

# The Classical Seifert–van Kampen Theorem

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

May 28, 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

<b>1 Overview</b>	<b>1</b>
<b>2 Development structure</b>	<b>1</b>
<b>3 Sources</b>	<b>2</b>
<b>4 Equivalence classes as quotient infrastructure</b>	<b>3</b>
<b>5 Free-product words</b>	<b>4</b>
<b>6 Amalgamation quotients of free-product words</b>	<b>8</b>
<b>7 Carrier-based amalgamated free products</b>	<b>10</b>
<b>8 Groups on carriers</b>	<b>17</b>
<b>9 Evaluating carrier amalgamated free-product words</b>	<b>21</b>
<b>10 Explicit-topology paths and homotopies</b>	<b>30</b>
<b>11 Fundamental-group quotients</b>	<b>56</b>

<b>12</b>	<b>Explicit-topology fundamental-group quotients</b>	<b>80</b>
<b>13</b>	<b>Pushout-style quotients of disjoint sums</b>	<b>96</b>
<b>14</b>	<b>Conditional Seifert–van Kampen interface</b>	<b>100</b>
<b>15</b>	<b>Classical Seifert–van Kampen for open unions</b>	<b>104</b>
15.1	Carrier-side setup . . . . .	104
15.2	Partitions and encoded loop words . . . . .	118
15.3	Homotopy invariance of the partition encoding . . . . .	273
15.4	Encoding, decoding, and the final bijection . . . . .	282
<b>16</b>	<b>Binary coproduct topology</b>	<b>286</b>
<b>17</b>	<b>Topological pushouts</b>	<b>291</b>
<b>18</b>	<b>Concrete decode data for topological pushouts</b>	<b>295</b>
<b>19</b>	<b>Top-level entry point</b>	<b>302</b>
	<b>theory</b> <i>Equivalence-Quotients</i>	
	<b>imports</b> <i>Main</i>	
	<b>begin</b>	

## 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* :: ( $'a \Rightarrow 'a \Rightarrow \text{bool}$ )  $\Rightarrow 'a \Rightarrow 'a \text{ set}$  **where**  
*equiv-class*  $R$   $x = \{y. R y x\}$

**definition** *quotient-space* :: ( $'a \Rightarrow 'a \Rightarrow \text{bool}$ )  $\Rightarrow 'a \text{ set set}$  **where**  
*quotient-space*  $R = \text{equiv-class } R \text{ ` UNIV}$

**lemma** *quotient-spaceI*:  
*equiv-class*  $R$   $x \in \text{quotient-space } R$   
**unfolding** *quotient-space-def* **by** *blast*

**lemma** *equiv-class-iff* [*simp*]:  
 $y \in \text{equiv-class } R x \longleftrightarrow R y x$   
**unfolding** *equiv-class-def* **by** *simp*

**locale** *equivalence-relation* =  
**fixes**  $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$   
**assumes** *refl* [*intro!*, *simp*]:  $R x x$

```

    and sym:  $R\ x\ y \implies R\ y\ x$ 
    and transitive:  $R\ x\ y \implies R\ y\ z \implies R\ x\ z$ 
begin

lemma equiv-class-eqI:
  assumes  $R\ x\ y$ 
  shows  $\text{equiv-class}\ R\ x = \text{equiv-class}\ R\ y$ 
  using assms sym transitive
  unfolding equiv-class-def
  by blast

lemma equiv-class-eq-iff:
   $\text{equiv-class}\ R\ x = \text{equiv-class}\ R\ y \iff R\ x\ y$ 
proof
  assume  $h : \text{equiv-class}\ R\ x = \text{equiv-class}\ R\ y$ 
  have  $x \in \text{equiv-class}\ R\ x$ 
    unfolding equiv-class-def by simp
  with  $h$  show  $R\ x\ y$ 
    unfolding equiv-class-def by simp
next
  assume  $R\ x\ y$ 
  then show  $\text{equiv-class}\ R\ x = \text{equiv-class}\ R\ y$ 
    by (rule equiv-class-eqI)
qed

lemma quotient-spaceE:
  assumes  $Q \in \text{quotient-space}\ R$ 
  obtains  $x$  where  $Q = \text{equiv-class}\ R\ x$ 
  using assms unfolding quotient-space-def by blast

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*  
**by** (*induction w*) *simp-all*

**lemma** *fpw-concat-assoc* [*simp*]:  
*fpw-concat (fpw-concat u v) w = fpw-concat u (fpw-concat v w)*  
**by** (*induction u*) *simp-all*

**lemma** *fpw-reverse-concat* [*simp*]:  
*fpw-reverse (fpw-concat u v) = fpw-concat (fpw-reverse v) (fpw-reverse u)*  
**by** (*induction u*) *simp-all*

**lemma** *fpw-reverse-reverse* [*simp*]:  
*fpw-reverse (fpw-reverse w) = w*  
**by** (*induction w*) *simp-all*

**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)
by (induction u) simp-all

```

```

lemma fpw-inverse-inverse [simp]:
  fpw-inverse (fpw-inverse w) = w
by (induction w) (simp-all add: fpw-inverse-concat)

```

```

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)*  
**using** *assms*  
**proof** (*induction rule: fpw-reduction.induct*)  
**case** (*refl w*)  
**then show** *?case* **by** *simp*  
**next**  
**case** (*sym u v*)  
**then show** *?case* **by** (*meson fpw-reduction.sym*)  
**next**  
**case** (*trans u v w*)  
**then show** *?case* **by** (*meson fpw-reduction.trans*)  
**next**  
**case** (*step u v*)  
**then show** *?case*  
**by** (*meson fpw-reduction.step fpw-reduction-step.context-left*)  
**qed**

**lemma** *fpw-reduction-right-context*:  
**assumes** *fpw-reduction u v*  
**shows** *fpw-reduction (WordRight b u) (WordRight b v)*  
**using** *assms*  
**proof** (*induction rule: fpw-reduction.induct*)  
**case** (*refl w*)  
**then show** *?case* **by** *simp*  
**next**  
**case** (*sym u v*)  
**then show** *?case* **by** (*meson fpw-reduction.sym*)  
**next**  
**case** (*trans u v w*)  
**then show** *?case* **by** (*meson fpw-reduction.trans*)  
**next**  
**case** (*step u v*)  
**then show** *?case*  
**by** (*meson fpw-reduction.step fpw-reduction-step.context-right*)  
**qed**

**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**

**show** *amalgam-equiv* *i1* *i2* *x* *x* **for** *x*

**by** (*rule* *amalgam-equiv.refl*)

**next**

**show** *amalgam-equiv* *i1* *i2* *x* *y* ==> *amalgam-equiv* *i1* *i2* *y* *x* **for** *x* *y*

**by** (*rule* *amalgam-equiv.sym*)

**next**

**show** *amalgam-equiv* *i1* *i2* *x* *y* ==> *amalgam-equiv* *i1* *i2* *y* *z* ==> *amalgam-equiv* *i1* *i2* *x* *z* **for** *x* *y* *z*

**by** (*rule* *amalgam-equiv.trans*)

**qed**

**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* ::

$(h \Rightarrow a) \Rightarrow (h \Rightarrow b) \Rightarrow$   
 $((a, b) \text{ free-product-word}) \text{ set set}$

**where**

$\text{amalgamated-free-product-space } i1 \ i2 = \text{quotient-space } (\text{amalgam-equiv } i1 \ i2)$

**lemma** *amalgam-class-eq-iff*:

$\text{amalgam-class } i1 \ i2 \ u = \text{amalgam-class } i1 \ i2 \ v \iff \text{amalgam-equiv } i1 \ i2 \ u \ v$

**unfolding** *amalgam-class-def*

**by** (*rule amalgam-equiv-equiv-equiv-class-eq-iff*)

**inductive** *full-amalg-equiv* ::

$(h \Rightarrow a::\text{group-add}) \Rightarrow (h \Rightarrow b::\text{group-add}) \Rightarrow$   
 $((a, b) \text{ free-product-word}) \Rightarrow ((a, b) \text{ free-product-word}) \Rightarrow \text{bool}$   
**for**  $i1 :: h \Rightarrow a$  **and**  $i2 :: h \Rightarrow b$

**where**

*refl* [*intro!*, *simp*]:  $\text{full-amalg-equiv } i1 \ i2 \ w \ w$

| *sym*:  $\text{full-amalg-equiv } i1 \ i2 \ u \ v \implies \text{full-amalg-equiv } i1 \ i2 \ v \ u$

| *trans*:

$\text{full-amalg-equiv } i1 \ i2 \ u \ v \implies \text{full-amalg-equiv } i1 \ i2 \ v \ w \implies \text{full-amalg-equiv } i1 \ i2 \ u \ w$

| *of-amalg*:

$\text{amalgam-equiv } i1 \ i2 \ u \ v \implies \text{full-amalg-equiv } i1 \ i2 \ u \ v$

| *of-reduction*:

$\text{fpw-reduction } u \ v \implies \text{full-amalg-equiv } i1 \ i2 \ u \ v$

**interpretation** *full-amalg-equiv-equiv*: *equivalence-relation full-amalg-equiv i1 i2*

**proof**

**show**  $\text{full-amalg-equiv } i1 \ i2 \ x \ x$  **for**  $x$

**by** (*rule full-amalg-equiv.refl*)

**next**

**show**  $\text{full-amalg-equiv } i1 \ i2 \ x \ y \implies \text{full-amalg-equiv } i1 \ i2 \ y \ x$  **for**  $x \ y$

**by** (*rule full-amalg-equiv.sym*)

**next**

**show**

$\text{full-amalg-equiv } i1 \ i2 \ x \ y \implies \text{full-amalg-equiv } i1 \ i2 \ y \ z \implies \text{full-amalg-equiv } i1 \ i2 \ x \ z$

**for**  $x \ y \ z$

**by** (*rule full-amalg-equiv.trans*)

**qed**

**definition** *full-amalg-class* ::

$(h \Rightarrow a::\text{group-add}) \Rightarrow (h \Rightarrow b::\text{group-add}) \Rightarrow$   
 $((a, b) \text{ free-product-word}) \Rightarrow ((a, b) \text{ free-product-word}) \text{ set}$

**where**

$\text{full-amalg-class } i1 \ i2 \ w = \text{equiv-class } (\text{full-amalg-equiv } i1 \ i2) \ w$

**definition** *full-amalgamated-free-product-space* ::

```

('h => 'a::group-add) => ('h => 'b::group-add) =>
  (('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
unfolding full-amalg-class-def
by (rule full-amalg-equiv-equiv-equiv-class-eq-iff)

lemma full-amalg-class-in-space [intro]:
  full-amalg-class i1 i2 w  $\in$  full-amalgamated-free-product-space i1 i2
unfolding full-amalg-class-def full-amalgamated-free-product-space-def quotient-space-def
by blast

