrier-group-hom G1 mult1 one1 inv1 K multK oneK invK j1* +
  J2: *carrier-group-hom G2 mult2 one2 inv2 K multK oneK invK j2*
  for *G1* :: *'a set*
    and *mult1* :: *'a => 'a => 'a*
    and *one1* :: *'a*

```
    and inv1 :: 'a => 'a
    and G2 :: 'b set
    and mult2 :: 'b => 'b => 'b
    and one2 :: 'b
    and inv2 :: 'b => 'b
    and H :: 'h set
    and i1 :: 'h => 'a
    and i2 :: 'h => 'b
    and K :: 'k set
    and multK :: 'k => 'k => 'k
    and oneK :: 'k
    and invK :: 'k => 'k
    and j1 :: 'a => 'k
    and j2 :: 'b => 'k +
  assumes i1-closed: h ∈ H ⟹ i1 h ∈ G1
    and i2-closed: h ∈ H ⟹ i2 h ∈ G2
    and agree: h ∈ H ⟹ j1 (i1 h) = j2 (i2 h)
begin

fun carrier-full-amalg-eval :: ('a, 'b) free-product-word => 'k where
  carrier-full-amalg-eval WordNil = oneK
| carrier-full-amalg-eval (WordLeft a rest) =
    multK (j1 a) (carrier-full-amalg-eval rest)
| carrier-full-amalg-eval (WordRight b rest) =
    multK (j2 b) (carrier-full-amalg-eval rest)

lemma carrier-full-amalg-eval-in-carrier:
  assumes fpw-in-space G1 G2 w
  shows carrier-full-amalg-eval w ∈ K
  ⟨proof⟩

lemma carrier-amalgam-step-preserves-space-iff:
  assumes step: carrier-amalgam-step H i1 i2 u v
  shows fpw-in-space G1 G2 u ⟷ fpw-in-space G1 G2 v
  ⟨proof⟩

lemma carrier-amalgam-equiv-preserves-space-iff:
  assumes rel: carrier-amalgam-equiv H i1 i2 u v
  shows fpw-in-space G1 G2 u ⟷ fpw-in-space G1 G2 v
  ⟨proof⟩

lemma carrier-amalgam-step-preserves-eval:
  assumes step: carrier-amalgam-step H i1 i2 u v
  shows carrier-full-amalg-eval u = carrier-full-amalg-eval v
  ⟨proof⟩

lemma carrier-amalgam-equiv-preserves-eval:
  assumes rel: carrier-amalgam-equiv H i1 i2 u v
  shows carrier-full-amalg-eval u = carrier-full-amalg-eval v
```

⟨*proof*⟩

lemma *carrier-fpw-reduction-step-preserves-eval*:
assumes *step*: *carrier-fpw-reduction-step G1 G2 mult1 one1 mult2 one2 u v*
shows *carrier-full-amalg-eval u = carrier-full-amalg-eval v*
⟨*proof*⟩

lemmas *carrier-fpw-reduction-space-iff* =
Carrier-Amalgamated-Free-Product.carrier-fpw-reduction-preserves-space-iff

lemma *carrier-fpw-reduction-preserves-space-iff*:
assumes *rel*: *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v*
shows *fpw-in-space G1 G2 u* ⟷ *fpw-in-space G1 G2 v*
⟨*proof*⟩

lemma *carrier-fpw-reduction-preserves-eval*:
assumes *rel*: *carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v*
shows *carrier-full-amalg-eval u = carrier-full-amalg-eval v*
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-preserves-space-iff*:
assumes *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *fpw-in-space G1 G2 u* ⟷ *fpw-in-space G1 G2 v*
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-preserves-eval*:
assumes *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *carrier-full-amalg-eval u = carrier-full-amalg-eval v*
⟨*proof*⟩

definition *carrier-full-amalg-has-good-rep* :: (*'a*, *'b*) *free-product-word* => *bool*
where
carrier-full-amalg-has-good-rep w ⟷
(∃ *v*. *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 w v*
∧ *fpw-in-space G1 G2 v*)

definition *carrier-full-amalg-some-good-rep* ::
(*'a*, *'b*) *free-product-word* => (*'a*, *'b*) *free-product-word*
where
carrier-full-amalg-some-good-rep w =
(*SOME v*. *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 w v*
∧ *fpw-in-space G1 G2 v*)

definition *carrier-full-amalg-decode* :: (*'a*, *'b*) *free-product-word* => *'k* **where**
carrier-full-amalg-decode w =
(*if carrier-full-amalg-has-good-rep w*
then carrier-full-amalg-eval (*carrier-full-amalg-some-good-rep w*)
else oneK)

lemma *carrier-full-amalg-has-good-repI*:
assumes *fpw-in-space G1 G2 w*
shows *carrier-full-amalg-has-good-rep w*
⟨*proof*⟩

lemma *carrier-full-amalg-some-good-rep*:
assumes *carrier-full-amalg-has-good-rep w*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
w (*carrier-full-amalg-some-good-rep w*)
and *fpw-in-space G1 G2* (*carrier-full-amalg-some-good-rep w*)
⟨*proof*⟩

lemma *carrier-full-amalg-has-good-rep-respects*:
assumes *uv*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *carrier-full-amalg-has-good-rep u* ⟷ *carrier-full-amalg-has-good-rep v*
⟨*proof*⟩

lemma *carrier-full-amalg-decode-in-carrier*:
carrier-full-amalg-decode w ∈ *K*
⟨*proof*⟩

lemma *carrier-full-amalg-decode-respects*:
assumes *uv*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *carrier-full-amalg-decode u* = *carrier-full-amalg-decode v*
⟨*proof*⟩

lemma *carrier-full-amalg-decode-eq-eval*:
assumes *w-in*: *fpw-in-space G1 G2 w*
shows *carrier-full-amalg-decode w* = *carrier-full-amalg-eval w*
⟨*proof*⟩

end

end
theory *Explicit-Path-Homotopy-Scaffold*
imports *HOL−Analysis.Homotopy*
begin

10 Explicit-topology paths and homotopies

HOL-Analysis already provides paths and homotopies for subsets of a type. For the pushout-oriented development, however, it is convenient to work directly with arbitrary topologies. This theory therefore rephrases the basic path and homotopy notions in explicit-topology form.

definition *loopin-space* ::
'a topology ⇒ *'a* ⇒ (*real* ⇒ *'a*) *set*
where
loopin-space X x = {*p*. *pathin X p* ∧ *p 0* = *x* ∧ *p 1* = *x*}

definition *homotopic-pathsin* ::
  'a topology ⇒ (*real* ⇒ *'a*) ⇒ (*real* ⇒ *'a*) ⇒ *bool*
where
  homotopic-pathsin X p q ≡
    homotopic-with (λ*r*. *r 0* = *p 0* ∧ *r 1* = *p 1*) (*top-of-set* {*0..1*}) *X p q*

definition *homotopic-loopsin* ::
  'a topology ⇒ (*real* ⇒ *'a*) ⇒ (*real* ⇒ *'a*) ⇒ *bool*
where
  homotopic-loopsin X p q ≡
    homotopic-with (λ*r*. *r 1* = *r 0*) (*top-of-set* {*0..1*}) *X p q*

definition *reversepathin* :: (*real* ⇒ *'a*) ⇒ *real* ⇒ *'a*
where
  reversepathin p = (λ*t*. *p* (*1* − *t*))

definition *joinpathin* :: (*real* ⇒ *'a*) ⇒ (*real* ⇒ *'a*) ⇒ *real* ⇒ *'a*
where
  joinpathin p q = (λ*t*. *if t* ≤ *1 / 2 then p* (*2* ∗ *t*) *else q* (*2* ∗ *t* − *1*))

definition *subpathin* :: *real* ⇒ *real* ⇒ (*real* ⇒ *'a*) ⇒ *real* ⇒ *'a*
where
  subpathin u v p = (λ*t*. *p* ((*v* − *u*) ∗ *t* + *u*))

lemma *reversepathin-eq-reversepath* [*simp*]:
  reversepathin p = *reversepath p*
  ⟨*proof*⟩

lemma *reversepathin-reversepathin* [*simp*]:
  reversepathin (*reversepathin p*) = *p*
  ⟨*proof*⟩

lemma *joinpathin-eq-joinpaths* [*simp*]:
  joinpathin p q = *p* +++ *q*
  ⟨*proof*⟩

lemma *subpathin-image*:
  subpathin u v p ‘ {*0..1*} = *p* ‘ *closed-segment u v*
  ⟨*proof*⟩

lemma *subpathin-image-eq*:
  assumes *u* ≤ *v*
  shows *subpathin u v p* ‘ {*0..1*} = *p* ‘ {*u..v*}
  ⟨*proof*⟩

lemma *subpathin-0-1* [*simp*]:
  subpathin 0 1 p = *p*
  ⟨*proof*⟩

lemma *joinpathin-subpathin-middle* [*simp*]:
joinpathin (*subpathin 0* (*1 / 2*) *p*) (*subpathin* (*1 / 2*) *1 p*) = *p*
⟨*proof*⟩

lemma *continuous-map-top01-id*:
continuous-map (*top-of-set* {*0..1*}) *euclideanreal id*
⟨*proof*⟩

lemma *loopin-spaceI*:
assumes *pathin X p*
and *p 0* = *x*
and *p 1* = *x*
shows *p* ∈ *loopin-space X x*
⟨*proof*⟩

lemma *loopin-spaceE*:
assumes *p* ∈ *loopin-space X x*
obtains *pathin X p p 0* = *x p 1* = *x*
⟨*proof*⟩

lemma *constant-loopin-in-space*:
assumes *x* ∈ *topspace X*
shows (λ-. *x*) ∈ *loopin-space X x*
⟨*proof*⟩

lemma *pathin-image-subset-topspace*:
assumes *pathin X p*
shows *p* ‘ {*0..1*} ⊆ *topspace X*
⟨*proof*⟩

lemma *pathin-reversepathin*:
assumes *pathin X p*
shows *pathin X* (*reversepathin p*)
⟨*proof*⟩

lemma *pathin-joinpathin*:
assumes *p*: *pathin X p*
and *q*: *pathin X q*
and *pq*: *p 1* = *q 0*
shows *pathin X* (*joinpathin p q*)
⟨*proof*⟩

lemma *pathin-subpathin*:
assumes *p*: *pathin X p*
and *u*: *u* ∈ {*0..1*}
and *v*: *v* ∈ {*0..1*}
shows *pathin X* (*subpathin u v p*)
⟨*proof*⟩

lemma *pathin-subdivision-open-cover*:
assumes *p*: *pathin X p*
and *cover*: *p ‘ {0..1} ⊆ ⋃ 𝒰*
and *openU*: *⋀U. U ∈ 𝒰 ⟹ openin X U*
shows *∃ n::nat. 0 < n ∧*
(∀ i<n. ∃ U∈𝒰.
subpathin (real i / real n) (real (Suc i) / real n) p ‘ {0..1} ⊆ U)
⟨*proof*⟩

lemma *loopin-space-joinpathin*:
assumes *p*: *p ∈ loopin-space X x*
and *q*: *q ∈ loopin-space X x*
shows *joinpathin p q ∈ loopin-space X x*
⟨*proof*⟩

lemma *homotopic-pathsin-top-of-set* [*simp*]:
homotopic-pathsin (top-of-set S) p q ⟷ homotopic-paths S p q
⟨*proof*⟩

lemma *homotopic-loopsin-top-of-set* [*simp*]:
homotopic-loopsin (top-of-set S) p q ⟷ homotopic-loops S p q
⟨*proof*⟩

lemma *homotopic-pathsin-imp-pathin*:
assumes *homotopic-pathsin X p q*
shows *pathin X p pathin X q*
⟨*proof*⟩

lemma *homotopic-pathsin-imp-endpoints*:
assumes *homotopic-pathsin X p q*
shows *q 0 = p 0 q 1 = p 1*
⟨*proof*⟩

lemma *homotopic-pathsin-refl* [*simp*]:
homotopic-pathsin X p p ⟷ pathin X p
⟨*proof*⟩

lemma *homotopic-pathsin-sym*:
assumes *homotopic-pathsin X p q*
shows *homotopic-pathsin X q p*
⟨*proof*⟩

lemma *homotopic-pathsin-trans*:
assumes *homotopic-pathsin X p q*
and *homotopic-pathsin X q r*
shows *homotopic-pathsin X p r*
⟨*proof*⟩

lemma *homotopic-pathsin-eq*:
assumes *pathin X p*
and $\bigwedge t.\ t \in \{0..1\} \Longrightarrow p\ t = q\ t$
shows *homotopic-pathsin X p q*
⟨*proof*⟩

lemma *continuous-map-homotopic-joinpathin-lemma*:
fixes *p q* :: *real* ⇒ *real* ⇒ *'a*
assumes *p*:
continuous-map (*prod-topology* (*top-of-set* {*0..1*}) (*top-of-set* {*0..1*})) *X*
(λ*y*. *p* (*fst y*) (*snd y*))
and *q*:
continuous-map (*prod-topology* (*top-of-set* {*0..1*}) (*top-of-set* {*0..1*})) *X*
(λ*y*. *q* (*fst y*) (*snd y*))
and *pq*: $\bigwedge t.\ t \in \{0..1\} \Longrightarrow p\ t\ 1 = q\ t\ 0$
shows
continuous-map (*prod-topology* (*top-of-set* {*0..1*}) (*top-of-set* {*0..1*})) *X*
(λ*y*. *joinpathin* (*p* (*fst y*)) (*q* (*fst y*)) (*snd y*))
⟨*proof*⟩

lemma *homotopic-pathsin-continuous-image*:
assumes *pq*: *homotopic-pathsin X p q*
and *h*: *continuous-map X Y f*
shows *homotopic-pathsin Y* (*f* ∘ *p*) (*f* ∘ *q*)
⟨*proof*⟩

lemma *homotopic-pathsin-reversepathin-D*:
assumes *homotopic-pathsin X p q*
shows *homotopic-pathsin X* (*reversepathin p*) (*reversepathin q*)
⟨*proof*⟩

lemma *homotopic-pathsin-reversepathin*:
homotopic-pathsin X (*reversepathin p*) (*reversepathin q*) ⟷ *homotopic-pathsin X p q*
⟨*proof*⟩

lemma *homotopic-pathsin-joinpathin*:
assumes *pp'*: *homotopic-pathsin X p p'*
and *qq'*: *homotopic-pathsin X q q'*
and *pq*: *p 1* = *q 0*
shows *homotopic-pathsin X* (*joinpathin p q*) (*joinpathin p' q'*)
⟨*proof*⟩

lemma *homotopic-pathsin-reparametrize*:
assumes *p*: *pathin X p*
and *contf*: *continuous-map* (*top-of-set* {*0..1*}) (*top-of-set* {*0..1*}) *f*
and *f01*: *f* ∈ {*0..1*} → {*0..1*}
and [*simp*]: *f 0* = *0 f 1* = *1*
and *q*: $\bigwedge t.\ t \in \{0..1\} \Longrightarrow q\ t = p\ (f\ t)$

shows *homotopic-pathsin X p q*
⟨*proof*⟩

lemma *homotopic-pathsin-rid-const*:
assumes *p*: *pathin X p*
shows *homotopic-pathsin X* (*joinpathin p* (λ-. *p 1*)) *p*
⟨*proof*⟩

lemma *homotopic-pathsin-lid-const*:
assumes *p*: *pathin X p*
shows *homotopic-pathsin X* (*joinpathin* (λ-. *p 0*) *p*) *p*
⟨*proof*⟩

lemma *homotopic-pathsin-assoc*:
assumes *p*: *pathin X p*
and *q*: *pathin X q*
and *r*: *pathin X r*
and *pq*: *p 1 = q 0*
and *qr*: *q 1 = r 0*
shows *homotopic-pathsin X* (*joinpathin p* (*joinpathin q r*)) (*joinpathin* (*joinpathin p q*) *r*)
⟨*proof*⟩

lemma *homotopic-pathsin-rinv-const*:
assumes *p*: *pathin X p*
shows *homotopic-pathsin X* (*joinpathin p* (*reversepathin p*)) (λ-. *p 0*)
⟨*proof*⟩

lemma *homotopic-pathsin-linv-const*:
assumes *p*: *pathin X p*
shows *homotopic-pathsin X* (*joinpathin* (*reversepathin p*) *p*) (λ-. *p 1*)
⟨*proof*⟩

lemma *homotopic-loopsin-imp-pathin*:
assumes *homotopic-loopsin X p q*
shows *pathin X p pathin X q*
⟨*proof*⟩

lemma *homotopic-loopsin-imp-loop*:
assumes *homotopic-loopsin X p q*
shows *p 1 = p 0 q 1 = q 0*
⟨*proof*⟩

lemma *homotopic-loopsin-refl* [*simp*]:
homotopic-loopsin X p p ⟷ *pathin X p* ∧ *p 1 = p 0*
⟨*proof*⟩

lemma *homotopic-loopsin-sym*:
assumes *homotopic-loopsin X p q*

shows *homotopic-loopsin X q p*
⟨*proof*⟩

lemma *homotopic-loopsin-trans*:
assumes *homotopic-loopsin X p q*
and *homotopic-loopsin X q r*
shows *homotopic-loopsin X p r*
⟨*proof*⟩

lemma *loopin-space-reversepathin*:
assumes *p* ∈ *loopin-space X x*
shows *reversepathin p* ∈ *loopin-space X x*
⟨*proof*⟩

end
theory *Fundamental-Group-Scaffold*
imports *HOL−Analysis.Homotopy Carrier-Group-Scaffold*
begin