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  $\in$  G  $\wedge$  fpw-in-space G H rest)
| fpw-in-space G H (WordRight b rest) = (b  $\in$  H  $\wedge$  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  $\in$  carrier-fpw-space G H  $\longleftrightarrow$  fpw-in-space G H w
unfolding carrier-fpw-space-def by simp

```

**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**

**show** carrier-amalgam-equiv K i1 i2 x x **for** x  
**by** (rule carrier-amalgam-equiv.refl)

**next**

**show** carrier-amalgam-equiv K i1 i2 x y ⇒ carrier-amalgam-equiv K i1 i2 y x

**for** x y

**by** (rule carrier-amalgam-equiv.sym)

**next**

**show**

carrier-amalgam-equiv K i1 i2 x y ⇒  
 carrier-amalgam-equiv K i1 i2 y z ⇒  
 carrier-amalgam-equiv K i1 i2 x z **for** x y z  
**by** (rule carrier-amalgam-equiv.trans)

qed

**definition** *carrier-amalgam-class* ::

$'h \text{ set} \Rightarrow ('h \Rightarrow 'a) \Rightarrow ('h \Rightarrow 'b) \Rightarrow$   
 $('a, 'b) \text{ free-product-word} \Rightarrow (('a, 'b) \text{ free-product-word}) \text{ set}$

**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*  $K \ i1 \ i2 \ u = \text{carrier-amalgam-class } K \ i1 \ i2 \ v$   
 $\longleftrightarrow \text{carrier-amalgam-equiv } K \ i1 \ i2 \ u \ v$

**unfolding** *carrier-amalgam-class-def*

**by** (*rule carrier-amalgam-equiv-equiv-equiv-class-eq-iff*)

**inductive** *carrier-fpw-reduction-step* ::

$'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow$   
 $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow$   
 $('a, 'b) \text{ free-product-word} \Rightarrow ('a, 'b) \text{ free-product-word} \Rightarrow \text{bool}$   
**for**  $G1 :: 'a \text{ set}$  **and**  $G2 :: 'b \text{ set}$   
**and**  $\text{mult1} :: 'a \Rightarrow 'a \Rightarrow 'a$  **and**  $\text{one1} :: 'a$   
**and**  $\text{mult2} :: 'b \Rightarrow 'b \Rightarrow 'b$  **and**  $\text{one2} :: 'b$

**where**

*combine-left*:

$\llbracket a \in G1; b \in G1; \text{mult1 } a \ b \in G1; \text{fpw-in-space } G1 \ G2 \ \text{rest} \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordLeft } a \ (\text{WordLeft } b \ \text{rest}))$   
 $(\text{WordLeft } (\text{mult1 } a \ b) \ \text{rest})$

| *combine-right*:

$\llbracket a \in G2; b \in G2; \text{mult2 } a \ b \in G2; \text{fpw-in-space } G1 \ G2 \ \text{rest} \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight } a \ (\text{WordRight } b \ \text{rest}))$   
 $(\text{WordRight } (\text{mult2 } a \ b) \ \text{rest})$

| *remove-left-one*:

$\llbracket \text{one1} \in G1; \text{fpw-in-space } G1 \ G2 \ \text{rest} \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordLeft } \text{one1} \ \text{rest}) \ \text{rest}$

| *remove-right-one*:

$\llbracket \text{one2} \in G2; \text{fpw-in-space } G1 \ G2 \ \text{rest} \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight } \text{one2} \ \text{rest}) \ \text{rest}$

| *context-left*:

$\llbracket a \in G1; \text{carrier-fpw-reduction-step } G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2} \ u \ v \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordLeft } a \ u) \ (\text{WordLeft } a \ v)$

| *context-right*:

$\llbracket b \in G2; \text{carrier-fpw-reduction-step } G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2} \ u \ v \rrbracket \Longrightarrow$   
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight } b \ u) \ (\text{WordRight } b \ v)$

```

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)
  using assms
proof (induction rule: carrier-fpw-reduction.induct)
  case (refl w)
  then show ?case by simp
next
  case (sym u v)
  then show ?case by (meson carrier-fpw-reduction.sym)
next
  case (trans u v w)
  then show ?case by (meson carrier-fpw-reduction.trans)
next
  case (step u v)
  then show ?case
  by (meson carrier-fpw-reduction.step carrier-fpw-reduction-step.context-left)
qed

lemma carrier-fpw-reduction-right-context:
  assumes carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 u v
  and b ∈ G2
  shows carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2 (WordRight b u)
  (WordRight b v)
  using assms
proof (induction rule: carrier-fpw-reduction.induct)

```

```

    case (refl w)
    then show ?case by simp
next
    case (sym u v)
    then show ?case by (meson carrier-fpw-reduction.sym)
next
    case (trans u v w)
    then show ?case by (meson carrier-fpw-reduction.trans)
next
    case (step u v)
    then show ?case
      by (meson carrier-fpw-reduction.step carrier-fpw-reduction-step.context-right)
qed

```

```

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
  using step
proof (induction rule: carrier-fpw-reduction-step.induct)
  case (combine-left a b rest)
  then show fpw-in-space G1 G2 (WordLeft a (WordLeft b rest))
    by simp
  show fpw-in-space G1 G2 (WordLeft (mult1 a b) rest)
    using combine-left by simp
next
  case (combine-right a b rest)
  then show fpw-in-space G1 G2 (WordRight a (WordRight b rest))
    by simp
  show fpw-in-space G1 G2 (WordRight (mult2 a b) rest)
    using combine-right by simp
next
  case (remove-left-one rest)
  then show fpw-in-space G1 G2 (WordLeft one1 rest)
    by simp
  show fpw-in-space G1 G2 rest
    using remove-left-one by simp
next
  case (remove-right-one rest)
  then show fpw-in-space G1 G2 (WordRight one2 rest)
    by simp
  show fpw-in-space G1 G2 rest
    using remove-right-one by simp
next
  case (context-left a u v)
  then show fpw-in-space G1 G2 (WordLeft a u)
    by simp
  show fpw-in-space G1 G2 (WordLeft a v)
    using context-left by simp
next

```

```

case (context-right b u v)
then show fpw-in-space G1 G2 (WordRight b u)
  by simp
show fpw-in-space G1 G2 (WordRight b v)
  using context-right by simp
qed

```

```

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  $\longleftrightarrow$  fpw-in-space G1 G2 v
using carrier-fpw-reduction-step-preserves-space[OF step] by blast

```

```

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  $\longleftrightarrow$  fpw-in-space G1 G2 v
using rel
proof (induction rule: carrier-fpw-reduction.induct)
  case (refl w)
    then show ?case by simp
  next
    case (sym u v)
      then show ?case by simp
  next
    case (trans u v w)
      then show ?case by blast
  next
    case (step u v)
      then show ?case
        by (rule carrier-fpw-reduction-step-preserves-space-iff)
qed

```

```

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  $\implies$ 
  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  $\implies$ 
  carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 v w  $\implies$ 
  carrier-full-amalg-equiv G1 G2 K i1 i2 mult1 one1 mult2 one2 u w

```

| of-amalg:  
   *carrier-amalgam-equiv*  $K\ i1\ i2\ u\ v \implies$   
   *carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ u\ v$   
 | of-reduction:  
   *carrier-fpw-reduction*  $G1\ G2\ mult1\ one1\ mult2\ one2\ u\ v \implies$   
   *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**

**show** *carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ x\ x$  **for**  $x$   
**by** (rule *carrier-full-amalg-equiv.refl*)

**next**

**show**

*carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ x\ y \implies$   
*carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ y\ x$  **for**  $x\ y$   
**by** (rule *carrier-full-amalg-equiv.sym*)

**next**

**show**

*carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ x\ y \implies$   
*carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ y\ z \implies$   
*carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ x\ z$  **for**  $x\ y\ z$   
**by** (rule *carrier-full-amalg-equiv.trans*)

**qed**

**definition** *carrier-full-amalg-class* ::

$'a\ set \implies 'b\ set \implies 'h\ set \implies ('h \implies 'a) \implies ('h \implies 'b) \implies$   
 $('a \implies 'a \implies 'a) \implies 'a \implies ('b \implies 'b \implies 'b) \implies 'b \implies$   
 $('a, 'b)\ free-product-word \implies (('a, 'b)\ free-product-word)\ set$

**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* ::

$'a\ set \implies 'b\ set \implies 'h\ set \implies ('h \implies 'a) \implies ('h \implies 'b) \implies$   
 $('a \implies 'a \implies 'a) \implies 'a \implies ('b \implies 'b \implies 'b) \implies 'b \implies$   
 $(('a, 'b)\ free-product-word)\ set\ set$

**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*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ v$   
 $\iff$  *carrier-full-amalg-equiv*  $G1\ G2\ K\ i1\ i2\ mult1\ one1\ mult2\ one2\ u\ v$   
**unfolding** *carrier-full-amalg-class-def*

by (rule carrier-full-amalg-equiv-equiv.equiv-class-eq-iff)

**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*  $\in$

*carrier-full-amalgamated-free-product-space G1 G2 K i1 i2 mult1 one1 mult2 one2*

**using** *assms*

**unfolding** *carrier-full-amalgamated-free-product-space-def*

**by** *simp*

**definition** *carrier-full-amalg-one* ::

*'a set*  $\Rightarrow$  *'b set*  $\Rightarrow$  *'h set*  $\Rightarrow$  (*'h*  $\Rightarrow$  *'a*)  $\Rightarrow$  (*'h*  $\Rightarrow$  *'b*)  $\Rightarrow$

(*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*)  $\Rightarrow$  *'a*  $\Rightarrow$  (*'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  *'b*  $\Rightarrow$

((*'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*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*

**and** *one* :: *'a*

**and** *inv* :: *'a*  $\Rightarrow$  *'a*

**assumes** *one-closed* [intro, simp]: *one*  $\in$  *G*

**and** *mult-closed* [intro]:  $\llbracket x \in G; y \in G \rrbracket \Longrightarrow$  *mult* *x y*  $\in$  *G*

**and** *inv-closed* [intro]: *x*  $\in$  *G*  $\Longrightarrow$  *inv* *x*  $\in$  *G*

**and** *mult-assoc*:  $\llbracket x \in G; y \in G; z \in G \rrbracket \Longrightarrow$  *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 -
  have  $mult\ (inv\ x)\ (mult\ x\ y) = mult\ (inv\ x)\ (mult\ x\ z)$ 
    using  $eq$  by  $simp$ 
  moreover have  $mult\ (inv\ x)\ (mult\ x\ y) = y$ 
  proof -
    have  $mult\ (inv\ x)\ (mult\ x\ y) = mult\ (mult\ (inv\ x)\ x)\ y$ 
      using  $x\ y\ inv-closed[OF\ x]$  by  $(simp\ add: mult-assoc[symmetric])$ 
    also have  $\dots = y$ 
      using  $x\ y$  by  $(simp\ add: mult-inv-left\ mult-one-left)$ 
    finally show  $?thesis$  .
  qed
  moreover have  $mult\ (inv\ x)\ (mult\ x\ z) = z$ 
  proof -
    have  $mult\ (inv\ x)\ (mult\ x\ z) = mult\ (mult\ (inv\ x)\ x)\ z$ 
      using  $x\ z\ inv-closed[OF\ x]$  by  $(simp\ add: mult-assoc[symmetric])$ 
    also have  $\dots = z$ 
      using  $x\ z$  by  $(simp\ add: mult-inv-left\ mult-one-left)$ 
    finally show  $?thesis$  .
  qed
  ultimately show  $?thesis$ 
    by  $simp$ 
qed

```

**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 -
  have  $mult\ (mult\ y\ x)\ (inv\ x) = mult\ (mult\ z\ x)\ (inv\ x)$ 
    using  $eq$  by  $simp$ 
  moreover have  $mult\ (mult\ y\ x)\ (inv\ x) = y$ 
  proof -
    have  $mult\ (mult\ y\ x)\ (inv\ x) = mult\ y\ (mult\ x\ (inv\ x))$ 
      using  $x\ y\ inv-closed[OF\ x]$  by  $(simp\ add: mult-assoc)$ 
    also have  $\dots = y$ 
      using  $x\ y$  by  $(simp\ add: mult-inv-right\ mult-one-right)$ 
    finally show  $?thesis$  .
  qed
  moreover have  $mult\ (mult\ z\ x)\ (inv\ x) = z$ 
  proof -
    have  $mult\ (mult\ z\ x)\ (inv\ x) = mult\ z\ (mult\ x\ (inv\ x))$ 
      using  $x\ z\ inv-closed[OF\ x]$  by  $(simp\ add: mult-assoc)$ 

```

```

    also have ... = z
      using x z by (simp add: mult-inv-right mult-one-right)
    finally show ?thesis .
  qed
  ultimately show ?thesis
    by simp
qed

```

```

lemma left-inverse-unique:
  assumes x: x ∈ G
    and y: y ∈ G
    and eq: mult y x = one
  shows y = inv x
proof (rule right-cancel[OF x y inv-closed[OF x]])
  show mult y x = mult (inv x) x
    using eq x by (simp add: mult-inv-left)
qed

```