11 Fundamental-group quotients

This is the set-based counterpart to the explicit-topology quotient development. It works directly with subsets of a topological space and the standard HOL-Analysis path notions, which is exactly the level needed for the final classical theorem about open sets *U* and *V*.

definition *loop-space* ::
'a::topological-space set => 'a => (real => 'a) set
where
loop-space S x =
{p. path p ∧ path-image p ⊆ S ∧ pathstart p = x ∧ pathfinish p = x}

definition *loop-class* ::
'a::topological-space set => 'a => (real => 'a) => (real => 'a) set
where
loop-class S x p = {q. q ∈ loop-space S x ∧ homotopic-paths S q p}

definition *fundamental-group-space* ::
'a::topological-space set => 'a => ((real => 'a) set) set
where
fundamental-group-space S x = loop-class S x ‘ loop-space S x

definition *some-loop* ::
'a::topological-space set => 'a => (real => 'a) set => (real => 'a)
where
some-loop S x Q = (SOME p. p ∈ loop-space S x ∧ Q = loop-class S x p)

definition *fundamental-group-one* ::
'a::topological-space set => 'a => (real => 'a) set

where
fundamental-group-one S x = loop-class S x (λ-. x)

definition *fundamental-group-mult* ::
'a::topological-space set => 'a =>
(real => 'a) set => (real => 'a) set => (real => 'a) set
where
fundamental-group-mult S x A B =
loop-class S x (some-loop S x A +++ some-loop S x B)

definition *fundamental-group-inv* ::
'a::topological-space set => 'a =>
(real => 'a) set => (real => 'a) set
where
fundamental-group-inv S x A = loop-class S x (reversepath (some-loop S x A))

definition *loop-image* ::
('a::topological-space => 'b::topological-space) =>
(real => 'a) => (real => 'b)
where
loop-image h p = h ∘ p

definition *fundamental-group-map* ::
'a::topological-space set => 'a =>
'b::topological-space set => 'b =>
('a => 'b) => (real => 'a) set => (real => 'b) set
where
fundamental-group-map S x T y h A =
loop-class T y (loop-image h (some-loop S x A))

lemma *loop-spaceI*:
assumes *path p*
and *path-image p ⊆ S*
and *pathstart p = x*
and *pathfinish p = x*
shows *p ∈ loop-space S x*
⟨*proof*⟩

lemma *constant-loop-in-space*:
assumes *x ∈ S*
shows *(λ-. x) ∈ loop-space S x*
⟨*proof*⟩

lemma *loop-class-in-space*:
assumes *p ∈ loop-space S x*
shows *loop-class S x p ∈ fundamental-group-space S x*
⟨*proof*⟩

lemma *fundamental-group-spaceE*:

assumes $Q \in$ *fundamental-group-space* $S\ x$
obtains p **where** $p \in$ *loop-space* $S\ x\ Q =$ *loop-class* $S\ x\ p$
⟨*proof*⟩

lemma *some-loop-spec*:
assumes $Q \in$ *fundamental-group-space* $S\ x$
shows *some-loop* $S\ x\ Q \in$ *loop-space* $S\ x$
and $Q =$ *loop-class* $S\ x$ (*some-loop* $S\ x\ Q$)
⟨*proof*⟩

lemma *loop-class-eqI*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
and *pq*: *homotopic-paths* $S\ p\ q$
shows *loop-class* $S\ x\ p =$ *loop-class* $S\ x\ q$
⟨*proof*⟩

lemma *loop-class-eq-iff*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
shows *loop-class* $S\ x\ p =$ *loop-class* $S\ x\ q \longleftrightarrow$ *homotopic-paths* $S\ p\ q$
⟨*proof*⟩

lemma *loop-space-join*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
shows p +++ $q \in$ *loop-space* $S\ x$
⟨*proof*⟩

lemma *loop-space-reversepath*:
assumes $p \in$ *loop-space* $S\ x$
shows *reversepath* $p \in$ *loop-space* $S\ x$
⟨*proof*⟩

lemma *loop-space-continuous-image*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and *cont-h*: *continuous-on* $S\ h$
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
shows *loop-image* $h\ p \in$ *loop-space* $T\ y$
⟨*proof*⟩

lemma *loop-image-join*:
loop-image h (p +++ q) = *loop-image* $h\ p$ +++ *loop-image* $h\ q$
⟨*proof*⟩

lemma *loop-image-reversepath*:
loop-image h (*reversepath* p) = *reversepath* (*loop-image* $h\ p$)
⟨*proof*⟩

lemma *loop-class-join-eqI*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and p': $p' \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
and q': $q' \in$ *loop-space* $S\ x$
and pp': *homotopic-paths* $S\ p\ p'$
and qq': *homotopic-paths* $S\ q\ q'$
shows *loop-class* $S\ x\ (p +\!\!+\!\!+ q) =$ *loop-class* $S\ x\ (p' +\!\!+\!\!+ q')$
⟨*proof*⟩

lemma *loop-class-reverse-eqI*:
assumes *p*: $p \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
and *pq*: *homotopic-paths* $S\ p\ q$
shows *loop-class* $S\ x$ (*reversepath* p) = *loop-class* $S\ x$ (*reversepath* q)
⟨*proof*⟩

lemma *homotopic-paths-rid-const*:
assumes *path* p
and *path-image* $p \subseteq S$
shows *homotopic-paths* $S\ (p +\!\!+\!\!+ (\lambda\text{-}.\ pathfinish\ p))\ p$
⟨*proof*⟩

lemma *homotopic-paths-lid-const*:
assumes *path* p
and *path-image* $p \subseteq S$
shows *homotopic-paths* $S\ ((\lambda\text{-}.\ pathstart\ p) +\!\!+\!\!+ p)\ p$
⟨*proof*⟩

lemma *homotopic-paths-rinv-const*:
assumes *path* p
and *path-image* $p \subseteq S$
shows *homotopic-paths* $S\ (p +\!\!+\!\!+$ *reversepath* $p)\ (\lambda\text{-}.\ pathstart\ p)$
⟨*proof*⟩

lemma *homotopic-paths-linv-const*:
assumes *path* p
and *path-image* $p \subseteq S$
shows *homotopic-paths* S (*reversepath* $p +\!\!+\!\!+ p)\ (\lambda\text{-}.\ pathfinish\ p)$
⟨*proof*⟩

lemma *fundamental-group-one-in-space*:
assumes $x \in S$
shows *fundamental-group-one* $S\ x \in$ *fundamental-group-space* $S\ x$
⟨*proof*⟩

lemma *fundamental-group-mult-eqI*:
assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$

and *B-in*: $B \in$ *fundamental-group-space* $S\ x$
and *p*: $p \in$ *loop-space* $S\ x$
and *q*: $q \in$ *loop-space* $S\ x$
and *A*: $A =$ *loop-class* $S\ x\ p$
and *B*: $B =$ *loop-class* $S\ x\ q$
shows *fundamental-group-mult* $S\ x\ A\ B =$ *loop-class* $S\ x\ (p +\!\!+\!\!+\ q)$
⟨*proof*⟩

lemma *fundamental-group-mult-in-space*:
assumes $A \in$ *fundamental-group-space* $S\ x$
and $B \in$ *fundamental-group-space* $S\ x$
shows *fundamental-group-mult* $S\ x\ A\ B \in$ *fundamental-group-space* $S\ x$
⟨*proof*⟩

lemma *fundamental-group-inv-eqI*:
assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$
and *p*: $p \in$ *loop-space* $S\ x$
and *A*: $A =$ *loop-class* $S\ x\ p$
shows *fundamental-group-inv* $S\ x\ A =$ *loop-class* $S\ x$ (*reversepath* p)
⟨*proof*⟩

lemma *fundamental-group-inv-in-space*:
assumes $A \in$ *fundamental-group-space* $S\ x$
shows *fundamental-group-inv* $S\ x\ A \in$ *fundamental-group-space* $S\ x$
⟨*proof*⟩

lemma *fundamental-group-map-eqI*:
assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$
and *p*: $p \in$ *loop-space* $S\ x$
and *A*: $A =$ *loop-class* $S\ x\ p$
and *cont-h*: *continuous-on* $S\ h$
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
shows *fundamental-group-map* $S\ x\ T\ y\ h\ A =$ *loop-class* $T\ y$ (*loop-image* $h\ p$)
⟨*proof*⟩

lemma *fundamental-group-map-in-space*:
assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$
and *cont-h*: *continuous-on* $S\ h$
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
shows *fundamental-group-map* $S\ x\ T\ y\ h\ A \in$ *fundamental-group-space* $T\ y$
⟨*proof*⟩

lemma *fundamental-group-map-mult*:
assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$
and *B-in*: $B \in$ *fundamental-group-space* $S\ x$
and *cont-h*: *continuous-on* $S\ h$
and *map-h*: $h \in S \to T$

and *hx*: $h\ x = y$
shows
$fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ (fundamental\text{-}group\text{-}mult\ S\ x\ A\ B) =$
$fundamental\text{-}group\text{-}mult\ T\ y$
$(fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ A)$
$(fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ B)$
⟨*proof*⟩

lemma *fundamental-group-map-id*:
assumes *A-in*: $A \in fundamental\text{-}group\text{-}space\ S\ x$
shows $fundamental\text{-}group\text{-}map\ S\ x\ S\ x\ id\ A = A$
⟨*proof*⟩

lemma *fundamental-group-map-one*:
assumes *x-in*: $x \in S$
and *cont-h*: *continuous-on S h*
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
shows
$fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ (fundamental\text{-}group\text{-}one\ S\ x) =$
$fundamental\text{-}group\text{-}one\ T\ y$
⟨*proof*⟩

lemma *fundamental-group-map-compose*:
assumes *A-in*: $A \in fundamental\text{-}group\text{-}space\ S\ x$
and *cont-h*: *continuous-on S h*
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
and *cont-k*: *continuous-on T k*
and *map-k*: $k \in T \to U$
and *ky*: $k\ y = z$
shows
$fundamental\text{-}group\text{-}map\ T\ y\ U\ z\ k\ (fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ A) =$
$fundamental\text{-}group\text{-}map\ S\ x\ U\ z\ (k \circ h)\ A$
⟨*proof*⟩

lemma *fundamental-group-mult-one-left*:
assumes $x \in S$
and $A \in fundamental\text{-}group\text{-}space\ S\ x$
shows $fundamental\text{-}group\text{-}mult\ S\ x\ (fundamental\text{-}group\text{-}one\ S\ x)\ A = A$
⟨*proof*⟩

lemma *fundamental-group-mult-one-right*:
assumes $x \in S$
and $A \in fundamental\text{-}group\text{-}space\ S\ x$
shows $fundamental\text{-}group\text{-}mult\ S\ x\ A\ (fundamental\text{-}group\text{-}one\ S\ x) = A$
⟨*proof*⟩

lemma *fundamental-group-mult-assoc*:

assumes *A-in*: $A \in$ *fundamental-group-space* $S\ x$
and *B-in*: $B \in$ *fundamental-group-space* $S\ x$
and *C-in*: $C \in$ *fundamental-group-space* $S\ x$
shows *fundamental-group-mult* $S\ x\ A$ (*fundamental-group-mult* $S\ x\ B\ C$) =
fundamental-group-mult $S\ x$ (*fundamental-group-mult* $S\ x\ A\ B$) C
⟨*proof*⟩

lemma *fundamental-group-mult-inv-right*:
assumes $x \in S$
and $A \in$ *fundamental-group-space* $S\ x$
shows *fundamental-group-mult* $S\ x\ A$ (*fundamental-group-inv* $S\ x\ A$) = *fundamental-group-one* $S\ x$
⟨*proof*⟩

lemma *fundamental-group-mult-inv-left*:
assumes $x \in S$
and $A \in$ *fundamental-group-space* $S\ x$
shows *fundamental-group-mult* $S\ x$ (*fundamental-group-inv* $S\ x\ A$) A = *fundamental-group-one* $S\ x$
⟨*proof*⟩

lemma *trivial-loop-class-in-space*:
assumes $x \in S$
shows *loop-class* $S\ x$ (λ-. x) $\in$ *fundamental-group-space* $S\ x$
⟨*proof*⟩

lemma *fundamental-group-carrier-group*:
assumes *x-in*: $x \in S$
shows *carrier-group*
(*fundamental-group-space* $S\ x$)
(*fundamental-group-mult* $S\ x$)
(*fundamental-group-one* $S\ x$)
(*fundamental-group-inv* $S\ x$)
⟨*proof*⟩

lemma *fundamental-group-map-carrier-group-hom*:
assumes *x-in*: $x \in S$
and *cont-h*: *continuous-on* $S\ h$
and *map-h*: $h \in S \to T$
and *hx*: $h\ x = y$
shows *carrier-group-hom*
(*fundamental-group-space* $S\ x$)
(*fundamental-group-mult* $S\ x$)
(*fundamental-group-one* $S\ x$)
(*fundamental-group-inv* $S\ x$)
(*fundamental-group-space* $T\ y$)
(*fundamental-group-mult* $T\ y$)
(*fundamental-group-one* $T\ y$)
(*fundamental-group-inv* $T\ y$)

(*fundamental-group-map S x T y h*)
⟨*proof*⟩

end
theory *Explicit-Fundamental-Group-Scaffold*
imports *Explicit-Path-Homotopy-Scaffold Fundamental-Group-Scaffold*
begin

12 Explicit-topology fundamental-group quotients

Building on the explicit-topology path layer, this theory forms loop classes, quotient carriers, and induced maps for arbitrary topological spaces. It is the main bridge from point-set topology to the carrier-group algebra used in the Seifert–van Kampen interface.

definition *loopin-class* ::
'a topology ⇒ *'a* ⇒ (*real* ⇒ *'a*) ⇒ (*real* ⇒ *'a*) *set*
where
loopin-class X x p = {*q*. *q* ∈ *loopin-space X x* ∧ *homotopic-pathsin X q p*}

definition *fundamental-groupin-space* ::
'a topology ⇒ *'a* ⇒ ((*real* ⇒ *'a*) *set*) *set*
where
fundamental-groupin-space X x = *loopin-class X x* ‘ *loopin-space X x*

definition *some-loopin* ::
'a topology ⇒ *'a* ⇒ (*real* ⇒ *'a*) *set* ⇒ (*real* ⇒ *'a*)
where
some-loopin X x Q = (*SOME p*. *p* ∈ *loopin-space X x* ∧ *Q* = *loopin-class X x p*)

definition *fundamental-groupin-one* ::
'a topology ⇒ *'a* ⇒ (*real* ⇒ *'a*) *set*
where
fundamental-groupin-one X x = *loopin-class X x* (λ-. *x*)

definition *fundamental-groupin-mult* ::
'a topology ⇒ *'a* ⇒
(*real* ⇒ *'a*) *set* ⇒ (*real* ⇒ *'a*) *set* ⇒ (*real* ⇒ *'a*) *set*
where
fundamental-groupin-mult X x A B =
loopin-class X x (*joinpathin* (*some-loopin X x A*) (*some-loopin X x B*))

definition *fundamental-groupin-inv* ::
'a topology ⇒ *'a* ⇒ (*real* ⇒ *'a*) *set* ⇒ (*real* ⇒ *'a*) *set*
where
fundamental-groupin-inv X x A = *loopin-class X x* (*reversepathin* (*some-loopin X x A*))

definition *loopin-image* ::
  $('a \Rightarrow 'b) \Rightarrow (real \Rightarrow 'a) \Rightarrow (real \Rightarrow 'b)$
where
  loopin-image h p = $h \circ p$

definition *fundamental-groupin-map* ::
  $'a$ *topology* $\Rightarrow$ $'a \Rightarrow$ $'b$ *topology* $\Rightarrow$ $'b \Rightarrow$
    $('a \Rightarrow 'b) \Rightarrow (real \Rightarrow 'a)$ *set* $\Rightarrow (real \Rightarrow 'b)$ *set*
where
  fundamental-groupin-map X x Y y h A =
     loopin-class Y y (*loopin-image h* (*some-loopin X x A*))

lemma *loopin-class-in-space*:
  assumes $p \in$ *loopin-space X x*
  shows *loopin-class X x p* $\in$ *fundamental-groupin-space X x*
  ⟨*proof*⟩