```

lemma right-inverse-unique:
  assumes x: x ∈ G
    and y: y ∈ G
    and eq: mult x y = one
  shows y = inv x
proof (rule left-cancel[OF x y inv-closed[OF x]])
  show mult x y = mult x (inv x)
    using eq x by (simp add: mult-inv-right)
qed

```

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 ∈ G ⟹ h x ∈ H
    and map-mult: [x ∈ G; y ∈ G] ⟹ h (mult x y) = mult' (h x) (h y)
begin

```

```

lemma map-one:
  h one = one'
proof -
  have h-one-closed: h one ∈ H
    using G.one-closed by (rule map-closed)
  have h-one-idem: h one = mult' (h one) (h one)
  proof -
    have h one = h (mult one one)
      using G.one-closed by (simp add: G.mult-one-left)

```

```

    also have ... = mult' (h one) (h one)
      using G.one-closed G.one-closed by (rule map-mult)
    finally show ?thesis .
  qed
  have mult' one' (h one) = mult' (h one) (h one)
    using h-one-closed h-one-idem by (simp add: H.mult-one-left)
  then have one' = h one
    by (rule H.right-cancel[OF h-one-closed H.one-closed h-one-closed])
  then show ?thesis
    by simp
  qed

lemma map-inv:
  assumes x: x ∈ G
  shows h (inv x) = inv' (h x)
  proof -
    have hx: h x ∈ H
      using x by (rule map-closed)
    have hinvx: h (inv x) ∈ H
      using x G.inv-closed[OF x] by (auto intro: map-closed)
    have eq-left: mult' (h x) (h (inv x)) = one'
    proof -
      have h (mult x (inv x)) = mult' (h x) (h (inv x))
        using x G.inv-closed[OF x] by (rule map-mult)
      then have mult' (h x) (h (inv x)) = h (mult x (inv x))
        by simp
      also have ... = h one
        using x by (simp add: G.mult-inv-right)
      also have ... = one'
        by (rule map-one)
      finally show ?thesis .
    qed
    have eq: mult' (h x) (h (inv x)) = mult' (h x) (inv' (h x))
      using eq-left hx by (simp add: H.mult-inv-right)
    show ?thesis
      by (rule H.left-cancel[OF hx hinvx H.inv-closed[OF hx] eq])
  qed

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
using assms
proof (induction w)
  case WordNil

```

```

then show ?case
  by simp
next
case (WordLeft a rest)
then have a-in:  $a \in G1$  and rest-in: fpw-in-space  $G1\ G2\ rest$ 
  by auto
have eval-rest-in: carrier-full-amalg-eval  $rest \in K$ 
  by (rule WordLeft.IH[OF rest-in])
then show ?case
  using a-in eval-rest-in by (simp add: Cod.mult-closed J1.map-closed)
next
case (WordRight b rest)
then have b-in:  $b \in G2$  and rest-in: fpw-in-space  $G1\ G2\ rest$ 
  by auto
have eval-rest-in: carrier-full-amalg-eval  $rest \in K$ 
  by (rule WordRight.IH[OF rest-in])
then show ?case
  using b-in eval-rest-in by (simp add: Cod.mult-closed J2.map-closed)
qed

```

```

lemma carrier-amalgam-step-preserves-space-iff:
  assumes step: carrier-amalgam-step  $H\ i1\ i2\ u\ v$ 
  shows fpw-in-space  $G1\ G2\ u \longleftrightarrow fpw-in-space\ G1\ G2\ v$ 
  using step
proof cases
  case (identify h rest)
  then show ?thesis
    using i1-closed[OF  $\langle h \in H \rangle$ ] i2-closed[OF  $\langle h \in H \rangle$ ] by simp
qed

```

```

lemma carrier-amalgam-equiv-preserves-space-iff:
  assumes rel: carrier-amalgam-equiv  $H\ i1\ i2\ u\ v$ 
  shows fpw-in-space  $G1\ G2\ u \longleftrightarrow fpw-in-space\ G1\ G2\ v$ 
  using rel
proof (induction rule: carrier-amalgam-equiv.induct)
  case (refl w)
  then show ?case
    by simp
next
  case (sym u v)
  then show ?case
    by simp
next
  case (trans u v w)
  then show ?case
    by blast
next
  case (step u v)
  then show ?case

```

```

    by (rule carrier-amalgam-step-preserves-space-iff)
next
  case (left-context u v a)
  then show ?case
    by simp
next
  case (right-context u v b)
  then show ?case
    by simp
qed

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
  using step
proof cases
  case (identify h rest)
  then show ?thesis
    using agree[OF ⟨h ∈ H⟩]
    by (simp add: J1.map-closed J2.map-closed)
qed

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
  using rel
proof (induction rule: carrier-amalgam-equiv.induct)
  case (refl w)
  then show ?case
    by simp
next
  case (sym u v)
  then show ?case
    by simp
next
  case (trans u v w)
  then show ?case
    by simp
next
  case (step u v)
  then show ?case
    by (rule carrier-amalgam-step-preserves-eval)
next
  case (left-context u v a)
  then show ?case
    by auto
next
  case (right-context u v b)
  then show ?case

```

by *auto*  
**qed**

**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*  
**using** *step*

**proof** (*induction rule: carrier-fpw-reduction-step.induct*)  
**case** (*combine-left a b rest*)  
**have** *a-in: a ∈ G1 and b-in: b ∈ G1 and rest-in: fpw-in-space G1 G2 rest*  
**using** *combine-left by auto*  
**have** *eval-rest-in: carrier-full-amalg-eval rest ∈ K*  
**by** (*rule carrier-full-amalg-eval-in-carrier[OF rest-in]*)  
**have** *ja-in: j1 a ∈ K*  
**by** (*rule J1.map-closed[OF a-in]*)  
**have** *jb-in: j1 b ∈ K*  
**by** (*rule J1.map-closed[OF b-in]*)  
**show** *?case*  
**using** *a-in b-in eval-rest-in ja-in jb-in*  
**by** (*simp add: J1.map-mult Cod.mult-assoc[symmetric]*)

**next**  
**case** (*combine-right a b rest*)  
**have** *a-in: a ∈ G2 and b-in: b ∈ G2 and rest-in: fpw-in-space G1 G2 rest*  
**using** *combine-right by auto*  
**have** *eval-rest-in: carrier-full-amalg-eval rest ∈ K*  
**by** (*rule carrier-full-amalg-eval-in-carrier[OF rest-in]*)  
**have** *ja-in: j2 a ∈ K*  
**by** (*rule J2.map-closed[OF a-in]*)  
**have** *jb-in: j2 b ∈ K*  
**by** (*rule J2.map-closed[OF b-in]*)  
**show** *?case*  
**using** *a-in b-in eval-rest-in ja-in jb-in*  
**by** (*simp add: J2.map-mult Cod.mult-assoc[symmetric]*)

**next**  
**case** (*remove-left-one rest*)  
**have** *rest-in: fpw-in-space G1 G2 rest*  
**using** *remove-left-one by auto*  
**have** *eval-rest-in: carrier-full-amalg-eval rest ∈ K*  
**by** (*rule carrier-full-amalg-eval-in-carrier[OF rest-in]*)  
**show** *?case*  
**using** *eval-rest-in*  
**by** (*simp add: J1.map-one Cod.mult-one-left*)

**next**  
**case** (*remove-right-one rest*)  
**have** *rest-in: fpw-in-space G1 G2 rest*  
**using** *remove-right-one by auto*  
**have** *eval-rest-in: carrier-full-amalg-eval rest ∈ K*  
**by** (*rule carrier-full-amalg-eval-in-carrier[OF rest-in]*)  
**show** *?case*

```

    using eval-rest-in
    by (simp add: J2.map-one Cod.mult-one-left)
next
case (context-left a u v)
then show ?case
  by simp
next
case (context-right b u v)
then show ?case
  by simp
qed

```

```

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  $\longleftrightarrow$  fpw-in-space G1 G2 v
proof -
  show ?thesis
    by (rule carrier-fpw-reduction-space-iff[OF rel])
qed

```

```

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
  using rel
proof (induction rule: carrier-fpw-reduction.induct)
  case (refl w)
  then show ?case
    by simp
next
  case (sym u v)
  then show ?case
    by simp
next
  case (trans u v w)
  then show ?case
    by simp
next
  case (step u v)
  then show ?case
    by (rule carrier-fpw-reduction-step-preserves-eval)
qed

```

```

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  $\longleftrightarrow$  fpw-in-space G1 G2 v
  using rel

```

```

proof (induction rule: carrier-full-amalg-equiv.induct)
  case (refl w)
  then show ?case
    by simp
next
  case (sym u v)
  then show ?case
    by simp
next
  case (trans u v w)
  then show ?case
    by blast
next
  case (of-amalg u v)
  then show ?case
    by (rule carrier-amalgam-equiv-preserves-space-iff)
next
  case (of-reduction u v)
  then show ?case
    by (rule carrier-fpw-reduction-preserves-space-iff)
qed

```

**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
  using rel
proof (induction rule: carrier-full-amalg-equiv.induct)
  case (refl w)
  then show ?case
    by simp
next
  case (sym u v)
  then show ?case
    by simp
next
  case (trans u v w)
  then show ?case
    by simp
next
  case (of-amalg u v)
  then show ?case
    by (rule carrier-amalgam-equiv-preserves-eval)
next
  case (of-reduction u v)
  then show ?case
    by (rule carrier-fpw-reduction-preserves-eval)
qed

```

**definition** *carrier-full-amalg-has-good-rep* :: ('a, 'b) free-product-word => bool

**where**

*carrier-full-amalg-has-good-rep*  $w \longleftrightarrow$   
( $\exists v.$  *carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2\ w\ v$   
 $\wedge$  *fpw-in-space*  $G1\ G2\ v$ )

**definition** *carrier-full-amalg-some-good-rep* ::

( $'a, 'b$ ) *free-product-word*  $\Rightarrow$  ( $'a, 'b$ ) *free-product-word*

**where**

*carrier-full-amalg-some-good-rep*  $w =$   
(*SOME*  $v.$  *carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2\ w\ v$   
 $\wedge$  *fpw-in-space*  $G1\ G2\ v$ )

**definition** *carrier-full-amalg-decode* :: ( $'a, 'b$ ) *free-product-word*  $\Rightarrow$   $'k$  **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*  $w$

**using** *assms* **unfolding** *carrier-full-amalg-has-good-rep-def* **by** *auto*

**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** –

**from** *assms* **obtain**  $v$  **where**

*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2\ w\ v$

**and** *fpw-in-space*  $G1\ G2\ v$

**unfolding** *carrier-full-amalg-has-good-rep-def* **by** *blast*

**then have** *ex*:

$\exists v.$  *carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2\ w\ v$   
 $\wedge$  *fpw-in-space*  $G1\ G2\ v$

**by** *blast*

**then have**

*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$

$w$  (*carrier-full-amalg-some-good-rep*  $w$ )

$\wedge$  *fpw-in-space*  $G1\ G2$  (*carrier-full-amalg-some-good-rep*  $w$ )

**unfolding** *carrier-full-amalg-some-good-rep-def*

**by** (*rule someI-ex*)

**then show**

*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*  $w$ )

**by** *auto*

**qed**

**lemma** *carrier-full-amalg-has-good-rep-respects:*  
**assumes** *uw: carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v*  
**shows** *carrier-full-amalg-has-good-rep u  $\longleftrightarrow$  carrier-full-amalg-has-good-rep v*  
**proof**  
**assume** *carrier-full-amalg-has-good-rep u*  
**then obtain** *w where*  
   *uw: carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u w*  
   **and** *w-in: fpw-in-space G1 G2 w*  
   **unfolding** *carrier-full-amalg-has-good-rep-def* **by** *blast*  
**have** *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 v w*  
   **using** *uw ww*  
   **by** (*meson carrier-full-amalg-equiv.sym carrier-full-amalg-equiv.trans*)  
**then show** *carrier-full-amalg-has-good-rep v*  
   **using** *w-in unfolding carrier-full-amalg-has-good-rep-def* **by** *blast*  
**next**  
**assume** *carrier-full-amalg-has-good-rep v*  
**then obtain** *w where*  
   *vw: carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 v w*  
   **and** *w-in: fpw-in-space G1 G2 w*  
   **unfolding** *carrier-full-amalg-has-good-rep-def* **by** *blast*  
**have** *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u w*  
   **using** *uw vw*  
   **by** (*meson carrier-full-amalg-equiv.trans*)  
**then show** *carrier-full-amalg-has-good-rep u*  
   **using** *w-in unfolding carrier-full-amalg-has-good-rep-def* **by** *blast*  
**qed**

**lemma** *carrier-full-amalg-decode-in-carrier:*  
   *carrier-full-amalg-decode w  $\in$  K*  
**proof** (*cases carrier-full-amalg-has-good-rep w*)  
   **case** *True*  
   **then have** *good-rep:*  
      *fpw-in-space G1 G2 (carrier-full-amalg-some-good-rep w)*  
      **by** (*rule carrier-full-amalg-some-good-rep*)  
   **show** *?thesis*  
      **unfolding** *carrier-full-amalg-decode-def*  
      **using** *True carrier-full-amalg-eval-in-carrier[OF good-rep]* **by** *simp*  
**next**  
   **case** *False*  
   **then show** *?thesis*  
      **unfolding** *carrier-full-amalg-decode-def* **by** *simp*  
**qed**

**lemma** *carrier-full-amalg-decode-respects:*  
**assumes** *uw: 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** (*cases carrier-full-amalg-has-good-rep u*)  
   **case** *False*