lemma *fundamental-groupin-spaceE*:
  assumes $Q \in$ *fundamental-groupin-space X x*
  obtains p **where** $p \in$ *loopin-space X x Q* = *loopin-class X x p*
  ⟨*proof*⟩

lemma *some-loopin-spec*:
  assumes $Q \in$ *fundamental-groupin-space X x*
  shows *some-loopin X x Q* $\in$ *loopin-space X x*
    and Q = *loopin-class X x* (*some-loopin X x Q*)
⟨*proof*⟩

lemma *loopin-class-eqI*:
  assumes *p*: $p \in$ *loopin-space X x*
    and *q*: $q \in$ *loopin-space X x*
    and *pq*: *homotopic-pathsin X p q*
  shows *loopin-class X x p* = *loopin-class X x q*
⟨*proof*⟩

lemma *loopin-class-eq-iff*:
  assumes *p*: $p \in$ *loopin-space X x*
    and *q*: $q \in$ *loopin-space X x*
  shows *loopin-class X x p* = *loopin-class X x q* $\longleftrightarrow$ *homotopic-pathsin X p q*
⟨*proof*⟩

lemma *fundamental-groupin-one-in-space*:
  assumes $x \in$ *topspace X*
  shows *fundamental-groupin-one X x* $\in$ *fundamental-groupin-space X x*
  ⟨*proof*⟩

lemma *fundamental-groupin-mult-eqI*:
  assumes *A-in*: $A \in$ *fundamental-groupin-space X x*

and *B-in*: $B \in$ *fundamental-groupin-space* $X\ x$
and *p*: $p \in$ *loopin-space* $X\ x$
and *q*: $q \in$ *loopin-space* $X\ x$
and *A*: $A =$ *loopin-class* $X\ x\ p$
and *B*: $B =$ *loopin-class* $X\ x\ q$
shows *fundamental-groupin-mult* $X\ x\ A\ B =$ *loopin-class* $X\ x$ (*joinpathin* $p\ q$)
⟨*proof*⟩

lemma *fundamental-groupin-mult-in-space*:
assumes $A \in$ *fundamental-groupin-space* $X\ x$
and $B \in$ *fundamental-groupin-space* $X\ x$
shows *fundamental-groupin-mult* $X\ x\ A\ B \in$ *fundamental-groupin-space* $X\ x$
⟨*proof*⟩

lemma *fundamental-groupin-inv-eqI*:
assumes *A-in*: $A \in$ *fundamental-groupin-space* $X\ x$
and *p*: $p \in$ *loopin-space* $X\ x$
and *A*: $A =$ *loopin-class* $X\ x\ p$
shows *fundamental-groupin-inv* $X\ x\ A =$ *loopin-class* $X\ x$ (*reversepathin* p)
⟨*proof*⟩

lemma *fundamental-groupin-inv-in-space*:
assumes $A \in$ *fundamental-groupin-space* $X\ x$
shows *fundamental-groupin-inv* $X\ x\ A \in$ *fundamental-groupin-space* $X\ x$
⟨*proof*⟩

lemma *loopin-image-in-space*:
assumes *p*: $p \in$ *loopin-space* $X\ x$
and *h*: *continuous-map* $X\ Y\ f$
and *fx*: $f\ x = y$
shows *loopin-image* $f\ p \in$ *loopin-space* $Y\ y$
⟨*proof*⟩

lemma *fundamental-groupin-map-rep*:
assumes *A*: $A \in$ *fundamental-groupin-space* $X\ x$
and *p*: $p \in$ *loopin-space* $X\ x$
and *rep*: $A =$ *loopin-class* $X\ x\ p$
and *h*: *continuous-map* $X\ Y\ f$
and *fx*: $f\ x = y$
shows *fundamental-groupin-map* $X\ x\ Y\ y\ f\ A =$ *loopin-class* $Y\ y$ (*loopin-image* f p)
⟨*proof*⟩

lemma *fundamental-groupin-map-in-space*:
assumes *A*: $A \in$ *fundamental-groupin-space* $X\ x$
and *h*: *continuous-map* $X\ Y\ f$
and *fx*: $f\ x = y$
shows *fundamental-groupin-map* $X\ x\ Y\ y\ f\ A \in$ *fundamental-groupin-space* $Y\ y$
⟨*proof*⟩

lemma *loopin-image-joinpathin* [*simp*]:
loopin-image h (*joinpathin p q*) = *joinpathin* (*loopin-image h p*) (*loopin-image h q*)
⟨*proof*⟩

lemma *loopin-image-reversepathin* [*simp*]:
loopin-image h (*reversepathin p*) = *reversepathin* (*loopin-image h p*)
⟨*proof*⟩

lemma *fundamental-groupin-map-mult*:
assumes *A-in*: *A* ∈ *fundamental-groupin-space X x*
and *B-in*: *B* ∈ *fundamental-groupin-space X x*
and *h*: *continuous-map X Y f*
and *fx*: *f x* = *y*
shows *fundamental-groupin-map X x Y y f* (*fundamental-groupin-mult X x A B*) =
fundamental-groupin-mult Y y
(*fundamental-groupin-map X x Y y f A*)
(*fundamental-groupin-map X x Y y f B*)
⟨*proof*⟩

lemma *fundamental-groupin-mult-one-left*:
assumes *x-in*: *x* ∈ *topspace X*
and *A-in*: *A* ∈ *fundamental-groupin-space X x*
shows *fundamental-groupin-mult X x* (*fundamental-groupin-one X x*) *A* = *A*
⟨*proof*⟩

lemma *fundamental-groupin-mult-one-right*:
assumes *x-in*: *x* ∈ *topspace X*
and *A-in*: *A* ∈ *fundamental-groupin-space X x*
shows *fundamental-groupin-mult X x A* (*fundamental-groupin-one X x*) = *A*
⟨*proof*⟩

lemma *fundamental-groupin-mult-assoc*:
assumes *A-in*: *A* ∈ *fundamental-groupin-space X x*
and *B-in*: *B* ∈ *fundamental-groupin-space X x*
and *C-in*: *C* ∈ *fundamental-groupin-space X x*
shows *fundamental-groupin-mult X x A* (*fundamental-groupin-mult X x B C*) =
fundamental-groupin-mult X x (*fundamental-groupin-mult X x A B*) *C*
⟨*proof*⟩

lemma *fundamental-groupin-mult-inv-right*:
assumes *x-in*: *x* ∈ *topspace X*
and *A-in*: *A* ∈ *fundamental-groupin-space X x*
shows *fundamental-groupin-mult X x A* (*fundamental-groupin-inv X x A*) = *fundamental-groupin-one X x*
⟨*proof*⟩

lemma *fundamental-groupin-mult-inv-left*:
assumes *x-in*: $x \in$ *topspace* X
and *A-in*: $A \in$ *fundamental-groupin-space* $X\ x$
shows *fundamental-groupin-mult* $X\ x$ (*fundamental-groupin-inv* $X\ x\ A$) $A =$ *fundamental-groupin-one* $X\ x$
⟨*proof*⟩

lemma *fundamental-groupin-carrier-group*:
assumes *x-in*: $x \in$ *topspace* X
shows *carrier-group*
(*fundamental-groupin-space* $X\ x$)
(*fundamental-groupin-mult* $X\ x$)
(*fundamental-groupin-one* $X\ x$)
(*fundamental-groupin-inv* $X\ x$)
⟨*proof*⟩

lemma *fundamental-groupin-map-carrier-group-hom*:
assumes *x-in*: $x \in$ *topspace* X
and *h*: *continuous-map* $X\ Y\ f$
and *fx*: $f\ x = y$
shows *carrier-group-hom*
(*fundamental-groupin-space* $X\ x$)
(*fundamental-groupin-mult* $X\ x$)
(*fundamental-groupin-one* $X\ x$)
(*fundamental-groupin-inv* $X\ x$)
(*fundamental-groupin-space* $Y\ y$)
(*fundamental-groupin-mult* $Y\ y$)
(*fundamental-groupin-one* $Y\ y$)
(*fundamental-groupin-inv* $Y\ y$)
(*fundamental-groupin-map* $X\ x\ Y\ y\ f$)
⟨*proof*⟩

lemma *loopin-space-top-of-set* [*simp*]:
loopin-space (*top-of-set* S) $x =$ *loop-space* $S\ x$
⟨*proof*⟩

lemma *loopin-class-top-of-set* [*simp*]:
loopin-class (*top-of-set* S) $x\ p =$ *loop-class* $S\ x\ p$
⟨*proof*⟩

lemma *fundamental-groupin-space-top-of-set* [*simp*]:
fundamental-groupin-space (*top-of-set* S) $x =$ *fundamental-group-space* $S\ x$
⟨*proof*⟩

lemma *fundamental-groupin-one-top-of-set* [*simp*]:
fundamental-groupin-one (*top-of-set* S) $x =$ *fundamental-group-one* $S\ x$
⟨*proof*⟩

lemma *loopin-image-eq-loop-image* [*simp*]:

loopin-image h p = loop-image h p
⟨*proof*⟩

end
theory *Pushout-Scaffold*
imports *Equivalence-Quotients*
begin

13 Pushout-style quotients of disjoint sums

The pushout of two maps is first presented here as a quotient of the disjoint sum by the obvious glue relation. This algebraic infrastructure is independent of any topology; the topological quotient is added later, once the point-set infrastructure is in place.

inductive *pushout-rel* ::
('c => 'a) => ('c => 'b) => ('a + 'b) => ('a + 'b) => bool
for *f* :: *'c => 'a* **and** *g* :: *'c => 'b*
where
refl [*intro*!, *simp*]: *pushout-rel f g x x*
| *sym*: *pushout-rel f g x y ==> pushout-rel f g y x*
| *trans*: *pushout-rel f g x y ==> pushout-rel f g y z ==> pushout-rel f g x z*
| *glue* [*intro*]: *pushout-rel f g (Inl (f c)) (Inr (g c))*

interpretation *pushout-equiv*: *equivalence-relation pushout-rel f g*
⟨*proof*⟩

definition *pushout-class* ::
('c => 'a) => ('c => 'b) => ('a + 'b) => ('a + 'b) set
where
pushout-class f g x = equiv-class (pushout-rel f g) x

definition *pushout-space* ::
('c => 'a) => ('c => 'b) => ('a + 'b) set set
where
pushout-space f g = quotient-space (pushout-rel f g)

definition *pushout-inl* ::
('c => 'a) => ('c => 'b) => 'a => ('a + 'b) set
where
pushout-inl f g a = pushout-class f g (Inl a)

definition *pushout-inr* ::
('c => 'a) => ('c => 'b) => 'b => ('a + 'b) set
where
pushout-inr f g b = pushout-class f g (Inr b)

lemma *pushout-class-eq-iff*:
pushout-class f g x = pushout-class f g y ⟷ pushout-rel f g x y

⟨*proof*⟩

lemma *pushout-inl-in-space* [*intro*]:
pushout-inl f g a ∈ *pushout-space f g*
⟨*proof*⟩

lemma *pushout-inr-in-space* [*intro*]:
pushout-inr f g b ∈ *pushout-space f g*
⟨*proof*⟩

lemma *pushout-glue*:
pushout-inl f g (*f c*) = *pushout-inr f g* (*g c*)
⟨*proof*⟩

definition *pushout-case-compatible* ::
('*c* => '*a*) => ('*c* => '*b*) => ('*a* => '*d*) => ('*b* => '*d*) => *bool*
where
pushout-case-compatible f g left right ⟷ (∀ *c*. *left* (*f c*) = *right* (*g c*))

lemma *pushout-rel-case-cong*:
assumes *compat*: *pushout-case-compatible f g left right*
and *rel*: *pushout-rel f g x y*
shows *case-sum left right x* = *case-sum left right y*
⟨*proof*⟩

definition *pushout-rec* ::
('*c* => '*a*) => ('*c* => '*b*) => ('*a* => '*d*) => ('*b* => '*d*) => ('*a* + '*b*) *set* =>
'*d*
where
pushout-rec f g left right X =
(*THE z*. ∃ *x*. *X* = *pushout-class f g x* ∧ *z* = *case-sum left right x*)

lemma *pushout-rec-well-defined*:
assumes *compat*: *pushout-case-compatible f g left right*
and *X-in*: *X* ∈ *pushout-space f g*
shows ∃! *z*. ∃ *x*. *X* = *pushout-class f g x* ∧ *z* = *case-sum left right x*
⟨*proof*⟩

lemma *pushout-rec-inl*:
assumes *compat*: *pushout-case-compatible f g left right*
shows *pushout-rec f g left right* (*pushout-inl f g a*) = *left a*
⟨*proof*⟩

lemma *pushout-rec-inr*:
assumes *compat*: *pushout-case-compatible f g left right*
shows *pushout-rec f g left right* (*pushout-inr f g b*) = *right b*
⟨*proof*⟩

end

theory *Seifert-Van-Kampen-Scaffold*
imports *Pushout-Scaffold Carrier-Amalgamated-Free-Product*
begin

14 Conditional Seifert–van Kampen interface

This theory packages a general quotient-level Seifert–van Kampen argument behind explicit encode/decode obligations. Once a decoding map factors through the full amalgamation quotient, and once the usual round-trip laws are available, the quotient-level bijection follows abstractly.

locale *svk-encode-decode-interface* =
fixes *i1* :: *'h* => *'a*::*group-add*
and *i2* :: *'h* => *'b*::*group-add*
and *encode* :: *'pi1-pushout* =>
(*'a*, *'b*) *free-product-word*
and *decode* :: (*'a*, *'b*) *free-product-word* => *'pi1-pushout*
assumes *decode-respects*:
full-amalg-equiv i1 i2 u v ==> *decode u* = *decode v*
and *decode-encode*:
decode (*encode x*) = *x*
and *encode-decode*:
full-amalg-equiv i1 i2 (*encode* (*decode w*)) *w*
begin

definition *svk-quotient-map* ::
'pi1-pushout => ((*'a*, *'b*) *free-product-word*) *set*
where
svk-quotient-map x = *full-amalg-class i1 i2* (*encode x*)

lemma *svk-quotient-map-in-space*:
svk-quotient-map x ∈ *full-amalgamated-free-product-space i1 i2*
⟨*proof*⟩

lemma *svk-quotient-map-injective*:
assumes *svk-quotient-map x* = *svk-quotient-map y*
shows *x* = *y*
⟨*proof*⟩

lemma *svk-quotient-map-surjective*:
assumes *Q* ∈ *full-amalgamated-free-product-space i1 i2*
shows ∃ *x*. *svk-quotient-map x* = *Q*
⟨*proof*⟩

theorem *seifert-van-kampen-bij-betw*:
bij-betw svk-quotient-map UNIV (*full-amalgamated-free-product-space i1 i2*)
⟨*proof*⟩

end

locale *carrier-svk-encode-decode-interface* =
  fixes *G1* :: *'a set*
    and *G2* :: *'b set*
    and *H* :: *'h set*
    and *i1* :: *'h => 'a*
    and *i2* :: *'h => 'b*
    and *mult1* :: *'a => 'a => 'a*
    and *one1* :: *'a*
    and *mult2* :: *'b => 'b => 'b*
    and *one2* :: *'b*
    and *encode* :: *'pi1-pushout => ('a, 'b) free-product-word*
    and *decode* :: *('a, 'b) free-product-word => 'pi1-pushout*
  assumes *encode-in-space*:
      fpw-in-space G1 G2 (encode x)
    and *decode-respects*:
     carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v ==> decode u = decode v
    and *decode-encode*:
      decode (encode x) = x
    and *encode-decode*:
     carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (encode (decode w)) w
begin

definition *carrier-svk-quotient-map* ::
  'pi1-pushout => (('a, 'b) free-product-word) set
where
  carrier-svk-quotient-map x =
    carrier-full-amalg-class G1 G2 H i1 i2 mult1 one1 mult2 one2 (encode x)

lemma *carrier-svk-quotient-map-in-space*:
  carrier-svk-quotient-map x ∈
     carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2
  ⟨*proof*⟩

lemma *carrier-svk-quotient-map-injective*:
  assumes *carrier-svk-quotient-map x = carrier-svk-quotient-map y*
  shows *x = y*
⟨*proof*⟩

lemma *carrier-svk-quotient-map-surjective*:
  assumes
     Q ∈ carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2
  shows ∃ *x*. *carrier-svk-quotient-map x = Q*
⟨*proof*⟩

theorem *carrier-seifert-van-kampen-bij-betw*:
bij-betw carrier-svk-quotient-map UNIV
(*carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2*)
⟨*proof*⟩

end

end
theory *Classical-Seifert-Van-Kampen*
imports
Carrier-Amalgamated-Free-Product-Eval
Explicit-Fundamental-Group-Scaffold
Seifert-Van-Kampen-Scaffold
begin