```

then have  $\neg$  carrier-full-amalg-has-good-rep v
  using carrier-full-amalg-has-good-rep-respects[OF uv] by blast
then show ?thesis
  unfolding carrier-full-amalg-decode-def using False by simp
next
case True
then have v-has: carrier-full-amalg-has-good-rep v
  using carrier-full-amalg-has-good-rep-respects[OF uv] by blast
have u-rep-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  u (carrier-full-amalg-some-good-rep u)
  using True by (rule carrier-full-amalg-some-good-rep)+
have v-rep-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  v (carrier-full-amalg-some-good-rep v)
  using v-has by (rule carrier-full-amalg-some-good-rep)+
have rep-eq:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (carrier-full-amalg-some-good-rep u)
  (carrier-full-amalg-some-good-rep v)
  using u-rep-rel v-rep-rel uv
by (meson carrier-full-amalg-equiv.sym carrier-full-amalg-equiv.trans)
have eval-eq:
  carrier-full-amalg-eval (carrier-full-amalg-some-good-rep u) =
  carrier-full-amalg-eval (carrier-full-amalg-some-good-rep v)
  by (rule carrier-full-amalg-equiv-preserves-eval[OF rep-eq])
show ?thesis
  unfolding carrier-full-amalg-decode-def
  using True v-has eval-eq by simp

```

**qed**

**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** –

**have** *has: carrier-full-amalg-has-good-rep w*

**by** (*rule carrier-full-amalg-has-good-repI[OF w-in]*)

**have** *rep-rel:*

*carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*  
*w (carrier-full-amalg-some-good-rep w)*

**using** *has* **by** (*rule carrier-full-amalg-some-good-rep*)+

**have** *eval-eq:*

*carrier-full-amalg-eval w =*  
*carrier-full-amalg-eval (carrier-full-amalg-some-good-rep w)*  
**by** (*rule carrier-full-amalg-equiv-preserves-eval[OF rep-rel]*)

**show** *?thesis*

**unfolding** *carrier-full-amalg-decode-def*

**using** *has eval-eq* **by** *simp*

**qed**

**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* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{loopin-space } X \ x = \{p. \text{pathin } X \ p \wedge p \ 0 = x \wedge p \ 1 = x\}$

**definition** *homotopic-pathsin* ::

$'a \text{ topology} \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow \text{bool}$

**where**

$\text{homotopic-pathsin } X \ p \ q \equiv$

$\text{homotopic-with } (\lambda r. r \ 0 = p \ 0 \wedge r \ 1 = p \ 1) (\text{top-of-set } \{0..1\}) \ X \ p \ q$

**definition** *homotopic-loopsin* ::

$'a \text{ topology} \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow \text{bool}$

**where**

$\text{homotopic-loopsin } X \ p \ q \equiv$

$\text{homotopic-with } (\lambda r. r \ 1 = r \ 0) (\text{top-of-set } \{0..1\}) \ X \ p \ q$

**definition** *reversepathin* ::  $(\text{real} \Rightarrow 'a) \Rightarrow \text{real} \Rightarrow 'a$

**where**

$\text{reversepathin } p = (\lambda t. p \ (1 - t))$

**definition** *joinpathin* ::  $(\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow \text{real} \Rightarrow 'a$

**where**

$\text{joinpathin } p \ q = (\lambda t. \text{if } t \leq 1 / 2 \text{ then } p \ (2 * t) \text{ else } q \ (2 * t - 1))$

**definition** *subpathin* ::  $\text{real} \Rightarrow \text{real} \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow \text{real} \Rightarrow 'a$

**where**

$\text{subpathin } u \ v \ p = (\lambda t. p \ ((v - u) * t + u))$

**lemma** *reversepathin-eq-reversepath* [*simp*]:

$\text{reversepathin } p = \text{reversepath } p$

**unfolding** *reversepathin-def* *reversepath-def* **by** *simp*

**lemma** *reversepathin-reversepathin* [*simp*]:

$reversepathin (reversepathin p) = p$   
**unfolding** *reversepathin-def* **by** (*rule ext*) *simp*

**lemma** *joinpathin-eq-joinpaths* [*simp*]:  
 $joinpathin p q = p +++ q$   
**unfolding** *joinpathin-def joinpaths-def* **by** *simp*

**lemma** *subpathin-image*:  
 $subpathin u v p \text{ ‘ } \{0..1\} = p \text{ ‘ closed-segment } u v$   
**by** (*simp add: subpathin-def image-image closed-segment-real-eq algebra-simps*)

**lemma** *subpathin-image-eq*:  
**assumes**  $u \leq v$   
**shows**  $subpathin u v p \text{ ‘ } \{0..1\} = p \text{ ‘ } \{u..v\}$   
**using** *assms* **by** (*simp add: subpathin-image closed-segment-eq-real-ivl*)

**lemma** *subpathin-0-1* [*simp*]:  
 $subpathin 0 1 p = p$   
**unfolding** *subpathin-def* **by** (*rule ext*) *simp*

**lemma** *joinpathin-subpathin-middle* [*simp*]:  
 $joinpathin (subpathin 0 (1 / 2) p) (subpathin (1 / 2) 1 p) = p$   
**unfolding** *joinpathin-def subpathin-def*  
**by** (*rule ext*) (*auto simp: field-simps*)

**lemma** *continuous-map-top01-id*:  
 $continuous-map (top-of-set \{0..1\}) euclideanreal id$   
**by** (*rule continuous-map-into-fulltopology[OF continuous-map-id]*)

**lemma** *loopin-spaceI*:  
**assumes**  $pathin X p$   
**and**  $p 0 = x$   
**and**  $p 1 = x$   
**shows**  $p \in loopin-space X x$   
**using** *assms* **unfolding** *loopin-space-def* **by** *blast*

**lemma** *loopin-spaceE*:  
**assumes**  $p \in loopin-space X x$   
**obtains**  $pathin X p$   $p 0 = x$   $p 1 = x$   
**using** *assms* **unfolding** *loopin-space-def* **by** *blast*

**lemma** *constant-loopin-in-space*:  
**assumes**  $x \in topspace X$   
**shows**  $(\lambda-. x) \in loopin-space X x$   
**using** *assms* **unfolding** *loopin-space-def* **by** *auto*

**lemma** *pathin-image-subset-topospace*:  
**assumes**  $pathin X p$   
**shows**  $p \text{ ‘ } \{0..1\} \subseteq topspace X$

```

proof –
  have  $p \in \{0..1\} \rightarrow \text{topspace } X$ 
    using assms by (rule path-image-subset-topspace)
  then show ?thesis
    by auto
qed

lemma pathin-reversepathin:
  assumes pathin  $X$   $p$ 
  shows pathin  $X$  (reversepathin  $p$ )
proof –
  have rev-cont: continuous-map (top-of-set {0..1}) (top-of-set {0..1}) ( $\lambda t::\text{real}.$ 
 $1 - t$ )
    by (intro continuous-map-into-subtopology continuous-intros) auto
  have continuous-map (top-of-set {0..1})  $X$  ( $p \circ (\lambda t::\text{real}.$   $1 - t)$ )
    using rev-cont assms unfolding pathin-def by (rule continuous-map-compose)
  then show ?thesis
    unfolding pathin-def reversepathin-def by (simp add: o-def)
qed

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 –
  let ?T01 = subtopology euclideanreal {0..1}
  have  $p$ -cont: continuous-map ?T01  $X$   $p$ 
    using  $p$  unfolding pathin-def .
  have  $q$ -cont: continuous-map ?T01  $X$   $q$ 
    using  $q$  unfolding pathin-def .
  have left-scale-eu:
    continuous-map (subtopology ?T01  $\{x \in \text{topspace } ?T01. 2 * x \leq 1\}$ ) euclidean-
    real ( $\lambda x::\text{real}.$   $2 * x$ )
  proof –
    have continuous-map ?T01 euclideanreal ( $\lambda x::\text{real}.$   $2 * x$ )
      by (simp add: continuous-map-iff-continuous continuous-intros)
    then show ?thesis
      by (rule continuous-map-from-subtopology)
  qed
  have left-scale-range:
    ( $\lambda x::\text{real}.$   $2 * x$ )  $\in$  topspace (subtopology ?T01  $\{x \in \text{topspace } ?T01. 2 * x \leq$ 
 $1\}$ )  $\rightarrow$  {0..1}
  proof
    fix  $x :: \text{real}$ 
    assume  $x$ -in:  $x \in \text{topspace}$  (subtopology ?T01  $\{x \in \text{topspace } ?T01. 2 * x \leq$ 
 $1\}$ )
    then have  $x01$ :  $x \in \{0..1\}$  and  $x$ -half:  $2 * x \leq 1$ 
      by auto

```

```

from  $x01$  have  $x\text{-ge}0$ :  $0 \leq x$ 
  by auto
have  $lower$ :  $0 \leq 2 * x$ 
  using  $x\text{-ge}0$  by linarith
show  $2 * x \in \{0..1\}$ 
  using  $lower$   $x\text{-half}$  by simp
qed
have  $left\text{-scale}\text{-cont}$ :
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 2 * x ≤ 1}) ?T01
( $\lambda x::real. 2 * x$ )
  by (rule continuous-map-into-subtopology[OF left-scale-eu left-scale-range])
have  $left\text{-cont}\text{-comp}$ :
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 2 * x ≤ 1}) X
( $p \circ (\lambda x::real. 2 * x)$ )
  by (rule continuous-map-compose[OF left-scale-cont p-cont])
have  $left\text{-eq}$ :
  ( $p \circ (\lambda x::real. 2 * x)$ ) = ( $\lambda x::real. p (2 * x)$ )
  by (auto simp: fun-eq-iff o-def)
from  $left\text{-cont}\text{-comp}$  have  $left\text{-cont}$ :
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 2 * x ≤ 1}) X ( $\lambda x::real.$ 
 $p (2 * x)$ )
  by (simp only: left-eq)
have  $right\text{-shift}\text{-eu}$ :
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 * x}) euclidean-
real ( $\lambda x::real. 2 * x - 1$ )
proof –
  have continuous-map ?T01 euclideanreal ( $\lambda x::real. 2 * x - 1$ )
  by (simp add: continuous-map-iff-continuous continuous-intros)
  then show ?thesis
  by (rule continuous-map-from-subtopology)
qed
have  $right\text{-shift}\text{-range}$ :
  ( $\lambda x::real. 2 * x - 1$ )  $\in$  topspace (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤
 $2 * x$ })  $\rightarrow$   $\{0..1\}$ 
proof
  fix  $x :: real$ 
  assume  $x\text{-in}$ :  $x \in$  topspace (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 *
 $x$ })
  then have  $x01$ :  $x \in \{0..1\}$  and  $x\text{-half}$ :  $1 \leq 2 * x$ 
  by auto
from  $x01$  have  $x\text{-le}1$ :  $x \leq 1$ 
  by auto
have  $lower$ :  $0 \leq 2 * x - 1$ 
  using  $x\text{-half}$  by linarith
have  $upper$ :  $2 * x - 1 \leq 1$ 
  using  $x\text{-le}1$  by linarith
show  $2 * x - 1 \in \{0..1\}$ 
  using  $lower$   $upper$  by simp
qed

```

```

have right-shift-cont:
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 * x}) ?T01
  (λx::real. 2 * x - 1)
  by (rule continuous-map-into-subtopology[OF right-shift-eu right-shift-range])
have right-cont-comp:
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 * x}) X
  (q ∘ (λx::real. 2 * x - 1))
  by (rule continuous-map-compose[OF right-shift-cont q-cont])
have right-eq:
  (q ∘ (λx::real. 2 * x - 1)) = (λx::real. q (2 * x - 1))
  by (auto simp: fun-eq-iff o-def)
from right-cont-comp have right-cont:
  continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 * x}) X (λx::real.
  q (2 * x - 1))
  by (simp only: right-eq)
have cont-if:
  continuous-map ?T01 X (λx::real. if 2 * x ≤ 1 then p (2 * x) else q (2 * x -
  1))
proof (rule continuous-map-cases-le[where p = λx::real. 2 * x and q = λ-. 1])
  show continuous-map ?T01 euclideanreal (λx::real. 2 * x)
    by (simp add: continuous-map-iff-continuous continuous-intros)
  show continuous-map ?T01 euclideanreal (λ-. 1)
    by simp
  show continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 2 * x ≤ 1}) X
  (λx::real. p (2 * x))
    by (rule left-cont)
  show continuous-map (subtopology ?T01 {x ∈ topspace ?T01. 1 ≤ 2 * x}) X
  (λx::real. q (2 * x - 1))
    by (rule right-cont)
  show p (2 * x) = q (2 * x - 1)
    if x ∈ topspace ?T01 (λx::real. 2 * x) x = (λ-. 1) x for x
    using pq that by simp
qed
have cont-join: continuous-map ?T01 X (joinpathin p q)
proof (rule continuous-map-eq[OF cont-if])
  fix x :: real
  assume x-in: x ∈ topspace ?T01
  show (λx::real. if 2 * x ≤ 1 then p (2 * x) else q (2 * x - 1)) x = joinpathin
  p q x
    using x-in by (simp add: joinpathin-def field-simps)
qed
show ?thesis
  using cont-join unfolding pathin-def .
qed

```

```

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 -
  have aff-cont:
    continuous-map (top-of-set {0..1}) (top-of-set {0..1}) ( $\lambda t::real. (v - u) * t + u$ )
  proof -
    have aff-cont-eu:
      continuous-map (top-of-set {0..1}) euclideanreal ( $\lambda t::real. (v - u) * t + u$ )
    by (simp add: continuous-map-iff-continuous continuous-intros)
    have aff-range:
      ( $\lambda t::real. (v - u) * t + u$ )  $\in$  topspace (top-of-set {0..1})  $\rightarrow$  {0..1}
    proof
      fix x :: real
      assume x-in:  $x \in$  topspace (top-of-set {0..1})
      have x01:  $x \in$  {0..1}
        using x-in by simp
      have nonneg:  $0 \leq x * v + (1 - x) * u$ 
        using u v x01 by (intro add-nonneg-nonneg mult-nonneg-nonneg) auto
      moreover have le1:  $x * v + (1 - x) * u \leq 1$ 
        using u v x01 by (intro convex-bound-le) auto
      ultimately show  $((v - u) * x + u) \in$  {0..1}
        by (simp add: algebra-simps)
    qed
    show ?thesis
      using aff-cont-eu aff-range by (auto simp: continuous-map-in-subtopology)
    qed
  have continuous-map (top-of-set {0..1}) X (p  $\circ$  ( $\lambda t::real. (v - u) * t + u$ ))
    using aff-cont p unfolding pathin-def by (rule continuous-map-compose)
  then show ?thesis
    unfolding pathin-def subpathin-def by (simp add: o-def)
qed

lemma pathin-subdivision-open-cover:
  assumes p: pathin X p
    and cover: p ' {0..1}  $\subseteq$   $\bigcup \mathcal{U}$ 
    and openU:  $\bigwedge U. U \in \mathcal{U} \implies$  openin X U
  shows  $\exists n::nat. 0 < n \wedge$ 
    ( $\forall i < n. \exists U \in \mathcal{U}.
      \text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) p ' \{0..1\} \subseteq U$ )
proof -
  have U-nonempty:  $\mathcal{U} \neq \{\}$ 
  proof
    assume  $\mathcal{U} = \{\}$ 
    moreover have p 0  $\in$  p ' {0..1}
      by auto
    ultimately show False
      using cover by auto
  qed
  define W where

```

$W U = (\text{SOME } T. \text{open } T \wedge \{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap T)$  **for**  $U$   
**have**  $\text{pre-open}: \text{openin } (\text{top-of-set } \{0..1\}) \{t \in \{0..1\}. p t \in U\}$  **if**  $U\text{-in}: U \in \mathcal{U}$   
**for**  $U$   
**using**  $p \text{ open} U [OF \text{ } U\text{-in}]$   
**unfolding**  $\text{pathin-def continuous-map-def}$   
**by**  $\text{auto}$   
**have**  $W\text{-spec}:$   
 $\text{open } (W U)$   
 $\{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap W U$   
**if**  $U\text{-in}: U \in \mathcal{U}$  **for**  $U$   
**proof** –  
**from**  $\text{pre-open} [OF \text{ } U\text{-in}]$  **obtain**  $T$  **where**  
 $T\text{-open}: \text{open } T$   
**and**  $T\text{-eq}: \{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap T$   
**by**  $(\text{auto simp}: \text{openin-open})$   
**have**  $\text{open } (W U) \wedge \{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap W U$   
**unfolding**  $W\text{-def}$   
**by**  $(\text{rule someI-ex})$   $(\text{use } T\text{-open } T\text{-eq in blast})$   
**then show**  $\text{open } (W U) \{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap W U$   
**by**  $\text{auto}$   
**qed**  
**have**  $\text{interval-cover}: \{0..1\} \subseteq \bigcup (W \text{ ' } \mathcal{U})$   
**proof**  
**fix**  $t :: \text{real}$   
**assume**  $t\text{-in}: t \in \{0..1\}$   
**have**  $p t \in p \text{ ' } \{0..1\}$   
**using**  $t\text{-in}$  **by**  $\text{blast}$   
**then obtain**  $U$  **where**  $U\text{-in}: U \in \mathcal{U}$  **and**  $pU: p t \in U$   
**using**  $\text{cover}$  **by**  $\text{blast}$   
**have**  $\{t \in \{0..1\}. p t \in U\} = \{0..1\} \cap W U$   
**by**  $(\text{rule } W\text{-spec}(2) [OF \text{ } U\text{-in}])$   
**then have**  $t \in W U$   
**using**  $t\text{-in } pU$  **by**  $\text{auto}$   
**then show**  $t \in \bigcup (W \text{ ' } \mathcal{U})$   
**using**  $U\text{-in}$  **by**  $\text{blast}$   
**qed**  
**have**  $\text{open-}W\text{-image}: \text{open } B$  **if**  $B \in W \text{ ' } \mathcal{U}$  **for**  $B$   
**proof** –  
**from**  $\text{that}$  **obtain**  $U$  **where**  $U\text{-in}: U \in \mathcal{U}$  **and**  $B\text{-eq}: B = W U$   
**by**  $\text{blast}$   
**show**  $\text{open } B$   
**unfolding**  $B\text{-eq}$  **by**  $(\text{rule } W\text{-spec}(1) [OF \text{ } U\text{-in}])$   
**qed**  
**have**  $W\text{-nonempty}: W \text{ ' } \mathcal{U} \neq \{\}$   
**using**  $U\text{-nonempty}$  **by**  $\text{auto}$   
**have**  $\text{compact01}: \text{compact } (\{0..1::\text{real}\})$   
**by**  $(\text{rule compact-Icc})$   
**from**  $\text{Lebesgue-number-lemma} [OF \text{ } \text{compact01 } W\text{-nonempty } \text{interval-cover } \text{open-}W\text{-image}]$

```

obtain  $\delta$  where  $\delta$ -pos:  $0 < \delta$ 
  and  $\delta$ -cover:
     $\bigwedge T. T \subseteq \{0..1\} \implies \text{diameter } T < \delta \implies \exists B \in W \text{ ' } \mathcal{U}. T \subseteq B$ 
  by blast
obtain  $m$  where  $m$ : inverse (of-nat (Suc  $m$ ))  $< \delta$ 
  using reals-Archimedean[OF  $\delta$ -pos] by blast
let  $?n = \text{Suc } m$ 
have segment-cover:
   $\exists U \in \mathcal{U}. \text{subpathin } (\text{real } i / \text{real } ?n) (\text{real } (\text{Suc } i) / \text{real } ?n) p \text{ ' } \{0..1\} \subseteq U$ 
if  $i$ -lt:  $i < ?n$  for  $i$ 
proof -
  let  $?a = \text{real } i / \text{real } ?n$ 
  let  $?b = \text{real } (\text{Suc } i) / \text{real } ?n$ 
  let  $?T = \{?a..?b\}$ 
  have  $a01$ :  $?a \in \{0..1\}$ 
    using  $i$ -lt by auto
  have  $b01$ :  $?b \in \{0..1\}$ 
    using  $i$ -lt by auto
  have  $ab$ :  $?a \leq ?b$ 
    using  $i$ -lt by (simp add: field-simps)
  have  $T$ -subset:  $?T \subseteq \{0..1\}$ 
    using  $a01$   $b01$  by auto
  have diameter  $?T = ?b - ?a$ 
    using  $ab$  by simp
  also have  $\dots = 1 / \text{real } ?n$ 
    by (simp add: field-simps)
  also have  $\dots < \delta$ 
    using  $m$  by (simp add: divide-inverse)
  finally have  $T$ -small: diameter  $?T < \delta$  .
  from  $\delta$ -cover[OF  $T$ -subset  $T$ -small] obtain  $B$  where
     $B$ -in:  $B \in W \text{ ' } \mathcal{U}$ 
    and  $T$ -B:  $?T \subseteq B$ 
  by blast
obtain  $U$  where  $U$ -in:  $U \in \mathcal{U}$  and  $B$ -eq:  $B = W U$ 
  using  $B$ -in by blast
have pre-eq:  $\{t \in \{0..1\}. p \ t \in U\} = \{0..1\} \cap W U$ 
  by (rule W-spec(2)[OF  $U$ -in])
have subpath-subset: subpathin  $?a \ ?b \ p \text{ ' } \{0..1\} \subseteq U$ 
proof
  fix  $x$ 
  assume  $x$ -in:  $x \in \text{subpathin } ?a \ ?b \ p \text{ ' } \{0..1\}$ 
  then obtain  $t$  where  $t$ -in:  $t \in \{0..1\}$  and  $x$ -eq:  $x = \text{subpathin } ?a \ ?b \ p \ t$ 
  by blast
  have  $s$ -in- $T$ :  $((?b - ?a) * t + ?a) \in ?T$ 
proof -
  from  $t$ -in have  $t$ -ge0:  $0 \leq t$  and  $t$ -le1:  $t \leq 1$ 
  by auto
  have diff-nonneg:  $0 \leq ?b - ?a$ 
    using  $ab$  by linarith

```