15 Classical Seifert–van Kampen for open unions

This theory specializes the general encode/decode interface to the classical open-cover hypotheses. Its long middle portion constructs encodings of loops by subdividing them into pieces that lie in U or V, proves invariance under refinement and homotopy, and then packages the resulting quotient as the carrier-based amalgamated free product of the three relevant fundamental groups.

lemma *path-top-of-setI*:
assumes *path p*
and *path-image p* ⊆ *S*
shows *pathin* (*top-of-set S*) *p*
⟨*proof*⟩

locale *classical-svk-setup* =
fixes *U* :: *'a*::*topological-space set*
and *V* :: *'a set*
and *x0* :: *'a*
assumes *U-open*: *open U*
and *V-open*: *open V*
and *x0-in*: *x0* ∈ *U* ∩ *V*
and *UV-path-connected*: *path-connected* (*U* ∩ *V*)
begin

abbreviation *W* **where** *W* ≡ *U* ∪ *V*

abbreviation *G1* **where** *G1* ≡ *fundamental-group-space U x0*
abbreviation *G2* **where** *G2* ≡ *fundamental-group-space V x0*
abbreviation *H* **where** *H* ≡ *fundamental-group-space* (*U* ∩ *V*) *x0*

abbreviation *mult1* **where** *mult1* ≡ *fundamental-group-mult U x0*
abbreviation *one1* **where** *one1* ≡ *fundamental-group-one U x0*

abbreviation *mult2* **where** *mult2* ≡ *fundamental-group-mult V x0*
abbreviation *one2* **where** *one2* ≡ *fundamental-group-one V x0*
abbreviation *multW* **where** *multW* ≡ *fundamental-group-mult W x0*
abbreviation *oneW* **where** *oneW* ≡ *fundamental-group-one W x0*

abbreviation *i1* **where** *i1* ≡ *fundamental-group-map* (*U* ∩ *V*) *x0 U x0 id*
abbreviation *i2* **where** *i2* ≡ *fundamental-group-map* (*U* ∩ *V*) *x0 V x0 id*
abbreviation *j1* **where** *j1* ≡ *fundamental-group-map U x0 W x0 id*
abbreviation *j2* **where** *j2* ≡ *fundamental-group-map V x0 W x0 id*

15.1 Carrier-side setup

The first part of the theory fixes the inclusion-induced maps between the three fundamental groups and packages them into the carrier-side evaluation locale used by the later decode map. This isolates the algebraic compatibility conditions that are immediate from the open-union inclusions.

lemma *x0-in-U* [*simp*]: *x0* ∈ *U*
⟨*proof*⟩

lemma *x0-in-V* [*simp*]: *x0* ∈ *V*
⟨*proof*⟩

lemma *x0-in-W* [*simp*]: *x0* ∈ *W*
⟨*proof*⟩

lemma *x0-in-UV* [*simp*]: *x0* ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *i1-in-G1*:
assumes *h* ∈ *H*
shows *i1 h* ∈ *G1*
⟨*proof*⟩

lemma *i2-in-G2*:
assumes *h* ∈ *H*
shows *i2 h* ∈ *G2*
⟨*proof*⟩

lemma *union-fundamental-group-maps-agree*:
assumes *h-in*: *h* ∈ *H*
shows *j1* (*i1 h*) = *j2* (*i2 h*)
⟨*proof*⟩

lemma *decode-locale*:
carrier-full-amalg-word-eval
G1 mult1 one1 (*fundamental-group-inv U x0*)
G2 mult2 one2 (*fundamental-group-inv V x0*)
H i1 i2
(*fundamental-group-space W x0*) *multW oneW* (*fundamental-group-inv W x0*)

j1 j2
⟨*proof*⟩

interpretation *decode*:
carrier-full-amalg-word-eval
G1 mult1 one1 fundamental-group-inv U x0
G2 mult2 one2 fundamental-group-inv V x0
H i1 i2
fundamental-group-space W x0 multW oneW fundamental-group-inv W x0
j1 j2
⟨*proof*⟩

abbreviation *svk-word-eval* **where** *svk-word-eval* ≡ *decode.carrier-full-amalg-eval*
abbreviation *svk-decode* **where** *svk-decode* ≡ *decode.carrier-full-amalg-decode*

lemma *svk-decode-in-space*:
svk-decode w ∈ *fundamental-group-space W x0*
⟨*proof*⟩

lemma *svk-decode-respects*:
assumes *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *svk-decode u* = *svk-decode v*
⟨*proof*⟩

lemma *svk-decode-eq-eval*:
assumes *fpw-in-space G1 G2 w*
shows *svk-decode w* = *svk-word-eval w*
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-left-context*:
assumes *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
and *a-in*: *a* ∈ *G1*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordLeft a u*) (*WordLeft a v*)
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-right-context*:
assumes *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
and *b-in*: *b* ∈ *G2*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordRight b u*) (*WordRight b v*)
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-left-pair-eq*:
assumes *a-in*: *a* ∈ *G1*
and *b-in*: *b* ∈ *G1*
and *ab-in*: *mult1 a b* ∈ *G1*
and *c-in*: *c* ∈ *G1*
and *d-in*: *d* ∈ *G1*

and *cd-in*: *mult1 c d* ∈ *G1*
and *rest-in*: *fpw-in-space G1 G2 rest*
and *eq*: *mult1 a b* = *mult1 c d*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordLeft a* (*WordLeft b rest*))
(*WordLeft c* (*WordLeft d rest*))
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-right-pair-eq*:
assumes *a-in*: *a* ∈ *G2*
and *b-in*: *b* ∈ *G2*
and *ab-in*: *mult2 a b* ∈ *G2*
and *c-in*: *c* ∈ *G2*
and *d-in*: *d* ∈ *G2*
and *cd-in*: *mult2 c d* ∈ *G2*
and *rest-in*: *fpw-in-space G1 G2 rest*
and *eq*: *mult2 a b* = *mult2 c d*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordRight a* (*WordRight b rest*))
(*WordRight c* (*WordRight d rest*))
⟨*proof*⟩

lemma *loop-subdivision-by-cover*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
shows ∃ *n*::*nat*. *0* < *n* ∧
(∀ *i*<*n*.
subpathin (*real i* / *real n*) (*real* (*Suc i*) / *real n*) *p* ‘ {*0*..*1*} ⊆ *U* ∨
subpathin (*real i* / *real n*) (*real* (*Suc i*) / *real n*) *p* ‘ {*0*..*1*} ⊆ *V*)
⟨*proof*⟩

definition *connector* :: *'a* ⇒ *real* ⇒ *'a* **where**
connector a =
(*if a* = *x0 then* (λ-. *x0*)
else (*SOME p*. *path p* ∧ *path-image p* ⊆ *U* ∩ *V* ∧ *pathstart p* = *x0* ∧ *pathfinish p* = *a*))

lemma *connector-x0* [*simp*]:
connector x0 = (λ-. *x0*)
⟨*proof*⟩

lemma *connector-witness*:
assumes *a-in*: *a* ∈ *U* ∩ *V*
shows ∃ *p*. *path p* ∧ *path-image p* ⊆ *U* ∩ *V* ∧ *pathstart p* = *x0* ∧ *pathfinish p* = *a*
⟨*proof*⟩

lemma *connector-path*:
assumes *a-in*: *a* ∈ *U* ∩ *V*
shows *path* (*connector a*)

⟨*proof*⟩

lemma *connector-image-subset*:
  assumes *a-in*: $a \in U \cap V$
  shows *path-image* (*connector a*) $\subseteq U \cap V$
⟨*proof*⟩

lemma *connector-start*:
  assumes *a-in*: $a \in U \cap V$
  shows *pathstart* (*connector a*) = *x0*
⟨*proof*⟩

lemma *connector-finish*:
  assumes *a-in*: $a \in U \cap V$
  shows *pathfinish* (*connector a*) = *a*
⟨*proof*⟩

lemmas *connector-spec* = *connector-path connector-image-subset connector-start connector-finish*

definition *segment-loop* :: (*real* ⇒ *'a*) ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *'a* **where**
  segment-loop p u v =
    (*connector* (*p u*) +++ *subpathin u v p*) +++ *reversepath* (*connector* (*p v*))

lemma *segment-loop-in-set*:
  assumes *p-path*: *path p*
    and *p-image*: *path-image p* ⊆ *W*
    and *uv01*: *u* ∈ {*0..1*} *v* ∈ {*0..1*}
    and *puv-in*: *p u* ∈ *U* ∩ *V p v* ∈ *U* ∩ *V*
    and *conn-u*: *path-image* (*connector* (*p u*)) ⊆ *S*
    and *conn-v*: *path-image* (*connector* (*p v*)) ⊆ *S*
    and *seg-in*: *subpathin u v p* ‘ {*0..1*} ⊆ *S*
    and *x0S*: *x0* ∈ *S*
  shows *segment-loop p u v* ∈ *loop-space S x0*
⟨*proof*⟩

lemma *segment-loop-in-U*:
  assumes *p-path*: *path p*
    and *p-image*: *path-image p* ⊆ *W*
    and *uv01*: *u* ∈ {*0..1*} *v* ∈ {*0..1*}
    and *puv-in*: *p u* ∈ *U* ∩ *V p v* ∈ *U* ∩ *V*
    and *seg-in*: *subpathin u v p* ‘ {*0..1*} ⊆ *U*
  shows *segment-loop p u v* ∈ *loop-space U x0*
⟨*proof*⟩

lemma *segment-loop-in-V*:
  assumes *p-path*: *path p*
    and *p-image*: *path-image p* ⊆ *W*
    and *uv01*: *u* ∈ {*0..1*} *v* ∈ {*0..1*}

and *puv-in*: $p\ u \in U \cap V\ p\ v \in U \cap V$
and *seg-in*: $subpathin\ u\ v\ p\ `\ \{0..1\} \subseteq V$
shows $segment\text{-}loop\ p\ u\ v \in loop\text{-}space\ V\ x0$
⟨*proof*⟩

lemma *segment-loop-in-W*:
assumes *p-path*: $path\ p$
and *p-image*: $path\text{-}image\ p \subseteq W$
and *uv01*: $u \in \{0..1\}\ v \in \{0..1\}$
and *puv-in*: $p\ u \in U \cap V\ p\ v \in U \cap V$
and *seg-in*: $subpathin\ u\ v\ p\ `\ \{0..1\} \subseteq W$
shows $segment\text{-}loop\ p\ u\ v \in loop\text{-}space\ W\ x0$
⟨*proof*⟩

lemma *path-subpathin*:
assumes $path\ p$
and $u \in \{0..1\}$
and $v \in \{0..1\}$
shows $path\ (subpathin\ u\ v\ p)$
⟨*proof*⟩

lemma *path-image-subpathin-subset*:
assumes $u \in \{0..1\}$
and $v \in \{0..1\}$
shows $path\text{-}image\ (subpathin\ u\ v\ p) \subseteq path\text{-}image\ p$
⟨*proof*⟩

lemma *reversepath-subpathin* [*simp*]:
$reversepath\ (subpathin\ u\ v\ p) = subpathin\ v\ u\ p$
⟨*proof*⟩

lemma *subpathin-refl* [*simp*]:
$subpathin\ u\ u\ p = (\lambda\text{-}.\ p\ u)$
⟨*proof*⟩

fun *svk-partition* :: $(real \Rightarrow 'a) \Rightarrow real\ list \Rightarrow bool\ list \Rightarrow bool$ **where**
$svk\text{-}partition\ p\ []\ bs = False$
| $svk\text{-}partition\ p\ [t]\ [] = (t = 1 \wedge p\ t \in U \cap V)$
| $svk\text{-}partition\ p\ [t]\ (b \# bs) = False$
| $svk\text{-}partition\ p\ (t \# u \# ts)\ [] = False$
| $svk\text{-}partition\ p\ (t \# u \# ts)\ (b \# bs) =$
$(t \in \{0..1\} \wedge p\ t \in U \cap V \wedge u \in \{0..1\} \wedge t < u \wedge$
$(if\ b\ then\ subpathin\ t\ u\ p\ `\ \{0..1\} \subseteq U\ else\ subpathin\ t\ u\ p\ `\ \{0..1\} \subseteq V) \wedge$
$svk\text{-}partition\ p\ (u \# ts)\ bs)$

15.2 Partitions and encoded loop words

A loop is encoded by subdividing the unit interval into pieces whose images lie entirely in U or entirely in V. The resulting bitstring records which side

each segment uses, and the partition word records the corresponding loop classes in the two factors of the amalgamated free product.

```
definition valid-partition :: (real ⇒ 'a) ⇒ real list ⇒ bool list ⇒ bool where
  valid-partition p ts bs ⟷ ts ≠ [] ∧ hd ts = 0 ∧ svk-partition p ts bs

fun cover-partition :: (real ⇒ 'a) ⇒ real list ⇒ bool list ⇒ bool where
  cover-partition p [t] [] = (t = 1)
| cover-partition p (t # u # ts) (b # bs) =
    (t ∈ {0..1} ∧ u ∈ {0..1} ∧ t < u ∧
      (if b then subpathin t u p ‘ {0..1} ⊆ U else subpathin t u p ‘ {0..1} ⊆ V) ∧
      cover-partition p (u # ts) bs)
| cover-partition p ts bs = False

fun rectangle-partition ::
  ((real × real) ⇒ 'a) ⇒ real ⇒ real ⇒ real list ⇒ bool list ⇒ bool
where
  rectangle-partition h c d [t] [] = (t = 1)
| rectangle-partition h c d (t # u # ts) (b # bs) =
    (t ∈ {0..1} ∧ u ∈ {0..1} ∧ t < u ∧
      (if b then h ‘ ({t..u} × {c..d}) ⊆ U else h ‘ ({t..u} × {c..d}) ⊆ V) ∧
      rectangle-partition h c d (u # ts) bs)
| rectangle-partition h c d ts bs = False

fun alternating-bits :: bool list ⇒ bool where
  alternating-bits [] = True
| alternating-bits [b] = True
| alternating-bits (b # c # bs) = (b ≠ c ∧ alternating-bits (c # bs))

fun partition-word ::
  (real ⇒ 'a) ⇒ real list ⇒ bool list ⇒
    ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word
where
  partition-word p (t # u # ts) (b # bs) =
    (if b then WordLeft (loop-class U x0 (segment-loop p t u))
     else WordRight (loop-class V x0 (segment-loop p t u)))
      (partition-word p (u # ts) bs)
| partition-word p ts bs = WordNil

fun partition-word-with-tail ::
  (real ⇒ 'a) ⇒ real list ⇒ bool list ⇒
    ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word ⇒
    ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word
where
  partition-word-with-tail p (t # u # ts) (b # bs) tail =
    (if b then WordLeft (loop-class U x0 (segment-loop p t u))
     else WordRight (loop-class V x0 (segment-loop p t u)))
      (partition-word-with-tail p (u # ts) bs tail)
| partition-word-with-tail p ts bs tail = tail
```

fun *bridge-word* ::
  bool ⇒ (*real* ⇒ *'a*) *set* ⇒
    ((*real* ⇒ *'a*) *set*, (*real* ⇒ *'a*) *set*) *free-product-word* ⇒
    ((*real* ⇒ *'a*) *set*, (*real* ⇒ *'a*) *set*) *free-product-word*
where
  bridge-word True h rest = *WordLeft* (*i1 h*) *rest*
| *bridge-word False h rest* = *WordRight* (*i2 h*) *rest*

fun *partition-loop* :: (*real* ⇒ *'a*) ⇒ *real list* ⇒ *real* ⇒ *'a* **where**
  partition-loop p (*t* # *u* # *ts*) = *segment-loop p t u* +++ *partition-loop p* (*u* # *ts*)
| *partition-loop p ts* = (λ-. *x0*)

lemma *partition-word-with-tail-nil* [*simp*]:
  partition-word-with-tail p ts bs WordNil = *partition-word p ts bs*
  ⟨*proof*⟩

lemma *subpathin-joinpaths-left-half* [*simp*]:
  subpathin 0 (*1* / *2*) (*p* +++ *q*) ‘ {*0*..*1*} = *p* ‘ {*0*..*1*}
⟨*proof*⟩

lemma *affine-unit-interval*:
  fixes *u v t* :: *real*
  assumes *u01*: *u* ∈ {*0*..*1*}
    and *v01*: *v* ∈ {*0*..*1*}
    and *t01*: *t* ∈ {*0*..*1*}
  shows (*v* − *u*) ∗ *t* + *u* ∈ {*0*..*1*}
⟨*proof*⟩

lemma *subpathin-joinpaths-tail-scaled-pointwise*:
  assumes *q0*: *pathstart q* = *pathfinish p*
    and *u01*: *u* ∈ {*0*..*1*}
    and *v01*: *v* ∈ {*0*..*1*}
    and *t01*: *t* ∈ {*0*..*1*}
  shows *subpathin* ((*1* + *u*) / *2*) ((*1* + *v*) / *2*) (*p* +++ *q*) *t* = *subpathin u v q t*
⟨*proof*⟩