```

have term-nonneg:  $0 \leq (?b - ?a) * t$ 
  by (rule mult-nonneg-nonneg[OF diff-nonneg t-ge0])
have term-le:  $(?b - ?a) * t \leq (?b - ?a) * 1$ 
  using t-le1 diff-nonneg by (rule mult-left-mono)
have lower:  $?a \leq (?b - ?a) * t + ?a$ 
  using term-nonneg by linarith
have upper:  $(?b - ?a) * t + ?a \leq ?b$ 
  using term-le by simp
show ?thesis
  using lower upper by simp
qed
have s-in-unit:  $((?b - ?a) * t + ?a) \in \{0..1\}$ 
  using s-in-T T-subset by blast
have s-in-W:  $((?b - ?a) * t + ?a) \in W \cup U$ 
  using s-in-T T-B B-eq by auto
have  $((?b - ?a) * t + ?a) \in \{t \in \{0..1\}. p \ t \in U\}$ 
  using s-in-unit s-in-W pre-eq by auto
then show  $x \in U$ 
  using x-eq by (simp add: subpathin-def)
qed
then show ?thesis
  using U-in by blast
qed
show ?thesis
  by (rule exI[of - ?n]) (use segment-cover in auto)
qed

lemma loopin-space-joinpathin:
  assumes  $p: p \in \text{loopin-space } X \ x$ 
  and  $q: q \in \text{loopin-space } X \ x$ 
  shows  $\text{joinpathin } p \ q \in \text{loopin-space } X \ x$ 
proof -
  from p obtain p-in where p-in:  $\text{pathin } X \ p \ p \ 0 = x \ p \ 1 = x$ 
  by (rule loopin-spaceE)
  from q obtain q-in where q-in:  $\text{pathin } X \ q \ q \ 0 = x \ q \ 1 = x$ 
  by (rule loopin-spaceE)
  have  $\text{pathin } X \ (\text{joinpathin } p \ q)$ 
  by (rule pathin-joinpathin[OF p-in(1) q-in(1)]) (simp add: p-in q-in)
  moreover have  $\text{joinpathin } p \ q \ 0 = x \ \text{joinpathin } p \ q \ 1 = x$ 
  by (simp-all add: joinpathin-def p-in q-in)
  ultimately show ?thesis
  by (rule loopin-spaceI)
qed

lemma homotopic-paths-in-top-of-set [simp]:
  homotopic-paths-in (top-of-set S)  $p \ q \iff \text{homotopic-paths } S \ p \ q$ 
  unfolding homotopic-paths-in-def homotopic-paths-def
  by (simp add: pathstart-def pathfinish-def)

```

**lemma** *homotopic-loopsin-top-of-set* [simp]:  
*homotopic-loopsin (top-of-set S) p q*  $\longleftrightarrow$  *homotopic-loops S p q*  
**unfolding** *homotopic-loopsin-def homotopic-loops-def*  
**by** (*simp add: pathstart-def pathfinish-def*)

**lemma** *homotopic-pathsin-imp-pathin*:  
**assumes** *homotopic-pathsin X p q*  
**shows** *pathin X p pathin X q*  
**using** *assms* **unfolding** *homotopic-pathsin-def pathin-def*  
**by** (*auto dest: homotopic-with-imp-continuous-maps*)

**lemma** *homotopic-pathsin-imp-endpoints*:  
**assumes** *homotopic-pathsin X p q*  
**shows**  $q\ 0 = p\ 0$   $q\ 1 = p\ 1$   
**using** *assms* **unfolding** *homotopic-pathsin-def*  
**by** (*auto dest: homotopic-with-imp-property*)

**lemma** *homotopic-pathsin-refl* [simp]:  
*homotopic-pathsin X p p*  $\longleftrightarrow$  *pathin X p*  
**unfolding** *homotopic-pathsin-def pathin-def*  
**by** (*simp add: homotopic-with-refl*)

**lemma** *homotopic-pathsin-sym*:  
**assumes** *homotopic-pathsin X p q*  
**shows** *homotopic-pathsin X q p*  
**proof** –  
**have**  $q\ 0 = p\ 0$   $q\ 1 = p\ 1$   
**using** *assms* **by** (*rule homotopic-pathsin-imp-endpoints*)  
**with** *assms* **show** *?thesis*  
**unfolding** *homotopic-pathsin-def*  
**by** (*simp add: homotopic-with-sym*)  
**qed**

**lemma** *homotopic-pathsin-trans*:  
**assumes** *homotopic-pathsin X p q*  
**and** *homotopic-pathsin X q r*  
**shows** *homotopic-pathsin X p r*  
**proof** –  
**have**  $q: q\ 0 = p\ 0$   $q\ 1 = p\ 1$   
**using** *assms*(1) **by** (*rule homotopic-pathsin-imp-endpoints*)  
**have**  $pq: \text{homotopic-with } (\lambda s. s\ 0 = p\ 0 \wedge s\ 1 = p\ 1) \text{ (top-of-set } \{0..1\}) X p q$   
**using** *assms*(1) **unfolding** *homotopic-pathsin-def* .  
**have**  $qr: \text{homotopic-with } (\lambda s. s\ 0 = p\ 0 \wedge s\ 1 = p\ 1) \text{ (top-of-set } \{0..1\}) X q r$   
**using** *assms*(2)  $q$  **unfolding** *homotopic-pathsin-def* **by** *simp*  
**from**  $pq\ qr$  **show** *?thesis*  
**unfolding** *homotopic-pathsin-def* **by** (*rule homotopic-with-trans*)  
**qed**

**lemma** *homotopic-pathsin-eq*:

```

assumes pathin  $X$   $p$ 
  and  $\bigwedge t. t \in \{0..1\} \implies p\ t = q\ t$ 
shows homotopic-paths  $\text{in } X$   $p$   $q$ 
proof –
  have continuous-map (top-of-set  $\{0..1\}$ )  $X$   $p$ 
    using assms(1) unfolding pathin-def by simp
  moreover have  $\forall x \in \text{topspace } (\text{top-of-set } \{0..1\}). p\ x = q\ x$ 
    using assms(2) by simp
  ultimately show ?thesis
    unfolding homotopic-paths-in-def
    by (intro homotopic-with-equal) auto
qed

lemma continuous-map-homotopic-joinpathin-lemma:
  fixes  $p\ q :: \text{real} \Rightarrow \text{real} \Rightarrow 'a$ 
  assumes  $p$ :
    continuous-map (prod-topology (top-of-set  $\{0..1\}$ ) (top-of-set  $\{0..1\}$ ))  $X$ 
      ( $\lambda y. p\ (\text{fst } y)\ (\text{snd } y)$ )
  and  $q$ :
    continuous-map (prod-topology (top-of-set  $\{0..1\}$ ) (top-of-set  $\{0..1\}$ ))  $X$ 
      ( $\lambda y. q\ (\text{fst } y)\ (\text{snd } y)$ )
  and  $pq$ :  $\bigwedge t. t \in \{0..1\} \implies p\ t\ 1 = q\ t\ 0$ 
shows
  continuous-map (prod-topology (top-of-set  $\{0..1\}$ ) (top-of-set  $\{0..1\}$ ))  $X$ 
    ( $\lambda y. \text{joinpathin } (p\ (\text{fst } y))\ (q\ (\text{fst } y))\ (\text{snd } y)$ )
proof –
  let  $?T01 = \text{subtopology euclideanreal } \{0..1\}$ 
  let  $?A = \text{prod-topology } ?T01\ ?T01$ 
  have sect:
    ( $\lambda t. p\ (\text{fst } t)\ (2 * \text{snd } t)$ ) = ( $\lambda y. p\ (\text{fst } y)\ (\text{snd } y)$ )  $\circ$  ( $\lambda y. (\text{fst } y, 2 * \text{snd } y)$ )
    ( $\lambda t. q\ (\text{fst } t)\ (2 * \text{snd } t - 1)$ ) = ( $\lambda y. q\ (\text{fst } y)\ (\text{snd } y)$ )  $\circ$  ( $\lambda y. (\text{fst } y, 2 * \text{snd } y - 1)$ )
  by force+
  show ?thesis
    unfolding joinpathin-def
  proof (rule continuous-map-cases-le)
    show continuous-map  $?A$  euclideanreal snd
      by (rule continuous-map-into-fulltopology[OF continuous-map-snd])
    show continuous-map  $?A$  euclideanreal ( $\lambda-. 1 / 2$ )
      by simp
    have left-pair-cont:
      continuous-map (subtopology  $?A$   $\{y \in \text{topspace } ?A. \text{snd } y \leq 1 / 2\}$ )  $?A$ 
        ( $\lambda t. (\text{fst } t, 2 * \text{snd } t)$ )
    proof (rule continuous-map-pairedI)
      show continuous-map (subtopology  $?A$   $\{y \in \text{topspace } ?A. \text{snd } y \leq 1 / 2\}$ )
         $?T01$  fst
        by (rule continuous-map-from-subtopology[OF continuous-map-fst])
      show continuous-map (subtopology  $?A$   $\{y \in \text{topspace } ?A. \text{snd } y \leq 1 / 2\}$ )
         $?T01$  ( $\lambda t. 2 * \text{snd } t$ )

```

```

proof –
  have snd-cont:
    continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2})
euclideanreal snd
    by (rule continuous-map-into-fulltopology[OF continuous-
ous-map-subtopology-snd])
  have scale-cont:
    continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2})
euclideanreal
    (λt. 2 * snd t)
  using snd-cont by (intro continuous-intros)
  have scale-range:
    (λt. 2 * snd t) ∈
    topspace (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) → {0..1}
proof
  fix t :: real × real
  assume t-in: t ∈ topspace (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1
/ 2})
  then have tA: t ∈ topspace ?A and t-half: snd t ≤ 1 / 2
  by auto
  have snd01: snd t ∈ {0..1}
  using tA by (auto simp: topspace-prod-topology)
  from snd01 have snd-ge0: 0 ≤ snd t and snd-le1: snd t ≤ 1
  by auto
  have lower: 0 ≤ 2 * (snd t :: real)
  using snd-ge0 by linarith
  have upper: 2 * (snd t :: real) ≤ 1
  using t-half by linarith
  have bounds: 0 ≤ 2 * (snd t :: real) ∧ 2 * (snd t :: real) ≤ 1
  using lower upper by blast
  show 2 * (snd t :: real) ∈ {0..1}
  proof –
    from bounds have lower': 0 ≤ 2 * (snd t :: real) and upper': 2 * (snd
t :: real) ≤ 1
    by auto
    show ?thesis
    using lower' upper' by (auto simp: atLeastAtMost-iff)
  qed
qed
show ?thesis
  by (rule continuous-map-into-subtopology[OF scale-cont scale-range])
qed
have left-comp:
  continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) X
  (((λy. p (fst y) (snd y)) ∘ (λy. (fst y, 2 * snd y))))
  by (rule continuous-map-compose[OF left-pair-cont p])
show continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) X
  (λt. p (fst t) (2 * snd t))

```

```

using left-comp by (simp add: o-def)
have right-pair-cont:
  continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) ?A
    (λt. (fst t, 2 * snd t - 1))
proof (rule continuous-map-pairedI)
  show continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y})
    ?T01 fst
    by (rule continuous-map-from-subtopology[OF continuous-map-fst])
  show continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y})
    ?T01 (λt. 2 * (snd t :: real) - 1)
    proof -
      have snd-cont:
        continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y})
          euclideanreal snd
          by (rule continuous-map-into-fulltopology[OF continu-
            ous-map-subtopology-snd])
      have affine-cont:
        continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y})
          euclideanreal
            (λt. 2 * (snd t :: real) - 1)
        using snd-cont by (intro continuous-intros)
      have affine-range:
        (λt. 2 * (snd t :: real) - 1) ∈
          topspace (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) → {0..1}
      proof
        fix t :: real × real
        assume t-in: t ∈ topspace (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd
          y})
          then have tA: t ∈ topspace ?A and t-half: 1 / 2 ≤ snd t
            by auto
          have snd01: snd t ∈ {0..1}
            using tA by (auto simp: topspace-prod-topology)
          from snd01 have snd-ge0: 0 ≤ snd t and snd-le1: snd t ≤ 1
            by auto
          have lower: 0 ≤ 2 * (snd t :: real) - 1
            using t-half by linarith
          have upper: 2 * (snd t :: real) - 1 ≤ 1
            using snd-le1 by linarith
          show 2 * (snd t :: real) - 1 ∈ {0..1}
            using lower upper by (auto simp: atLeastAtMost-iff)
        qed
      show ?thesis
        by (rule continuous-map-into-subtopology[OF affine-cont affine-range])
      qed
    qed
have right-comp:
  continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) X
    (((λy. q (fst y) (snd y)) ∘ (λy. (fst y, 2 * snd y - 1))))
  by (rule continuous-map-compose[OF right-pair-cont q])

```