lemma *subpathin-joinpaths-tail-scaled* [*simp*]:
  assumes *q0*: *pathstart q* = *pathfinish p*
    and *u01*: *u* ∈ {*0*..*1*}
    and *v01*: *v* ∈ {*0*..*1*}
  shows *subpathin* ((*1* + *u*) / *2*) ((*1* + *v*) / *2*) (*p* +++ *q*) ‘ {*0*..*1*} = *subpathin u v q* ‘ {*0*..*1*}
⟨*proof*⟩

lemma *homotopic-paths-join-left*:
  assumes *qr*: *homotopic-paths S q r*
    and *p-path*: *path p*
    and *p-img*: *path-image p* ⊆ *S*

and *pq*: *pathfinish p = pathstart q*
shows *homotopic-paths S* (*p +++ q*) (*p +++ r*)
⟨*proof*⟩

lemma *homotopic-paths-join-right*:
assumes *pq*: *homotopic-paths S p q*
and *r-path*: *path r*
and *r-img*: *path-image r* ⊆ *S*
and *qr*: *pathfinish p = pathstart r*
shows *homotopic-paths S* (*p +++ r*) (*q +++ r*)
⟨*proof*⟩

lemma *homotopic-paths-cancel-middle-local*:
assumes *r-path*: *path r*
and *r-img*: *path-image r* ⊆ *S*
and *c-path*: *path c*
and *c-img*: *path-image c* ⊆ *S*
and *s-path*: *path s*
and *s-img*: *path-image s* ⊆ *S*
and *rc*: *pathfinish r = pathfinish c*
and *cs*: *pathstart s = pathfinish c*
shows *homotopic-paths S* ((((*r +++ reversepath c*) *+++ c*) *+++ s*)) (*r +++ s*)
⟨*proof*⟩

lemma *segment-loop-base-full-in-set*:
assumes *p-loop*: *p* ∈ *loop-space S x0*
shows *homotopic-paths S* (*segment-loop p 0 1*) *p*
⟨*proof*⟩

lemma *segment-loop-joinpaths-head* [*simp*]:
assumes *p-loop*: *p* ∈ *loop-space S x0*
and *q-loop*: *q* ∈ *loop-space W x0*
shows *segment-loop* (*p +++ q*) *0* (*1 / 2*) = *segment-loop p 0 1*
⟨*proof*⟩

lemma *segment-loop-joinpaths-tail-scaled* [*simp*]:
assumes *p-loop*: *p* ∈ *loop-space W x0*
and *q-loop*: *q* ∈ *loop-space W x0*
and *u01*: *u* ∈ {*0..1*}
and *v01*: *v* ∈ {*0..1*}
shows *segment-loop* (*p +++ q*) ((*1 + u*) */ 2*) ((*1 + v*) */ 2*) = *segment-loop q u v*
⟨*proof*⟩

fun *word-loop* ::
((*real* ⇒ *'a*) *set*, (*real* ⇒ *'a*) *set*) *free-product-word* ⇒ *real* ⇒ *'a*
where
word-loop WordNil = (λ-. *x0*)

```
| word-loop (WordLeft a rest) =
    (if rest = WordNil then some-loop U x0 a else some-loop U x0 a +++ word-loop
rest)
| word-loop (WordRight b rest) =
    (if rest = WordNil then some-loop V x0 b else some-loop V x0 b +++ word-loop
rest)

fun word-partition-times ::
  ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word ⇒ real list
where
  word-partition-times WordNil = [0, 1]
| word-partition-times (WordLeft a rest) =
    (if rest = WordNil
     then [0, 1]
     else 0 # map (λt. (1 + t) / 2) (word-partition-times rest))
| word-partition-times (WordRight b rest) =
    (if rest = WordNil
     then [0, 1]
     else 0 # map (λt. (1 + t) / 2) (word-partition-times rest))

fun word-partition-bits ::
  ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word ⇒ bool list
where
  word-partition-bits WordNil = [True]
| word-partition-bits (WordLeft a rest) =
    (if rest = WordNil then [True] else True # word-partition-bits rest)
| word-partition-bits (WordRight b rest) =
    (if rest = WordNil then [False] else False # word-partition-bits rest)

lemma joinpaths-tail-scaled-point [simp]:
  assumes p-loop: p ∈ loop-space W x0
    and q-loop: q ∈ loop-space W x0
    and t01: t ∈ {0..1}
  shows (p +++ q) ((1 + t) / 2) = q t
⟨proof⟩

lemma word-loop-in-W:
  assumes w-in: fpw-in-space G1 G2 w
  shows word-loop w ∈ loop-space W x0
  ⟨proof⟩

lemma svk-partition-joinpaths-tail-scaled:
  assumes p-loop: p ∈ loop-space W x0
    and q-loop: q ∈ loop-space W x0
    and q-part: svk-partition q ts bs
  shows svk-partition (p +++ q) (map (λt. (1 + t) / 2) ts) bs
  ⟨proof⟩

lemma word-loop-valid-partition:
```

assumes *w-in*: *fpw-in-space G1 G2 w*
shows *valid-partition* (*word-loop w*) (*word-partition-times w*) (*word-partition-bits w*)
⟨*proof*⟩

lemma *partition-word-joinpaths-tail-scaled*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *q-loop*: $q \in$ *loop-space W x0*
and *q-part*: *svk-partition q ts bs*
shows *partition-word* (p +++ q) (*map* ($\lambda t.\ (1 + t)\ /\ 2$) *ts*) *bs* = *partition-word q ts bs*
⟨*proof*⟩

lemma *segment-loop-some-loop-left-class*:
assumes *a-in*: $a \in$ *G1*
shows *loop-class U x0* (*segment-loop* (*some-loop U x0 a*) *0 1*) = *a*
⟨*proof*⟩

lemma *segment-loop-some-loop-right-class*:
assumes *b-in*: $b \in$ *G2*
shows *loop-class V x0* (*segment-loop* (*some-loop V x0 b*) *0 1*) = *b*
⟨*proof*⟩

lemma *partition-word-word-loop-equiv*:
assumes *w-in*: *fpw-in-space G1 G2 w*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word* (*word-loop w*) (*word-partition-times w*) (*word-partition-bits w*)) *w*
⟨*proof*⟩

lemma *valid-partition-hd*:
assumes *valid-partition p ts bs*
shows *ts* ≠ [] *hd ts* = *0*
⟨*proof*⟩

lemma *valid-partition-cases*:
assumes *valid-partition p* (*t* # *ts*) *bs*
shows *t* = *0* **and** *svk-partition p* (*t* # *ts*) *bs*
⟨*proof*⟩

lemma *svk-partition-head-props*:
assumes *svk-partition p* (*t* # *ts*) *bs*
shows *t* ∈ {*0..1*} **and** *p t* ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *svk-partition-tail*:
assumes *svk-partition p* (*t* # *u* # *ts*) (*b* # *bs*)
shows *svk-partition p* (*u* # *ts*) *bs*
⟨*proof*⟩

lemma *svk-partition-step-props*:
assumes *svk-partition p* (*t # u # ts*) (*b # bs*)
shows *t* ∈ {*0..1*}
and *p t* ∈ *U* ∩ *V*
and *u* ∈ {*0..1*}
and *t* < *u*
and (*if b then subpathin t u p* ‘ {*0..1*} ⊆ *U else subpathin t u p* ‘ {*0..1*} ⊆ *V*)
⟨*proof*⟩

lemma *svk-partition-next-in-intersection*:
assumes *svk-partition p* (*t # u # ts*) (*b # bs*)
shows *p u* ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *svk-partition-nonempty*:
assumes *svk-partition p ts bs*
shows *ts* ≠ []
⟨*proof*⟩

lemma *svk-partition-last-eq-one*:
assumes *part*: *svk-partition p ts bs*
shows *last ts* = *1*
⟨*proof*⟩

lemma *svk-partition-last-in-intersection*:
assumes *part*: *svk-partition p ts bs*
shows *p* (*last ts*) ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *svk-partition-last-props*:
assumes *part*: *svk-partition p ts bs*
shows *ts* ≠ [] **and** *last ts* = *1* **and** *p* (*last ts*) ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *valid-partition-last-props*:
assumes *valid-partition p ts bs*
shows *ts* ≠ [] **and** *last ts* = *1* **and** *p* (*last ts*) ∈ *U* ∩ *V*
⟨*proof*⟩

lemma *subpathin-endpoints-in-set*:
assumes *seg-in*: *subpathin u v p* ‘ {*0..1*} ⊆ *S*
shows *p u* ∈ *S* **and** *p v* ∈ *S*
⟨*proof*⟩

lemma *subpathin-image-subset-union*:
assumes *tu*: *t* ≤ *u*
and *uv*: *u* ≤ *v*
shows *subpathin t v p* ‘ {*0..1*} ⊆ *subpathin t u p* ‘ {*0..1*} ∪ *subpathin u v p* ‘

$\{0..1\}$
⟨*proof*⟩

lemma *subpathin-image-subset-trans*:
  assumes *tu*: $t \leq u$
    and *uv*: $u \leq v$
    and *left*: *subpathin t u p* ‘ $\{0..1\} \subseteq S$
    and *right*: *subpathin u v p* ‘ $\{0..1\} \subseteq S$
  shows *subpathin t v p* ‘ $\{0..1\} \subseteq S$
  ⟨*proof*⟩

lemma *cover-partition-step-props*:
  assumes *cover-partition p* $(t \# u \# ts)$ $(b \# bs)$
  shows $t \in \{0..1\}$
    and $u \in \{0..1\}$
    and $t < u$
    and (*if b then subpathin t u p* ‘ $\{0..1\} \subseteq U$ *else subpathin t u p* ‘ $\{0..1\} \subseteq V$)
    and *cover-partition p* $(u \# ts)$ *bs*
  ⟨*proof*⟩

lemma *cover-partition-consI*:
  assumes $t \in \{0..1\}$
    and $u \in \{0..1\}$
    and $t < u$
    and (*if b then subpathin t u p* ‘ $\{0..1\} \subseteq U$ *else subpathin t u p* ‘ $\{0..1\} \subseteq V$)
    and *cover-partition p* $(u \# ts)$ *bs*
  shows *cover-partition p* $(t \# u \# ts)$ $(b \# bs)$
  ⟨*proof*⟩

lemma *cover-partition-switch-point*:
  assumes *cp*: *cover-partition p* $(t \# u \# v \# ts)$ $(b \# c \# bs)$
    and *diff*: $b \neq c$
  shows $p\ u \in U \cap V$
⟨*proof*⟩

lemma *cover-partition-pair-svk-partition*:
  assumes *cp*: *cover-partition p* $[t, u]$ $[b]$
    and *ptUV*: $p\ t \in U \cap V$
    and *puUV*: $p\ u \in U \cap V$
  shows *svk-partition p* $[t, u]$ $[b]$
⟨*proof*⟩

lemma *cover-partition-compress-svk-partition*:
  assumes *cp*: *cover-partition p* $(t \# ts)$ *bs*
    and *ptUV*: $p\ t \in U \cap V$
    and *plastUV*: p (*last* $(t \# ts)) \in U \cap V$
  shows $\exists\, ts'\ bs'.$ *svk-partition p* $(t \# ts')$ bs'
  ⟨*proof*⟩

lemma *cover-partition-last-eq-one*:
assumes *cp*: *cover-partition p ts bs*
and *ts-ne*: *ts* $\neq$ []
shows *last ts* = *1*
⟨*proof*⟩

lemma *nat-real-div-in-unit-interval*:
assumes *n-pos*: *0* < *n*
and *i-le*: *i* ≤ *n*
shows *real i* / *real n* ∈ {*0..1*}
⟨*proof*⟩

lemma *nat-real-div-strict-mono*:
assumes *n-pos*: *0* < *n*
and *i-lt*: *i* < *n*
shows *real i* / *real n* < *real* (*Suc i*) / *real n*
⟨*proof*⟩

fun *subdivision-times* :: *nat* ⇒ *nat* ⇒ *real list* **where**
subdivision-times n 0 = [*1*]
| *subdivision-times n* (*Suc k*) = *real* (*n* − *Suc k*) / *real n* # *subdivision-times n k*

fun *subdivision-bits* :: (*nat* ⇒ *bool*) ⇒ *nat* ⇒ *nat* ⇒ *bool list* **where**
subdivision-bits side n 0 = []
| *subdivision-bits side n* (*Suc k*) = *side* (*n* − *Suc k*) # *subdivision-bits side n k*

lemma *cover-partition-subdivision-from*:
assumes *n-pos*: *0* < *n*
and *k-le*: *k* ≤ *n*
and *cover*: ⋀*i*. *n* − *k* ≤ *i* ⟹ *i* < *n* ⟹
(*if side i*
then subpathin (*real i* / *real n*) (*real* (*Suc i*) / *real n*) *p* ‘ {*0..1*} ⊆ *U*
else subpathin (*real i* / *real n*) (*real* (*Suc i*) / *real n*) *p* ‘ {*0..1*} ⊆ *V*)
shows *cover-partition p* (*subdivision-times n k*) (*subdivision-bits side n k*)
⟨*proof*⟩

lemma *subdivision-times-start*:
assumes *n-pos*: *0* < *n*
shows *subdivision-times n n* = *0* # *subdivision-times n* (*n* − *1*)
⟨*proof*⟩

lemma *loop-has-valid-partition*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
shows ∃ *ts bs*. *valid-partition p ts bs*
⟨*proof*⟩

lemma *svk-partition-partition-word-in-space*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
and *part*: *svk-partition p ts bs*

shows *fpw-in-space G1 G2* (*partition-word p ts bs*)
⟨*proof*⟩

lemma *valid-partition-partition-word-in-space*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *fpw-in-space G1 G2* (*partition-word p ts bs*)
⟨*proof*⟩

lemma *svk-partition-partition-loop-in-W*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *svk-partition p ts bs*
shows *partition-loop p ts* $\in$ *loop-space W x0*
⟨*proof*⟩

lemma *valid-partition-partition-loop-in-W*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *partition-loop p ts* $\in$ *loop-space W x0*
⟨*proof*⟩

lemma *i1-loop-class-eq*:
assumes *p-loop*: $p \in$ *loop-space* $(U \cap V)$ *x0*
shows *i1* (*loop-class* $(U \cap V)$ *x0 p*) = *loop-class U x0 p*
⟨*proof*⟩

lemma *i2-loop-class-eq*:
assumes *p-loop*: $p \in$ *loop-space* $(U \cap V)$ *x0*
shows *i2* (*loop-class* $(U \cap V)$ *x0 p*) = *loop-class V x0 p*
⟨*proof*⟩

lemma *j1-segment-loop-eq*:
assumes *segU*: *segment-loop p t u* $\in$ *loop-space U x0*
shows *j1* (*loop-class U x0* (*segment-loop p t u*)) =
loop-class W x0 (*segment-loop p t u*)
⟨*proof*⟩

lemma *j2-segment-loop-eq*:
assumes *segV*: *segment-loop p t u* $\in$ *loop-space V x0*
shows *j2* (*loop-class V x0* (*segment-loop p t u*)) =
loop-class W x0 (*segment-loop p t u*)
⟨*proof*⟩

lemma *svk-partition-eval-partition-word*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *svk-partition p ts bs*
shows *svk-word-eval* (*partition-word p ts bs*) =
loop-class W x0 (*partition-loop p ts*)
⟨*proof*⟩

lemma *valid-partition-eval-partition-word*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *svk-word-eval* (*partition-word p ts bs*) =
loop-class W x0 (*partition-loop p ts*)
⟨*proof*⟩

lemma *valid-partition-decode-partition-word*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *svk-decode* (*partition-word p ts bs*) =
loop-class W x0 (*partition-loop p ts*)
⟨*proof*⟩

lemma *pair-interval-member*:
fixes *x y* :: *real* × *real* **and** *x1 x2 y1 y2 u v* :: *real*
assumes *x*: $x = (x1, x2)$
and *y*: $y = (y1, y2)$
and *mix1*: $u *_R x1 + v *_R y1 \in \{0..1\}$
and *mix2*: $u *_R x2 + v *_R y2 \in \{0..1\}$
shows $u *_R x + v *_R y \in \{0..1\} \times \{0..1\}$
⟨*proof*⟩

lemma *affine-closed-segment-member*:
fixes *a b u* :: *real*
assumes *u01*: $u \in \{0..1\}$
shows $(b - a) * u + a \in$ *closed-segment a b*
⟨*proof*⟩

lemma *affine-subinterval-member*:
fixes *a b u* :: *real*
assumes *ab*: $a \leq b$
and *u01*: $u \in \{0..1\}$
shows $(b - a) * u + a \in \{a..b\}$
⟨*proof*⟩