```

show continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) X
  (λt. q (fst t) (2 * snd t - 1))
  using right-comp by (simp add: o-def)
show p (fst y) (2 * snd y) = q (fst y) (2 * snd y - 1)
  if y ∈ topspace ?A snd y = 1 / 2 for y
proof -
  have fst01: fst y ∈ {0..1}
    using that(1) by (auto simp: topspace-prod-topology)
  have mid1: 2 * snd y = 1
    using that(2) by simp
  have mid0: 2 * snd y - 1 = 0
    using that(2) by simp
  have p (fst y) 1 = q (fst y) 0
    by (rule pq[OF fst01])
  then show ?thesis
    using mid1 mid0 by simp
qed
qed
qed

```

```

lemma homotopic-paths-in-continuous-image:
  assumes pq: homotopic-paths-in X p q
    and h: continuous-map X Y f
  shows homotopic-paths-in Y (f ∘ p) (f ∘ q)
proof -
  have pq': homotopic-with (λr. r 0 = p 0 ∧ r 1 = p 1) (top-of-set {0..1}) X p q
    using pq unfolding homotopic-paths-in-def .
  show ?thesis
    unfolding homotopic-paths-in-def
    by (rule homotopic-with-compose-continuous-map-left[OF pq' h]) simp
qed

```

```

lemma homotopic-paths-in-reversepath-in-D:
  assumes homotopic-paths-in X p q
  shows homotopic-paths-in X (reversepath-in p) (reversepath-in q)
proof -
  have rev-cont: continuous-map (top-of-set {0..1}) (top-of-set {0..1}) (λt::real.
1 - t)
  by (intro continuous-map-into-subtopology continuous-intros) auto
  have pq':
    homotopic-with (λr. r 0 = p 0 ∧ r 1 = p 1) (top-of-set {0..1}) X p q
    using assms unfolding homotopic-paths-in-def .
  have homotopic-with (λr. r 0 = reversepath-in p 0 ∧ r 1 = reversepath-in p 1)
    (top-of-set {0..1}) X (p ∘ (λt::real. 1 - t)) (q ∘ (λt::real. 1 - t))
  proof (rule homotopic-with-compose-continuous-map-right[OF pq' rev-cont])
  fix j :: real ⇒ 'a
  assume j-end: j 0 = p 0 ∧ j 1 = p 1
  show (λr. r 0 = reversepath-in p 0 ∧ r 1 = reversepath-in p 1)
    (j ∘ (λt::real. 1 - t))

```

```

    using j-end by (simp add: reversepathin-def)
  qed
  then show ?thesis
    unfolding homotopic-pathsin-def reversepathin-def o-def .
  qed

lemma homotopic-pathsin-reversepathin:
  homotopic-pathsin X (reversepathin p) (reversepathin q)  $\longleftrightarrow$  homotopic-pathsin
  X p q
  using homotopic-pathsin-reversepathin-D by force

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 -
  let ?T01 = subtopology euclideanreal {0..1}
  let ?A = prod-topology ?T01 ?T01
  obtain h1 :: real  $\times$  real  $\Rightarrow$  'a where
    h1-cont: continuous-map (prod-topology (top-of-set {0..1}) (top-of-set {0..1}))
  X h1
    and h1-0:  $\forall x. h1 (0, x) = p x$ 
    and h1-1:  $\forall x. h1 (1, x) = p' x$ 
    and h1-prop:  $\forall t \in \{0..1\}. h1 (t, 0) = p 0 \wedge h1 (t, 1) = p 1$ 
    using pp' unfolding homotopic-pathsin-def homotopic-with-def by auto
  obtain h2 :: real  $\times$  real  $\Rightarrow$  'a where
    h2-cont: continuous-map (prod-topology (top-of-set {0..1}) (top-of-set {0..1}))
  X h2
    and h2-0:  $\forall x. h2 (0, x) = q x$ 
    and h2-1:  $\forall x. h2 (1, x) = q' x$ 
    and h2-prop:  $\forall t \in \{0..1\}. h2 (t, 0) = q 0 \wedge h2 (t, 1) = q 1$ 
    using qq' unfolding homotopic-pathsin-def homotopic-with-def by auto
  define k1 where k1 t x = h1 (t, x) for t x
  define k2 where k2 t x = h2 (t, x) for t x
  have k1-cont: continuous-map ?A X ( $\lambda y. k1 (fst y) (snd y)$ )
proof -
  have ( $\lambda y. k1 (fst y) (snd y)$ ) = h1
    unfolding k1-def by (auto simp: fun-eq-iff)
  then show ?thesis
    using h1-cont by simp
  qed
  have k2-cont: continuous-map ?A X ( $\lambda y. k2 (fst y) (snd y)$ )
proof -
  have ( $\lambda y. k2 (fst y) (snd y)$ ) = h2
    unfolding k2-def by (auto simp: fun-eq-iff)
  then show ?thesis
    using h2-cont by simp
  qed
  qed

```

```

have k1-0:  $\forall x. k1\ 0\ x = p\ x$ 
  unfolding k1-def using h1-0 by simp
have k1-1:  $\forall x. k1\ 1\ x = p'\ x$ 
  unfolding k1-def using h1-1 by simp
have k2-0:  $\forall x. k2\ 0\ x = q\ x$ 
  unfolding k2-def using h2-0 by simp
have k2-1:  $\forall x. k2\ 1\ x = q'\ x$ 
  unfolding k2-def using h2-1 by simp
have k1-prop:  $\forall t \in \{0..1\}. k1\ t\ 0 = p\ 0 \wedge k1\ t\ 1 = p\ 1$ 
  unfolding k1-def using h1-prop by simp
have k2-prop:  $\forall t \in \{0..1\}. k2\ t\ 0 = q\ 0 \wedge k2\ t\ 1 = q\ 1$ 
  unfolding k2-def using h2-prop by simp
have mid-eq:  $k1\ t\ 1 = k2\ t\ 0$  if  $t \in \{0..1\}$  for  $t$ 
  using k1-prop[rule-format, OF that] k2-prop[rule-format, OF that] pq by auto
show ?thesis
  unfolding homotopic-paths-in-def homotopic-with-def
proof (rule exI[where  $x = \lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y)$ ], intro conjI)
  show continuous-map ?A X  $(\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))$ 
    by (rule continuous-map-homotopic-joinpathin-lemma[OF k1-cont k2-cont])
  (rule mid-eq)
  show  $\forall x :: \text{real}. (\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (0, x) = \text{joinpathin}\ p\ q\ x$ 
    proof
      fix x :: real
      show  $(\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (0, x) = \text{joinpathin}\ p\ q\ x$ 
        using k1-0 k2-0 by (cases  $x \leq 1 / 2$ ) (auto simp: joinpathin-def)
    qed
  show  $\forall x :: \text{real}. (\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (1, x) = \text{joinpathin}\ p'\ q'\ x$ 
    proof
      fix x :: real
      show  $(\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (1, x) = \text{joinpathin}\ p'\ q'\ x$ 
        using k1-1 k2-1 by (cases  $x \leq 1 / 2$ ) (auto simp: joinpathin-def)
    qed
  show  $\forall t \in \{0..1\}. (\lambda r. r\ 0 = \text{joinpathin}\ p\ q\ 0 \wedge r\ 1 = \text{joinpathin}\ p\ q\ 1)$ 
     $(\lambda x. (\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (t, x))$ 
    proof
      fix t :: real
      assume t:  $t \in \{0..1\}$ 
      have k1t0:  $k1\ t\ 0 = p\ 0$ 
        using k1-prop[rule-format, OF t] by auto
      have k2t1:  $k2\ t\ 1 = q\ 1$ 
        using k2-prop[rule-format, OF t] by auto
      show  $(\lambda r. r\ 0 = \text{joinpathin}\ p\ q\ 0 \wedge r\ 1 = \text{joinpathin}\ p\ q\ 1)$ 
         $(\lambda x. (\lambda y. \text{joinpathin}\ (k1\ (fst\ y))\ (k2\ (fst\ y))\ (snd\ y))\ (t, x))$ 
        using k1t0 k2t1 by (simp add: joinpathin-def)
    qed

```

qed  
 qed  
 qed

**lemma** *homotopic-paths-in-reparametrize*:

**assumes**  $p$ : *pathin*  $X$   $p$

**and**  $contf$ : *continuous-map* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ )  $f$

**and**  $f01$ :  $f \in \{0..1\} \rightarrow \{0..1\}$

**and** [*simp*]:  $f\ 0 = 0\ f\ 1 = 1$

**and**  $q$ :  $\bigwedge t. t \in \{0..1\} \implies q\ t = p\ (f\ t)$

**shows** *homotopic-paths-in*  $X$   $p$   $q$

**proof** –

**have**  $pf$ : *pathin*  $X$  ( $p \circ f$ )

**using**  $contf$   $p$  **unfolding** *pathin-def* **by** (*rule continuous-map-compose*)

**have**  $qf'$ : *homotopic-paths-in*  $X$  ( $p \circ f$ )  $q$

**using**  $pf$   $q$  **by** (*intro homotopic-paths-in-eq*) *auto*

**have**  $qf$ : *homotopic-paths-in*  $X$   $q$  ( $p \circ f$ )

**by** (*rule homotopic-paths-in-sym*[*OF*  $qf'$ ])

**have**  $fp$ : *homotopic-paths-in*  $X$  ( $p \circ f$ )  $p$

**unfolding** *homotopic-paths-in-def* *homotopic-with-def*

**proof** (*rule exI*[**where**  $x = p \circ (\lambda y. (1 - fst\ y) * ((f \circ snd)\ y) + fst\ y * snd\ y)$ ], *intro conjI*)

**have**  $snd-cont$ :

*continuous-map* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ ))  
 (*top-of-set*  $\{0..1\}$ )  $snd$

**by** (*rule continuous-map-snd*)

**have**  $fst-cont$ :

*continuous-map* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ ))  
 (*top-of-set*  $\{0..1\}$ )  $fst$

**by** (*rule continuous-map-fst*)

**have**  $fsnd-cont$ :

*continuous-map* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ ))  
 (*top-of-set*  $\{0..1\}$ ) ( $f \circ snd$ )

**by** (*rule continuous-map-compose*[*OF*  $snd-cont$   $contf$ ])

**have**  $param-cont-eu$ :

*continuous-map* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ )) *euclideanreal*

$(\lambda y. (1 - fst\ y) * ((f \circ snd)\ y) + fst\ y * snd\ y)$

**using**  $fst-cont$   $snd-cont$   $fsnd-cont$

**by** (*intro continuous-intros*

*continuous-map-into-fulltopology*[*OF*  $fst-cont$ ]

*continuous-map-into-fulltopology*[*OF*  $snd-cont$ ]

*continuous-map-into-fulltopology*[*OF*  $fsnd-cont$ ])

**have**  $param-range$ :

$(\lambda y. (1 - fst\ y) * ((f \circ snd)\ y) + fst\ y * snd\ y) \in$

*topspace* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ ))  $\rightarrow \{0..1\}$

**proof**

**fix**  $y$  :: *real*  $\times$  *real*

**assume**  $y$ -in:  $y \in$  *topspace* (*prod-topology* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*