lemma *affine-unit-interval-member*:
fixes *a b u* :: *real*
assumes *a01*: $a \in \{0..1\}$
and *b01*: $b \in \{0..1\}$
and *ab*: $a \leq b$
and *u01*: $u \in \{0..1\}$
shows $(b - a) * u + a \in \{0..1\}$
⟨*proof*⟩

lemma *square-edge-homotopic*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set S*) *h*

shows *homotopic-paths S*
((λ*t*. *h* (*t*, *0*)) +++ (λ*t*. *h* (*1*, *t*)))
((λ*t*. *h* (*0*, *t*)) +++ (λ*t*. *h* (*t*, *1*)))
⟨*proof*⟩

lemma *rectangle-edge-homotopic*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set S*) *h*
and *a01*: *a* ∈ {*0..1*} **and** *b01*: *b* ∈ {*0..1*}
and *c01*: *c* ∈ {*0..1*} **and** *d01*: *d* ∈ {*0..1*}
and *ab*: *a* ≤ *b* **and** *cd*: *c* ≤ *d*
shows *homotopic-paths S*
(*subpathin a b* (λ*t*. *h* (*t*, *c*)) +++ (λ*t*. *h* (*b*, (*d* − *c*) ∗ *t* + *c*)))
((λ*t*. *h* (*a*, (*d* − *c*) ∗ *t* + *c*)) +++ *subpathin a b* (λ*t*. *h* (*t*, *d*)))
⟨*proof*⟩

lemma *rectangle-edge-homotopic-in-set*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *a01*: *a* ∈ {*0..1*} **and** *b01*: *b* ∈ {*0..1*}
and *c01*: *c* ∈ {*0..1*} **and** *d01*: *d* ∈ {*0..1*}
and *ab*: *a* ≤ *b* **and** *cd*: *c* ≤ *d*
and *rectS*: *h* ‘ ({*a..b*} × {*c..d*}) ⊆ *S*
shows *homotopic-paths S*
(*subpathin a b* (λ*t*. *h* (*t*, *c*)) +++ (λ*t*. *h* (*b*, (*d* − *c*) ∗ *t* + *c*)))
((λ*t*. *h* (*a*, (*d* − *c*) ∗ *t* + *c*)) +++ *subpathin a b* (λ*t*. *h* (*t*, *d*)))
⟨*proof*⟩

definition *vertical-strip-path* ::
((*real* × *real*) ⇒ *'a*) ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *'a*
where
vertical-strip-path h t c d = (λ*u*. *h* (*t*, (*d* − *c*) ∗ *u* + *c*))

definition *bridge-loop* ::
((*real* × *real*) ⇒ *'a*) ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *'a*
where
bridge-loop h t c d =
(*connector* (*h* (*t*, *c*)) +++ *vertical-strip-path h t c d*) +++
reversepath (*connector* (*h* (*t*, *d*)))

lemma *bridge-loop-eq-segment-loop* [*simp*]:
bridge-loop h t c d = *segment-loop* (*vertical-strip-path h t c d*) *0 1*
⟨*proof*⟩

lemma *vertical-strip-path-image-subset*:
assumes *cd*: *c* ≤ *d*
shows *vertical-strip-path h t c d* ‘ {*0..1*} ⊆ *h* ‘ ({*t*} × {*c..d*})
⟨*proof*⟩

lemma *rectangle-segment-loop-bridge-homotopic*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* W) h
and *a01*: $a \in \{0..1\}$ **and** *b01*: $b \in \{0..1\}$
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$
and *ab*: $a \leq b$ **and** *cd*: $c \leq d$
and *rectS*: $h \text{ ` } (\{a..b\} \times \{c..d\}) \subseteq S$
and *leftUV*: $h \text{ ` } (\{a\} \times \{c..d\}) \subseteq U \cap V$
and *rightUV*: $h \text{ ` } (\{b\} \times \{c..d\}) \subseteq U \cap V$
and *UVS*: $U \cap V \subseteq S$
shows *homotopic-paths* S
(*segment-loop* ($\lambda t.\ h\ (t,\ c)$) $a\ b$ +++ *bridge-loop* $h\ b\ c\ d$)
(*bridge-loop* $h\ a\ c\ d$ +++ *segment-loop* ($\lambda t.\ h\ (t,\ d)$) $a\ b$)
⟨*proof*⟩

lemma *horizontal-rectangle-segment-loop-in-set*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* W) h
and *a01*: $a \in \{0..1\}$ **and** *b01*: $b \in \{0..1\}$ **and** *c01*: $c \in \{0..1\}$
and *ab*: $a \leq b$
and *segS*: $h \text{ ` } (\{a..b\} \times \{c\}) \subseteq S$
and *acUV*: $h\ (a,\ c) \in U \cap V$
and *bcUV*: $h\ (b,\ c) \in U \cap V$
and *UVS*: $U \cap V \subseteq S$
shows *segment-loop* ($\lambda t.\ h\ (t,\ c)$) $a\ b \in$ *loop-space* S *x0*
⟨*proof*⟩

lemma *vertical-bridge-loop-in-set*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* W) h
and *t01*: $t \in \{0..1\}$ **and** *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$
and *cd*: $c \leq d$
and *edgeUV*: $h \text{ ` } (\{t\} \times \{c..d\}) \subseteq U \cap V$
and *UVS*: $U \cap V \subseteq S$
shows *bridge-loop* $h\ t\ c\ d \in$ *loop-space* S *x0*
⟨*proof*⟩

lemma *rectangle-segment-loop-bridge-class-eq*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* W) h
and *a01*: $a \in \{0..1\}$ **and** *b01*: $b \in \{0..1\}$
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$
and *ab*: $a \leq b$ **and** *cd*: $c \leq d$
and *rectS*: $h \text{ ` } (\{a..b\} \times \{c..d\}) \subseteq S$

and *leftUV*: h ‘ $(\{a\} \times \{c..d\}) \subseteq U \cap V$
and *rightUV*: h ‘ $(\{b\} \times \{c..d\}) \subseteq U \cap V$
and *UVS*: $U \cap V \subseteq S$
shows *fundamental-group-mult S x0*
(*loop-class S x0* (*segment-loop* ($\lambda t.$ h $(t,$ $c)$) a b))
(*loop-class S x0* (*bridge-loop h b c d*)) =
fundamental-group-mult S x0
(*loop-class S x0* (*bridge-loop h a c d*))
(*loop-class S x0* (*segment-loop* ($\lambda t.$ h $(t,$ $d)$) a b))
⟨*proof*⟩

lemma *bridge-word-identify*:
assumes *h-in*: $h \in H$
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*bridge-word True h rest*) (*bridge-word False h rest*)
and *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*bridge-word False h rest*) (*bridge-word True h rest*)
⟨*proof*⟩

lemma *rectangle-partition-step-props*:
assumes *rectangle-partition h c d* $(t \# u \# ts)$ $(b \# bs)$
shows $t \in \{0..1\}$
and $u \in \{0..1\}$
and $t < u$
and (*if b then* h ‘ $(\{t..u\} \times \{c..d\}) \subseteq U$ *else* h ‘ $(\{t..u\} \times \{c..d\}) \subseteq V$)
and *rectangle-partition h c d* $(u \# ts)$ *bs*
⟨*proof*⟩

lemma *rectangle-partition-switch-edge*:
assumes *rp*: *rectangle-partition h c d* $(t \# u \# v \# ts)$ $(b \# e \# bs)$
and *diff*: $b \neq e$
shows h ‘ $(\{u\} \times \{c..d\}) \subseteq U \cap V$
⟨*proof*⟩

lemma *carrier-full-amalg-equiv-side-context*:
assumes *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
and *a-in*: (*if b then a* $\in$ *G1 else a* $\in$ *G2*)
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*if b then WordLeft a u else WordRight a u*)
(*if b then WordLeft a v else WordRight a v*)
⟨*proof*⟩

lemma *bridge-word-switch*:
assumes *h-in*: $h \in H$
and *bc*: $b \neq c$
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*bridge-word b h rest*) (*bridge-word c h rest*)
⟨*proof*⟩

lemma *rectangle-segment-bridge-left-equiv*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* *W*) *h*
and *a01*: $a \in \{0..1\}$ **and** *b01*: $b \in \{0..1\}$
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$
and *ab*: $a \leq b$ **and** *cd*: $c \leq d$
and *rectU*: $h\ `\ (\{a..b\} \times \{c..d\}) \subseteq U$
and *leftUV*: $h\ `\ (\{a\} \times \{c..d\}) \subseteq U \cap V$
and *rightUV*: $h\ `\ (\{b\} \times \{c..d\}) \subseteq U \cap V$
and *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordLeft* (*loop-class U x0* (*segment-loop* ($\lambda t.$ *h* (*t*, *c*)) *a b*))
(*bridge-word True* (*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h b c d*)) *rest*))
(*bridge-word True* (*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h a c d*))
(*WordLeft* (*loop-class U x0* (*segment-loop* ($\lambda t.$ *h* (*t*, *d*)) *a b*)) *rest*))
⟨*proof*⟩

lemma *rectangle-segment-bridge-right-equiv*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* *W*) *h*
and *a01*: $a \in \{0..1\}$ **and** *b01*: $b \in \{0..1\}$
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$
and *ab*: $a \leq b$ **and** *cd*: $c \leq d$
and *rectV*: $h\ `\ (\{a..b\} \times \{c..d\}) \subseteq V$
and *leftUV*: $h\ `\ (\{a\} \times \{c..d\}) \subseteq U \cap V$
and *rightUV*: $h\ `\ (\{b\} \times \{c..d\}) \subseteq U \cap V$
and *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordRight* (*loop-class V x0* (*segment-loop* ($\lambda t.$ *h* (*t*, *c*)) *a b*))
(*bridge-word False* (*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h b c d*)) *rest*))
(*bridge-word False* (*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h a c d*))
(*WordRight* (*loop-class V x0* (*segment-loop* ($\lambda t.$ *h* (*t*, *d*)) *a b*)) *rest*))
⟨*proof*⟩

lemma *rectangle-partition-partition-word-with-tail-in-space*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* *W*) *h*
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$ **and** *cd*: $c \leq d$
and *y-in*: $y \in \{c, d\}$
and *part*: *rectangle-partition h c d ts bs*
and *edgeUV*: $\bigwedge t.\ t \in set\ ts \Longrightarrow h\ `\ (\{t\} \times \{c..d\}) \subseteq U \cap V$
and *tail-in*: *fpw-in-space G1 G2 rest*
shows *fpw-in-space G1 G2* (*partition-word-with-tail* ($\lambda t.$ *h* (*t*, *y*)) *ts bs rest*)
⟨*proof*⟩

lemma *rectangle-partition-partition-word-with-tail-equiv*:

fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set* *W*) *h*
and *c01*: $c \in \{0..1\}$ **and** *d01*: $d \in \{0..1\}$ **and** *cd*: $c \leq d$
and *part*: *rectangle-partition h c d* (*t* # *u* # *ts*) (*b* # *bs*)
and *edgeUV*: $\bigwedge x.\ x \in set\ (t\ \#\ u\ \#\ ts) \Longrightarrow h\ `\ (\{x\} \times \{c..d\}) \subseteq U \cap V$
and *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word-with-tail* ($\lambda x.\ h\ (x,\ c)$) (*t* # *u* # *ts*) (*b* # *bs*)
(*bridge-word* (*last* (*b* # *bs*))
(*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h* (*last* (*t* # *u* # *ts*)) *c d*)) *rest*))
(*bridge-word b* (*loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h t c d*))
(*partition-word-with-tail* ($\lambda x.\ h\ (x,\ d)$) (*t* # *u* # *ts*) (*b* # *bs*) *rest*))
⟨*proof*⟩

lemma *homotopic-paths-join-subpathin*:
assumes *p-path*: *path p*
and *p-img*: *path-image p* ⊆ *S*
and *u01*: $u \in \{0..1\}$
and *v01*: $v \in \{0..1\}$
and *w01*: $w \in \{0..1\}$
and *uv*: $u \leq v$
and *vw*: $v \leq w$
shows *homotopic-paths S* (*subpathin u v p* +++ *subpathin v w p*) (*subpathin u w p*)
⟨*proof*⟩

lemma *subpathin-full* [*simp*]:
subpathin 0 1 p = *p*
⟨*proof*⟩

lemma *segment-loop-refl*:
assumes *puUV*: *p u* ∈ *U* ∩ *V*
shows *homotopic-paths W* (*segment-loop p u u*) (λ-. *x0*)
⟨*proof*⟩

lemma *segment-loop-full*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
shows *homotopic-paths W* (*segment-loop p 0 1*) *p*
⟨*proof*⟩

lemma *homotopic-paths-cancel-middle*:
assumes *r-path*: *path r*
and *r-img*: *path-image r* ⊆ *S*
and *c-path*: *path c*
and *c-img*: *path-image c* ⊆ *S*
and *s-path*: *path s*
and *s-img*: *path-image s* ⊆ *S*
and *rc*: *pathfinish r* = *pathfinish c*

and *cs*: *pathstart s = pathfinish c*
shows *homotopic-paths S ((((r +++ reversepath c) +++ c) +++ s)) (r +++ s)*
⟨*proof*⟩

lemma *segment-loop-join*:
assumes *p-path*: *path p*
and *p-img*: *path-image p ⊆ W*
and *u01*: *u ∈ {0..1}*
and *v01*: *v ∈ {0..1}*
and *w01*: *w ∈ {0..1}*
and *uv*: *u ≤ v*
and *vw*: *v ≤ w*
and *puUV*: *p u ∈ U ∩ V*
and *pvUV*: *p v ∈ U ∩ V*
and *pwUV*: *p w ∈ U ∩ V*
shows *homotopic-paths W (segment-loop p u v +++ segment-loop p v w) (segment-loop p u w)*
⟨*proof*⟩

lemma *segment-loop-join-in-set*:
assumes *p-path*: *path p*
and *p-imgS*: *path-image p ⊆ S*
and *SW*: *S ⊆ W*
and *UVS*: *U ∩ V ⊆ S*
and *u01*: *u ∈ {0..1}*
and *v01*: *v ∈ {0..1}*
and *w01*: *w ∈ {0..1}*
and *uv*: *u ≤ v*
and *vw*: *v ≤ w*
and *puUV*: *p u ∈ U ∩ V*
and *pvUV*: *p v ∈ U ∩ V*
and *pwUV*: *p w ∈ U ∩ V*
shows *homotopic-paths S (segment-loop p u v +++ segment-loop p v w) (segment-loop p u w)*
⟨*proof*⟩

lemma *partition-loop-nil* [*simp*]:
partition-loop p [] = (λ-. x0)
⟨*proof*⟩

lemma *partition-loop-singleton* [*simp*]:
partition-loop p [t] = (λ-. x0)
⟨*proof*⟩

lemma *svk-partition-partition-loop-homotopic-segment-loop*:
assumes *p-loop*: *p ∈ loop-space W x0*
and *part*: *svk-partition p (t # ts) bs*
shows *homotopic-paths W (partition-loop p (t # ts)) (segment-loop p t 1)*

⟨*proof*⟩

lemma *valid-partition-partition-loop-homotopic-segment-loop*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *homotopic-paths W* (*partition-loop p ts*) (*segment-loop p 0 1*)
⟨*proof*⟩

lemma *valid-partition-partition-loop-homotopic*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *homotopic-paths W* (*partition-loop p ts*) *p*
⟨*proof*⟩

lemma *valid-partition-partition-loop-eq*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *loop-class W x0* (*partition-loop p ts*) = *loop-class W x0 p*
⟨*proof*⟩

lemma *valid-partition-decode-partition-word-eq-loop-class*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p ts bs*
shows *svk-decode* (*partition-word p ts bs*) = *loop-class W x0 p*
⟨*proof*⟩

lemma *subpathin-image-subset-left*:
assumes *t01*: $t \in \{0..1\}$
and *u01*: $u \in \{0..1\}$
and *v01*: $v \in \{0..1\}$
and *tu*: $t \leq u$
and *uv*: $u \leq v$
shows *subpathin t u p* ‘ $\{0..1\} \subseteq$ *subpathin t v p* ‘ $\{0..1\}$
⟨*proof*⟩

lemma *subpathin-image-subset-right*:
assumes *t01*: $t \in \{0..1\}$
and *u01*: $u \in \{0..1\}$
and *v01*: $v \in \{0..1\}$
and *tu*: $t \leq u$
and *uv*: $u \leq v$
shows *subpathin u v p* ‘ $\{0..1\} \subseteq$ *subpathin t v p* ‘ $\{0..1\}$
⟨*proof*⟩

lemma *subpathin-subpathin*:
subpathin a b (*subpathin u v p*) =
subpathin $(((v - u) * a) + u)$ $(((v - u) * b) + u)$ *p*
⟨*proof*⟩

lemma *segment-loop-subpathin*:
  segment-loop (*subpathin u v p*) *a b* =
    segment-loop p (((*v* − *u*) ∗ *a*) + *u*) (((*v* − *u*) ∗ *b*) + *u*)
  ⟨*proof*⟩

lemma *segment-loop-mult-eq-left*:
  assumes *p-path*: *path p*
    and *p-imgW*: *path-image p* ⊆ *W*
    and *t01*: *t* ∈ {*0..1*}
    and *u01*: *u* ∈ {*0..1*}
    and *v01*: *v* ∈ {*0..1*}
    and *tu*: *t* < *u*
    and *uv*: *u* < *v*
    and *ptUV*: *p t* ∈ *U* ∩ *V*
    and *puUV*: *p u* ∈ *U* ∩ *V*
    and *pvUV*: *p v* ∈ *U* ∩ *V*
    and *seg-tvU*: *subpathin t v p* ‘ {*0..1*} ⊆ *U*
  shows *mult1* (*loop-class U x0* (*segment-loop p t u*))
      (*loop-class U x0* (*segment-loop p u v*)) =
    loop-class U x0 (*segment-loop p t v*)
⟨*proof*⟩

lemma *segment-loop-mult-eq-right*:
  assumes *p-path*: *path p*
    and *p-imgW*: *path-image p* ⊆ *W*
    and *t01*: *t* ∈ {*0..1*}
    and *u01*: *u* ∈ {*0..1*}
    and *v01*: *v* ∈ {*0..1*}
    and *tu*: *t* < *u*
    and *uv*: *u* < *v*
    and *ptUV*: *p t* ∈ *U* ∩ *V*
    and *puUV*: *p u* ∈ *U* ∩ *V*
    and *pvUV*: *p v* ∈ *U* ∩ *V*
    and *seg-tvV*: *subpathin t v p* ‘ {*0..1*} ⊆ *V*
  shows *mult2* (*loop-class V x0* (*segment-loop p t u*))
      (*loop-class V x0* (*segment-loop p u v*)) =
    loop-class V x0 (*segment-loop p t v*)
⟨*proof*⟩

lemma *segment-word-split-left-equiv*:
  assumes *p-loop*: *p* ∈ *loop-space W x0*
    and *t01*: *t* ∈ {*0..1*}
    and *u01*: *u* ∈ {*0..1*}
    and *v01*: *v* ∈ {*0..1*}
    and *tu*: *t* < *u*
    and *uv*: *u* < *v*
    and *ptUV*: *p t* ∈ *U* ∩ *V*
    and *puUV*: *p u* ∈ *U* ∩ *V*
    and *pvUV*: *p v* ∈ *U* ∩ *V*

and *seg-tvU*: *subpathin t v p ‘ {0..1} ⊆ U*
and *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordLeft* (*loop-class U x0* (*segment-loop p t u*))
(*WordLeft* (*loop-class U x0* (*segment-loop p u v*)) *rest*))
(*WordLeft* (*loop-class U x0* (*segment-loop p t v*)) *rest*)
⟨*proof*⟩

lemma *segment-word-split-right-equiv*:
assumes *p-loop*: *p ∈ loop-space W x0*
and *t01*: *t ∈ {0..1}*
and *u01*: *u ∈ {0..1}*
and *v01*: *v ∈ {0..1}*
and *tu*: *t < u*
and *uv*: *u < v*
and *ptUV*: *p t ∈ U ∩ V*
and *puUV*: *p u ∈ U ∩ V*
and *pvUV*: *p v ∈ U ∩ V*
and *seg-tvV*: *subpathin t v p ‘ {0..1} ⊆ V*
and *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*WordRight* (*loop-class V x0* (*segment-loop p t u*))
(*WordRight* (*loop-class V x0* (*segment-loop p u v*)) *rest*))
(*WordRight* (*loop-class V x0* (*segment-loop p t v*)) *rest*)
⟨*proof*⟩

lemma *segment-word-switch*:
assumes *p-path*: *path p*
and *p-imgW*: *path-image p ⊆ W*
and *t01*: *t ∈ {0..1}*
and *u01*: *u ∈ {0..1}*
and *tu*: *t < u*
and *ptUV*: *p t ∈ U ∩ V*
and *puUV*: *p u ∈ U ∩ V*
and *segUV*: *subpathin t u p ‘ {0..1} ⊆ U ∩ V*
and *rest-in*: *fpw-in-space G1 G2 rest*
and *bc*: *b ≠ c*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*if b then WordLeft* (*loop-class U x0* (*segment-loop p t u*)) *rest*
else WordRight (*loop-class V x0* (*segment-loop p t u*)) *rest*)
(*if c then WordLeft* (*loop-class U x0* (*segment-loop p t u*)) *rest*
else WordRight (*loop-class V x0* (*segment-loop p t u*)) *rest*)
⟨*proof*⟩

lemma *svk-partition-split-head*:
assumes *part*: *svk-partition p* (*t # v # us*) (*b # bs*)
and *tu*: *t < u*
and *uv*: *u < v*
and *puUV*: *p u ∈ U ∩ V*

shows *svk-partition p* (*t # u # v # us*) (*b # b # bs*)
⟨*proof*⟩

lemma *svk-partition-same-start-equiv*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
and *part1*: *svk-partition p* (*t # ts*) *bs*
and *part2*: *svk-partition p* (*t # us*) *cs*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word p* (*t # ts*) *bs*)
(*partition-word p* (*t # us*) *cs*)
⟨*proof*⟩

lemma *valid-partition-same-loop-partition-word-equiv*:
assumes *p-loop*: *p* ∈ *loop-space W x0*
and *part1*: *valid-partition p ts bs*
and *part2*: *valid-partition p us cs*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word p ts bs*) (*partition-word p us cs*)
⟨*proof*⟩

lemma *strip-neighbourhood*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *a01*: *a* ∈ {*0..1*}
and *b01*: *b* ∈ {*0..1*}
and *y0-01*: *y0* ∈ {*0..1*}
and *ab*: *a* ≤ *b*
and *rowS*: *h* ‘ ({*a..b*} × {*y0*}) ⊆ *S*
and *openS*: *open S*
shows ∃ *N*. *openin* (*top-of-set* {*0..1*}) *N* ∧ *y0* ∈ *N* ∧ *h* ‘ ({*a..b*} × *N*) ⊆ *S*
⟨*proof*⟩

lemma *svk-partition-local-neighbourhood*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *y0-01*: *y0* ∈ {*0..1*}
and *part*: *svk-partition* (λ*x*. *h* (*x*, *y0*)) *ts bs*
shows ∃ *N*. *openin* (*top-of-set* {*0..1*}) *N* ∧ *y0* ∈ *N* ∧
(∀ *x*∈*set ts*. *h* ‘ ({*x*} × *N*) ⊆ *U* ∩ *V*) ∧
(∀ *y z*. *y* ≤ *z* ⟶ {*y..z*} ⊆ *N* ⟶ *rectangle-partition h y z ts bs*)
⟨*proof*⟩

lemma *rectangle-partition-svk-partition-row*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *part*: *rectangle-partition h c d ts bs*
and *edgeUV*: ⋀*x*. *x* ∈ *set ts* ⟹ *h* ‘ ({*x*} × {*c..d*}) ⊆ *U* ∩ *V*
and *y-in*: *y* ∈ {*c..d*}

shows *svk-partition* ($\lambda x.\ h\ (x,\ y)$) *ts bs*
⟨*proof*⟩

lemma *rectangle-partition-valid-partition-row*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *part*: *rectangle-partition h c d ts bs*
and *edgeUV*: $\bigwedge x.\ x \in set\ ts \Longrightarrow h\ `\ (\{x\} \times \{c..d\}) \subseteq U \cap V$
and *y-in*: $y \in \{c..d\}$
and *ts-ne*: $ts \neq []$
and *hd0*: $hd\ ts = 0$
shows *valid-partition* ($\lambda x.\ h\ (x,\ y)$) *ts bs*
⟨*proof*⟩

lemma *homotopy-row-in-loop-space*:
fixes $h :: (real \times real) \Rightarrow 'a$
assumes *h-cont*: *continuous-map* (*top-of-set* ($\{0..1\} \times \{0..1\}$)) (*top-of-set W*) *h*
and *y01*: $y \in \{0..1\}$
and *start*: $h\ (0,\ y) = x0$
and *finish*: $h\ (1,\ y) = x0$
shows $(\lambda x.\ h\ (x,\ y)) \in$ *loop-space W x0*
⟨*proof*⟩

lemma *bridge-word-one-equiv*:
assumes *rest-in*: *fpw-in-space G1 G2 rest*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*bridge-word b* (*fundamental-group-one* ($U \cap V$) *x0*) *rest*) *rest*
⟨*proof*⟩

lemma *partition-word-with-tail-respects*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *svk-partition p ts bs*
and *rel*: *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 r s*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word-with-tail p ts bs r*)
(*partition-word-with-tail p ts bs s*)
⟨*proof*⟩

lemma *valid-partition-tail-bridge-one-equiv*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *part*: *valid-partition p* ($t \#\ u \#\ ts$) ($b \#\ bs$)
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word-with-tail p* ($t \#\ u \#\ ts$) ($b \#\ bs$)
(*bridge-word* (*last* ($b \#\ bs$)) (*fundamental-group-one* ($U \cap V$) *x0*) *WordNil*))
(*partition-word p* ($t \#\ u \#\ ts$) ($b \#\ bs$))
⟨*proof*⟩

lemma *bridge-loop-constant-one*:
fixes $h :: (real \times real) \Rightarrow 'a$

assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *t01*: $t \in \{0..1\}$
and *c01*: $c \in \{0..1\}$
and *d01*: $d \in \{0..1\}$
and *cd*: $c \leq d$
and *const*: $\forall y \in \{c..d\}$. *h* (*t*, *y*) = *x0*
shows *loop-class* (*U* ∩ *V*) *x0* (*bridge-loop h t c d*) = *fundamental-group-one* (*U* ∩ *V*) *x0*
⟨*proof*⟩

lemma *rectangle-partition-partition-word-equiv*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *c01*: $c \in \{0..1\}$
and *d01*: $d \in \{0..1\}$
and *cd*: $c \leq d$
and *part*: *rectangle-partition h c d ts bs*
and *ts-ne*: *ts* ≠ []
and *hd0*: *hd ts* = *0*
and *edgeUV*: ⋀*x*. *x* ∈ *set ts* ⟹ *h* ‘ ({*x*} × {*c..d*}) ⊆ *U* ∩ *V*
and *end0*: $\forall y \in \{c..d\}$. *h* (*0*, *y*) = *x0*
and *end1*: $\forall y \in \{c..d\}$. *h* (*1*, *y*) = *x0*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word* (λ*x*. *h* (*x*, *c*)) *ts bs*)
(*partition-word* (λ*x*. *h* (*x*, *d*)) *ts bs*)
⟨*proof*⟩

lemma *valid-partition-nearby-partition-word-equiv*:
fixes *h* :: (*real* × *real*) ⇒ *'a*
assumes *h-cont*: *continuous-map* (*top-of-set* ({*0..1*} × {*0..1*})) (*top-of-set W*) *h*
and *end0*: $\forall y \in \{0..1\}$. *h* (*0*, *y*) = *x0*
and *end1*: $\forall y \in \{0..1\}$. *h* (*1*, *y*) = *x0*
and *y0-01*: $y0 \in \{0..1\}$
and *part0*: *valid-partition* (λ*x*. *h* (*x*, *y0*)) *ts bs*
shows $\exists e{>}0.\ \forall z \in \{0..1\}$. *dist z y0* < *e* ⟶
valid-partition (λ*x*. *h* (*x*, *z*)) *ts bs* ∧
carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
(*partition-word* (λ*x*. *h* (*x*, *y0*)) *ts bs*)
(*partition-word* (λ*x*. *h* (*x*, *z*)) *ts bs*)
⟨*proof*⟩

definition *some-valid-partition* :: (*real* ⇒ *'a*) ⇒ *real list* × *bool list* **where**
some-valid-partition p =
(*SOME tb*. *case tb of* (*ts*, *bs*) ⇒ *valid-partition p ts bs*)

lemma *some-valid-partition-spec*:

assumes *p-loop*: $p \in$ *loop-space W x0*
shows *valid-partition p*
(*fst* (*some-valid-partition p*))
(*snd* (*some-valid-partition p*))
⟨*proof*⟩

15.3 Homotopy invariance of the partition encoding

The central technical step is that different valid partitions of homotopic loops produce equivalent words in the carrier-side amalgamated free product. The proof uses a rectangular decomposition of a homotopy and keeps the word encoding stable while moving from one boundary loop to the other.

lemma *valid-partition-homotopic-partition-word-equiv*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *q-loop*: $q \in$ *loop-space W x0*
and *pq*: *homotopic-paths W p q*
and *p-part*: *valid-partition p ts bs*
and *q-part*: *valid-partition q us cs*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word p ts bs*) (*partition-word q us cs*)
⟨*proof*⟩

lemma *valid-partition-loop-class-partition-word-equiv*:
assumes *p-loop*: $p \in$ *loop-space W x0*
and *q-loop*: $q \in$ *loop-space W x0*
and *eq*: *loop-class W x0 p* $=$ *loop-class W x0 q*
and *p-part*: *valid-partition p ts bs*
and *q-part*: *valid-partition q us cs*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(*partition-word p ts bs*) (*partition-word q us cs*)
⟨*proof*⟩

lemma *svk-decode-word-loop*:
assumes *w-in*: *fpw-in-space G1 G2 w*
shows *svk-decode w* $=$ *loop-class W x0* (*word-loop w*)
⟨*proof*⟩

15.4 Encoding, decoding, and the final bijection

With existence, refinement invariance, and homotopy invariance in place, the encode/decode pair can now be defined directly on loop classes. The remaining lemmas verify the round-trip laws required by the abstract interface and turn them into the classical Seifert–van Kampen bijection.

definition *svk-encode* ::
(*real* $\Rightarrow$ *'a*) *set* $\Rightarrow$ ((*real* $\Rightarrow$ *'a*) *set*, (*real* $\Rightarrow$ *'a*) *set*) *free-product-word*
where
svk-encode A $=$
(*let p* $=$ *some-loop W x0 A*;

tb = some-valid-partition p
in case tb of (ts, bs) ⇒ partition-word p ts bs)

lemma *svk-encode-in-space*:
assumes *A-in*: *A ∈ fundamental-group-space W x0*
shows *fpw-in-space G1 G2 (svk-encode A)*
⟨*proof*⟩

lemma *svk-decode-encode*:
assumes *A-in*: *A ∈ fundamental-group-space W x0*
shows *svk-decode (svk-encode A) = A*
⟨*proof*⟩

lemma *svk-encode-decode*:
assumes *w-in*: *fpw-in-space G1 G2 w*
shows *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*
(svk-encode (svk-decode w)) w
⟨*proof*⟩

lemma *svk-decode-surjective*:
assumes *A-in*: *A ∈ fundamental-group-space W x0*
shows *∃ w. fpw-in-space G1 G2 w ∧ svk-decode w = A*
⟨*proof*⟩

theorem *svk-decode-image*:
svk-decode ‘ {w. fpw-in-space G1 G2 w} = fundamental-group-space W x0
⟨*proof*⟩

definition *classical-svk-quotient-map* ::
(real ⇒ 'a) set ⇒
(((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word) set
where
classical-svk-quotient-map A =
carrier-full-amalg-class G1 G2 H i1 i2 mult1 one1 mult2 one2 (svk-encode A)

lemma *classical-svk-quotient-map-in-space*:
assumes *A-in*: *A ∈ fundamental-group-space W x0*
shows *classical-svk-quotient-map A ∈*
carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2
⟨*proof*⟩

lemma *classical-svk-quotient-map-injective*:
assumes *AB*: *classical-svk-quotient-map A = classical-svk-quotient-map B*
and *A-in*: *A ∈ fundamental-group-space W x0*
and *B-in*: *B ∈ fundamental-group-space W x0*
shows *A = B*
⟨*proof*⟩

lemma *classical-svk-quotient-map-surjective*:
assumes *Q-in*:
Q ∈ *carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2*
shows ∃ *A* ∈ *fundamental-group-space W x0*. *classical-svk-quotient-map A* = *Q*
⟨*proof*⟩

At this point the encoding and decoding maps satisfy the abstract round-trip laws from the interface theory. The final theorem therefore presents the classical Seifert–van Kampen statement as a bijection between the fundamental group of $U \cup V$ at *x0* and the carrier-based amalgamated free product assembled from U, V, and $U \cap V$.

theorem *classical-seifert-van-kampen-bij-betw*:
bij-betw classical-svk-quotient-map (*fundamental-group-space W x0*)
(*carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2*)
⟨*proof*⟩

end

end
theory *Binary-Sum-Topology*
imports *HOL−Analysis.Abstract-Topology*
begin

16 Binary coproduct topology

Topological pushouts are constructed by first equipping the disjoint sum with its coproduct topology and only then quotienting by the glue relation. This short theory records that coproduct topology together with the basic continuity lemmas used by the later pushout arguments.

definition *binary-sum-topology* :: *'a topology* => *'b topology* => (*'a* + *'b*) *topology*
where
binary-sum-topology X Y =
topology (λ*U*.
U ⊆ *Inl* ‘ *topspace X* ∪ *Inr* ‘ *topspace Y* ∧
openin X {*x*. *Inl x* ∈ *U*} ∧
openin Y {*y*. *Inr y* ∈ *U*})

lemma *is-binary-sum-topology*:
istopology (λ*U*.
U ⊆ *Inl* ‘ *topspace X* ∪ *Inr* ‘ *topspace Y* ∧
openin X {*x*. *Inl x* ∈ *U*} ∧
openin Y {*y*. *Inr y* ∈ *U*})
⟨*proof*⟩

lemma *openin-binary-sum-topology*:

openin (*binary-sum-topology X Y*) *U* ⟷
U ⊆ *Inl* ‘ *topspace X* ∪ *Inr* ‘ *topspace Y* ∧
openin X {*x*. *Inl x* ∈ *U*} ∧
openin Y {*y*. *Inr y* ∈ *U*}
⟨*proof*⟩

lemma *topspace-binary-sum-topology* [*simp*]:
topspace (*binary-sum-topology X Y*) = *Inl* ‘ *topspace X* ∪ *Inr* ‘ *topspace Y*
⟨*proof*⟩

lemma *subset-topspace-binary-sum-inl*:
(*Inl* :: *'a* ⇒ *'a* + *'b*) ‘ *topspace X* ⊆ *topspace* (*binary-sum-topology X Y*)
⟨*proof*⟩

lemma *subset-topspace-binary-sum-inr*:
(*Inr* :: *'b* ⇒ *'a* + *'b*) ‘ *topspace Y* ⊆ *topspace* (*binary-sum-topology X Y*)
⟨*proof*⟩

lemma *continuous-map-binary-sum-inl*:
continuous-map X (*binary-sum-topology X Y*) (*Inl* :: *'a* ⇒ *'a* + *'b*)
⟨*proof*⟩

lemma *continuous-map-binary-sum-inr*:
continuous-map Y (*binary-sum-topology X Y*) (*Inr* :: *'b* ⇒ *'a* + *'b*)
⟨*proof*⟩

lemma *open-map-binary-sum-inl*:
open-map X (*binary-sum-topology X Y*) (*Inl* :: *'a* ⇒ *'a* + *'b*)
⟨*proof*⟩

lemma *open-map-binary-sum-inr*:
open-map Y (*binary-sum-topology X Y*) (*Inr* :: *'b* ⇒ *'a* + *'b*)
⟨*proof*⟩

lemma *continuous-map-from-binary-sum-topology*:
assumes *f*: *continuous-map X Z f*
and *g*: *continuous-map Y Z g*
shows *continuous-map* (*binary-sum-topology X Y*) *Z* (*case-sum f g*)
⟨*proof*⟩

end
theory *Topological-Pushout-Scaffold*
imports *Binary-Sum-Topology Pushout-Scaffold*
begin

17 Topological pushouts

The quotient-level pushout relation from *Pushout-Scaffold* becomes a genuine topological pushout here. The theory defines the quotient topology on the glued disjoint sum and proves the universal maps from the two summands into that topological pushout.

definition *pushout-topspace* ::
  'a topology => 'b topology => ('c => 'a) => ('c => 'b) => ('a + 'b) set set
where
  pushout-topspace X Y f g = pushout-class f g ‘ topspace (binary-sum-topology X Y)

definition *pushout-topology* ::
  'a topology => 'b topology => ('c => 'a) => ('c => 'b) => (('a + 'b) set) topology
where
  pushout-topology X Y f g =
    topology (λU.
      U ⊆ pushout-topspace X Y f g ∧
      openin (binary-sum-topology X Y)
        {z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ U})

lemma *pushout-topspace-alt*:
  pushout-topspace X Y f g = pushout-inl f g ‘ topspace X ∪ pushout-inr f g ‘ topspace Y
  ⟨*proof*⟩

lemma *pushout-topspace-subset-pushout-space*:
  pushout-topspace X Y f g ⊆ pushout-space f g
  ⟨*proof*⟩

lemma *is-pushout-topology*:
  istopology (λU.
      U ⊆ pushout-topspace X Y f g ∧
      openin (binary-sum-topology X Y)
        {z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ U})
⟨*proof*⟩

lemma *openin-pushout-topology*:
  openin (pushout-topology X Y f g) U ⟷
    U ⊆ pushout-topspace X Y f g ∧
    openin (binary-sum-topology X Y)
      {z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ U}
⟨*proof*⟩

lemma *topspace-pushout-topology* [*simp*]:
  topspace (pushout-topology X Y f g) = pushout-topspace X Y f g
⟨*proof*⟩

lemma *quotient-map-pushout-class*:
quotient-map (*binary-sum-topology X Y*) (*pushout-topology X Y f g*) (*pushout-class f g*)
⟨*proof*⟩

lemma *continuous-map-pushout-class*:
continuous-map (*binary-sum-topology X Y*) (*pushout-topology X Y f g*) (*pushout-class f g*)
⟨*proof*⟩

lemma *pushout-inl-in-topspace*:
assumes *a* ∈ *topspace X*
shows *pushout-inl f g a* ∈ *pushout-topspace X Y f g*
⟨*proof*⟩

lemma *pushout-inr-in-topspace*:
assumes *b* ∈ *topspace Y*
shows *pushout-inr f g b* ∈ *pushout-topspace X Y f g*
⟨*proof*⟩

lemma *continuous-map-pushout-inl* [*continuous-intros*]:
continuous-map X (*pushout-topology X Y f g*) (*pushout-inl f g*)
⟨*proof*⟩

lemma *continuous-map-pushout-inr* [*continuous-intros*]:
continuous-map Y (*pushout-topology X Y f g*) (*pushout-inr f g*)
⟨*proof*⟩

lemma *pushout-rec-pushout-class*:
assumes *compat*: *pushout-case-compatible f g left right*
and *x-in*: *x* ∈ *topspace* (*binary-sum-topology X Y*)
shows *pushout-rec f g left right* (*pushout-class f g x*) = *case-sum left right x*
⟨*proof*⟩

lemma *continuous-map-pushout-rec*:
assumes *compat*: *pushout-case-compatible f g left right*
and *left-cont*: *continuous-map X Z left*
and *right-cont*: *continuous-map Y Z right*
shows *continuous-map* (*pushout-topology X Y f g*) *Z* (*pushout-rec f g left right*)
⟨*proof*⟩

lemma *pushout-rec-universal*:
assumes *compat*: *pushout-case-compatible f g left right*
and *left-cont*: *continuous-map X Z left*
and *right-cont*: *continuous-map Y Z right*
shows *continuous-map* (*pushout-topology X Y f g*) *Z* (*pushout-rec f g left right*)
and ⋀*a*. *pushout-rec f g left right* (*pushout-inl f g a*) = *left a*
and ⋀*b*. *pushout-rec f g left right* (*pushout-inr f g b*) = *right b*
⟨*proof*⟩

end
theory *Topological-Seifert-Van-Kampen*
imports
Carrier-Amalgamated-Free-Product-Eval
Explicit-Fundamental-Group-Scaffold
Topological-Pushout-Scaffold
Seifert-Van-Kampen-Scaffold
begin

18 Concrete decode data for topological pushouts

Once the pushout topology and the carrier-side amalgamation evaluator are available, the remaining task is to package a concrete decoding map back into the fundamental group of the pushout. This theory shows how such decode data yields the abstract carrier-level Seifert–van Kampen bijection in the topological pushout setting.

lemma *loopin-image-compose* [*simp*]:
loopin-image g (*loopin-image f p*) = *loopin-image* (*g* ∘ *f*) *p*
⟨*proof*⟩

lemma *fundamental-groupin-map-eq*:
assumes *A-in*: *A* ∈ *fundamental-groupin-space X x*
and *f-cont*: *continuous-map X Y f*
and *g-cont*: *continuous-map X Y g*
and *fx*: *f x* = *y*
and *gx*: *g x* = *y*
and *fg*: ⋀*u*. *f u* = *g u*
shows *fundamental-groupin-map X x Y y f A* = *fundamental-groupin-map X x Y y g A*
⟨*proof*⟩

lemma *fundamental-groupin-map-compose*:
assumes *A-in*: *A* ∈ *fundamental-groupin-space X x*
and *f-cont*: *continuous-map X Y f*
and *g-cont*: *continuous-map Y Z g*
and *fx*: *f x* = *y*
and *gy*: *g y* = *z*
shows *fundamental-groupin-map Y y Z z g* (*fundamental-groupin-map X x Y y f A*) =
fundamental-groupin-map X x Z z (*g* ∘ *f*) *A*
⟨*proof*⟩

locale *topological-svk-setup* =
fixes *X* :: *'a topology*
and *Y* :: *'b topology*
and *Z* :: *'c topology*
and *f* :: *'c* ⇒ *'a*

and $g :: 'c \Rightarrow 'b$
and $z0 :: 'c$
assumes *z0-in*: $z0 \in$ *topspace* Z
and *f-cont*: *continuous-map* Z X f
and *g-cont*: *continuous-map* Z Y g
begin

abbreviation $x0$ **where** $x0 \equiv f\ z0$
abbreviation $y0$ **where** $y0 \equiv g\ z0$
abbreviation P **where** $P \equiv$ *pushout-topology* X Y f g
abbreviation $p0$ **where** $p0 \equiv$ *pushout-inl* f g $x0$

abbreviation $G1$ **where** $G1 \equiv$ *fundamental-groupin-space* X $x0$
abbreviation $G2$ **where** $G2 \equiv$ *fundamental-groupin-space* Y $y0$
abbreviation H **where** $H \equiv$ *fundamental-groupin-space* Z $z0$
abbreviation *mult1* **where** *mult1* $\equiv$ *fundamental-groupin-mult* X $x0$
abbreviation *one1* **where** *one1* $\equiv$ *fundamental-groupin-one* X $x0$
abbreviation *mult2* **where** *mult2* $\equiv$ *fundamental-groupin-mult* Y $y0$
abbreviation *one2* **where** *one2* $\equiv$ *fundamental-groupin-one* Y $y0$
abbreviation *multP* **where** *multP* $\equiv$ *fundamental-groupin-mult* P $p0$
abbreviation *oneP* **where** *oneP* $\equiv$ *fundamental-groupin-one* P $p0$

abbreviation $i1$ **where** $i1 \equiv$ *fundamental-groupin-map* Z $z0$ X $x0$ f
abbreviation $i2$ **where** $i2 \equiv$ *fundamental-groupin-map* Z $z0$ Y $y0$ g
abbreviation $j1$ **where** $j1 \equiv$ *fundamental-groupin-map* X $x0$ P $p0$ (*pushout-inl* f g)
abbreviation $j2$ **where** $j2 \equiv$ *fundamental-groupin-map* Y $y0$ P $p0$ (*pushout-inr* f g)

lemma *x0-in* [*simp*]: $x0 \in$ *topspace* X
⟨*proof*⟩

lemma *y0-in* [*simp*]: $y0 \in$ *topspace* Y
⟨*proof*⟩

lemma *pushout-basepoint-eq* [*simp*]:
pushout-inr f g $y0 = p0$
⟨*proof*⟩

lemma *pushout-basepoint-in* [*simp*]:
$p0 \in$ *topspace* P
⟨*proof*⟩

lemma *pushout-fundamental-group-maps-agree*:
assumes *h-in*: $h \in H$
shows $j1\ (i1\ h) = j2\ (i2\ h)$
⟨*proof*⟩

lemma *i1-in-G1*:

assumes $h \in H$
shows *i1* $h \in$ *G1*
⟨*proof*⟩

lemma *i2-in-G2*:
assumes $h \in H$
shows *i2* $h \in$ *G2*
⟨*proof*⟩

lemma *decode-locale*:
carrier-full-amalg-word-eval
G1 mult1 one1 (*fundamental-groupin-inv X x0*)
G2 mult2 one2 (*fundamental-groupin-inv Y y0*)
H i1 i2
(*fundamental-groupin-space P p0*) *multP oneP* (*fundamental-groupin-inv P p0*)
j1 j2
⟨*proof*⟩

interpretation *decode*:
carrier-full-amalg-word-eval
G1 mult1 one1 fundamental-groupin-inv X x0
G2 mult2 one2 fundamental-groupin-inv Y y0
H i1 i2
fundamental-groupin-space P p0 multP oneP fundamental-groupin-inv P p0
j1 j2
⟨*proof*⟩

abbreviation *svk-word-eval* **where** *svk-word-eval* ≡ *decode.carrier-full-amalg-eval*
abbreviation *svk-decode* **where** *svk-decode* ≡ *decode.carrier-full-amalg-decode*

lemma *svk-decode-in-space*:
svk-decode w ∈ *fundamental-groupin-space P p0*
⟨*proof*⟩

lemma *svk-decode-respects*:
assumes *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*
shows *svk-decode u* = *svk-decode v*
⟨*proof*⟩

lemma *svk-decode-eq-eval*:
assumes *fpw-in-space G1 G2 w*
shows *svk-decode w* = *svk-word-eval w*
⟨*proof*⟩

end

locale *topological-svk-open-cover* =
topological-svk-setup X Y Z f g z0
for *X* :: *'a topology*

```
    and Y :: 'b topology
    and Z :: 'c topology
    and f :: 'c ⇒ 'a
    and g :: 'c ⇒ 'b
    and z0 :: 'c +
  assumes left-open: openin P (pushout-inl f g ‘ topspace X)
    and right-open: openin P (pushout-inr f g ‘ topspace Y)
begin

lemma loopin-subdivision-by-summands:
  assumes p-loop: p ∈ loopin-space P p0
  shows ∃n::nat. 0 < n ∧
      (∀i<n.
        subpathin (real i / real n) (real (Suc i) / real n) p ‘ {0..1}
          ⊆ pushout-inl f g ‘ topspace X ∨
        subpathin (real i / real n) (real (Suc i) / real n) p ‘ {0..1}
          ⊆ pushout-inr f g ‘ topspace Y)
⟨proof⟩

end

locale topological-svk =
  topological-svk-setup X Y Z f g z0
  for X :: 'a topology
    and Y :: 'b topology
    and Z :: 'c topology
    and f :: 'c ⇒ 'a
    and g :: 'c ⇒ 'b
    and z0 :: 'c +
  fixes encode
  assumes encode-in-space: fpw-in-space G1 G2 (encode A)
    and decode-encode: svk-decode (encode A) = A
    and encode-decode:
        carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (encode
(svk-decode w)) w
begin

lemma svk-locale:
  carrier-svk-encode-decode-interface
    G1 G2 H i1 i2 mult1 one1 mult2 one2 encode svk-decode
⟨proof⟩

interpretation svk: carrier-svk-encode-decode-interface
  G1 G2 H i1 i2 mult1 one1 mult2 one2 encode svk-decode
  ⟨proof⟩
```

The theorem below isolates the topological conclusion once an encoding map has been supplied and the usual round-trip laws have been verified. The classical open-union theorem later instantiates this interface by constructing

that encoding from loop partitions subordinate to U and V.

theorem *topological-seifert-van-kampen-bij-betw*:
bij-betw svk.carrier-svk-quotient-map UNIV
(*carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2 one2*)
⟨*proof*⟩

end

end
theory *Seifert-Van-Kampen*
imports
Classical-Seifert-Van-Kampen
Topological-Seifert-Van-Kampen
begin

19 Top-level entry point

This session formalizes a quotient-oriented version of the classical Seifert–van Kampen theorem in Isabelle/HOL. It develops reusable infrastructure for pushout glue relations, free-product words, carrier-based amalgamated free products, and explicit path/homotopy quotients. The unconditional classical open-union theorem is proved in *Classical-Seifert-Van-Kampen*.

Auxiliary theories package the abstract encode/decode interface and the pushout-level constructions used internally by the proof. The headline theorem exported by this entry is the classical result for open unions.

end

References

[1] R. Brown. *Topology and Groupoids*. BookSurge LLC, Charleston, SC, 3 edition, 2006.

[2] A. Hatcher. *Algebraic Topology*. Cambridge University Press, Cambridge, 2002.

[3] H. Seifert. Konstruktion dreidimensionaler geschlossener räume. *Berichte über die Verhandlungen der Sächsischen Akademie der Wissenschaften zu Leipzig, Mathematisch-Physische Klasse*, 83:26–66, 1931.

[4] E. R. van Kampen. On the connection between the fundamental groups of some related spaces. *American Journal of Mathematics*, 55:261–267, 1933. JSTOR stable URL: https://www.jstor.org/stable/51000091.