```

{0..1}))
  have fst01: fst y ∈ {0..1} and snd01: snd y ∈ {0..1}
    using y-in by (auto simp: topspace-prod-topology)
  have fsnd01: f (snd y) ∈ {0..1}
    using f01 snd01 by auto
  have nonneg: 0 ≤ (1 - fst y) * (f (snd y)) + fst y * snd y
    using fst01 snd01 fsnd01 by (intro add-nonneg-nonneg mult-nonneg-nonneg)
auto
  moreover have le1: (1 - fst y) * (f (snd y)) + fst y * snd y ≤ 1
    using fst01 snd01 fsnd01 by (intro convex-bound-le) auto
  ultimately show ((1 - fst y) * ((f ∘ snd) y) + fst y * snd y) ∈ {0..1}
    by simp
qed
have param-cont:
  continuous-map (prod-topology (top-of-set {0..1}) (top-of-set {0..1}))
    (top-of-set {0..1}) (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)
    using param-cont-eu param-range by (simp add: continu-
ous-map-in-subtopology)
  show continuous-map
    (prod-topology (top-of-set {0..1}) (top-of-set {0..1})) X
    (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y))
    using p unfolding pathin-def by (rule continuous-map-compose[OF
param-cont])
  show ∀ x::real. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (0, x) =
(p ∘ f) x
  proof
    fix x :: real
    show (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (0, x) = (p ∘ f)
x
    by (simp add: o-def algebra-simps)
  qed
  show ∀ x::real. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (1, x) =
p x
  proof
    fix x :: real
    show (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (1, x) = p x
    by (simp add: o-def algebra-simps)
  qed
  show ∀ t ∈ {0..1}. (λr. r 0 = (p ∘ f) 0 ∧ r 1 = (p ∘ f) 1)
    (λx. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (t, x))
  proof
    fix t :: real
    assume t: t ∈ {0..1}
    show (λr. r 0 = (p ∘ f) 0 ∧ r 1 = (p ∘ f) 1)
      (λx::real. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y)) (t, x))
    proof -
      have end0: (λx::real. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y))
(t, x)) 0 = (p ∘ f) 0
        by (simp add: o-def algebra-simps)

```

```

      have end1: (λx::real. (p ∘ (λy. (1 - fst y) * ((f ∘ snd) y) + fst y * snd y))
(t, x)) 1 = (p ∘ f) 1
      by (simp add: o-def algebra-simps)
      from end0 end1 show ?thesis
      by (simp add: o-def)
    qed
  qed
  have homotopic-pathsin X q p
    by (rule homotopic-pathsin-trans[OF gf fp])
  then show ?thesis
    by (rule homotopic-pathsin-sym)
  qed

lemma homotopic-pathsin-rid-const:
  assumes p: pathin X p
  shows homotopic-pathsin X (joinpathin p (λ-. p 1)) p
proof -
  have contf: continuous-map (top-of-set {0..1}) (top-of-set {0..1})
    (λt::real. if t ≤ 1 / 2 then 2 * t else 1)
  proof (rule continuous-map-into-subtopology)
    show continuous-map (top-of-set {0..1}) euclideanreal
      (λt::real. if t ≤ 1 / 2 then 2 * t else 1)
    proof (rule continuous-map-cases-le[where p = λt::real. t and q = λ-. 1 / 2])
      show continuous-map (top-of-set {0..1}) euclideanreal (λt::real. t)
        by (simp add: continuous-map-iff-continuous)
      show continuous-map (top-of-set {0..1}) euclideanreal (λ-. 1 / 2)
        by simp
    show continuous-map (subtopology (top-of-set {0..1}) {x ∈ topspace (top-of-set
{0..1}). x ≤ 1 / 2})
      euclideanreal (λt::real. 2 * t)
    proof -
      have continuous-map (top-of-set {0..1}) euclideanreal (λt::real. 2 * t)
        by (simp add: continuous-map-iff-continuous continuous-intros)
      then show ?thesis
        by (rule continuous-map-from-subtopology)
    qed
    show continuous-map (subtopology (top-of-set {0..1}) {x ∈ topspace (top-of-set
{0..1}). 1 / 2 ≤ x})
      euclideanreal (λ::real. 1)
    by simp
    show 2 * x = (1::real) if x ∈ topspace (top-of-set {0..1}) x = 1 / 2 for x
      using that by simp
    qed
    show (λt::real. if t ≤ 1 / 2 then 2 * t else 1) ∈ topspace (top-of-set {0..1})
    → {0..1}
    by auto
  qed
  show ?thesis

```

```

    apply (rule homotopic-paths-in-sym)
    apply (rule homotopic-paths-in-reparametrize[where  $f = \lambda t::\text{real. if } t \leq 1 / 2$ 
then  $2 * t$  else  $1$ ])
    using p contf
    apply (auto simp: joinpath-in-def)
    done
qed

lemma homotopic-paths-in-lid-const:
  assumes p: path-in X p
  shows homotopic-paths-in X (joinpath-in ( $\lambda \cdot. p$  0) p) p
proof -
  have contf: continuous-map (top-of-set {0..1}) (top-of-set {0..1})
    ( $\lambda t::\text{real. if } t \leq 1 / 2$  then 0 else  $2 * t - 1$ )
  proof (rule continuous-map-into-subtopology)
    show continuous-map (top-of-set {0..1}) euclidean-real
      ( $\lambda t::\text{real. if } t \leq 1 / 2$  then 0 else  $2 * t - 1$ )
    proof (rule continuous-map-cases-le[where  $p = \lambda t::\text{real. } t$  and  $q = \lambda \cdot. 1 / 2$ ])
      show continuous-map (top-of-set {0..1}) euclidean-real ( $\lambda t::\text{real. } t$ )
        by (simp add: continuous-map-iff-continuous)
      show continuous-map (top-of-set {0..1}) euclidean-real ( $\lambda \cdot. 1 / 2$ )
        by simp
      show continuous-map (subtopology (top-of-set {0..1}) { $x \in \text{topspace (top-of-set$ 
{0..1}).  $x \leq 1 / 2$ }
euclidean-real ( $\lambda \cdot::\text{real. } 0$ )
        by simp
      show continuous-map (subtopology (top-of-set {0..1}) { $x \in \text{topspace (top-of-set$ 
{0..1}).  $1 / 2 \leq x$ }
euclidean-real ( $\lambda t::\text{real. } 2 * t - 1$ )
    proof -
      have continuous-map (top-of-set {0..1}) euclidean-real ( $\lambda t::\text{real. } 2 * t - 1$ )
        by (simp add: continuous-map-iff-continuous continuous-intros)
      then show ?thesis
        by (rule continuous-map-from-subtopology)
    qed
    show ( $0::\text{real}$ ) =  $2 * x - 1$  if  $x \in \text{topspace (top-of-set {0..1})}$   $x = 1 / 2$  for
x
      using that by simp
    qed
    show ( $\lambda t::\text{real. if } t \leq 1 / 2$  then 0 else  $2 * t - 1$ )  $\in \text{topspace (top-of-set {0..1})}$ 
 $\rightarrow \{0..1\}$ 
      by auto
    qed
    show ?thesis
      apply (rule homotopic-paths-in-sym)
      apply (rule homotopic-paths-in-reparametrize[where  $f = \lambda t::\text{real. if } t \leq 1 / 2$ 
then 0 else  $2 * t - 1$ ])
      using p contf
      apply (auto simp: joinpath-in-def)

```

done  
qed

**lemma** *homotopic-paths-in-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-paths-in*  $X$  (*joinpathin*  $p$  (*joinpathin*  $q$   $r$ )) (*joinpathin* (*joinpathin*  $p$   $q$ )  $r$ )  
**proof** –  
 let  $?f = \lambda t::\text{real. if } t \leq 1 / 2 \text{ then inverse } 2 * t \text{ else if } t \leq 3 / 4 \text{ then } t - 1 / 4 \text{ else } 2 * t - 1$   
 have *contf-on*: *continuous-on*  $\{0..1\}$   $?f$   
**proof** (*rule continuous-on-cases-1*)  
 show *continuous-on*  $\{t \in \{0..1\}. t \leq 1 / 2\}$  ( $\lambda t::\text{real. inverse } 2 * t$ )  
 by (*intro continuous-intros*)  
 show *continuous-on*  $\{t \in \{0..1\}. 1 / 2 \leq t\}$   
 ( $\lambda t::\text{real. if } t \leq 3 / 4 \text{ then } t - 1 / 4 \text{ else } 2 * t - 1$ )  
**proof** (*rule continuous-on-cases-1*)  
 show *continuous-on*  $\{t \in \{t \in \{0..1\}. 1 / 2 \leq t\}. t \leq 3 / 4\}$  ( $\lambda t::\text{real. } t - 1 / 4$ )  
 by (*intro continuous-intros*)  
 show *continuous-on*  $\{t \in \{t \in \{0..1\}. 1 / 2 \leq t\}. 3 / 4 \leq t\}$  ( $\lambda t::\text{real. } 2 * t - 1$ )  
 by (*intro continuous-intros*)  
 show  $(3 / 4::\text{real}) \in \{t \in \{0..1\}. (1 / 2::\text{real}) \leq t\} \implies$   
 $(3 / 4::\text{real}) - 1 / 4 = 2 * (3 / 4) - 1$   
 by *simp*  
**qed**  
 show  $(1 / 2::\text{real}) \in \{0..1\} \implies$   
 $\text{inverse } 2 * (1 / 2::\text{real}) =$   
 $(\text{if } (1 / 2::\text{real}) \leq 3 / 4 \text{ then } 1 / 2 - 1 / 4 \text{ else } 2 * (1 / 2) - 1)$   
 by *simp*  
**qed**  
 have *contf-eu*: *continuous-map* (*top-of-set*  $\{0..1\}$ ) *euclideanreal*  $?f$   
 using *contf-on* by *simp*  
 have *contf*:  
*continuous-map* (*top-of-set*  $\{0..1\}$ ) (*top-of-set*  $\{0..1\}$ )  $?f$   
 using *contf-eu* by (*rule continuous-map-into-subtopology*) *auto*  
 have *join-pq*: pathin  $X$  (*joinpathin*  $p$   $q$ )  
 by (*rule pathin-joinpathin*[*OF*  $p$   $q$   $pq$ ])  
 have *join-qr*: pathin  $X$  (*joinpathin*  $q$   $r$ )  
 by (*rule pathin-joinpathin*[*OF*  $q$   $r$   $qr$ ])  
 have *join-pq-end*: *joinpathin*  $p$   $q \ 1 = r \ 0$   
 using *qr* by (*simp add: joinpathin-def*)  
 have *join-pq-r*: pathin  $X$  (*joinpathin* (*joinpathin*  $p$   $q$ )  $r$ )  
 by (*rule pathin-joinpathin*[*OF* *join-pq*  $r$  *join-pq-end*])

```

have join-pq-r-expanded:
  pathin X (λt. if t * 2 ≤ 1
    then if 2 * t * 2 ≤ 1 then p (2 * (2 * t)) else q (2 * (2 * t) - 1)
    else r (2 * t - 1))
  using join-pq-r by (simp add: joinpathin-def)
show ?thesis
  apply (rule homotopic-pathsin-sym)
  apply (rule homotopic-pathsin-reparametrize[
    where f = ?f])
  using contf join-pq-r-expanded join-pq-r join-pq r qr join-qr p q pq
  apply (auto simp: joinpathin-def field-simps algebra-simps)
  apply (rule arg-cong[where f = q])
  apply (simp add: field-simps)
  done
qed

lemma homotopic-pathsin-rinv-const:
  assumes p: pathin X p
  shows homotopic-pathsin X (joinpathin p (reversepathin p)) (λ-. p 0)
proof -
  let ?T01 = subtopology euclideanreal {0..1}
  let ?A = prod-topology ?T01 ?T01
  let ?h =
    λy. if snd y ≤ 1 / 2
      then p (fst y * (2 * snd y))
      else p (fst y * (1 - (2 * snd y - 1)))
  have p-cont: continuous-map ?T01 X p
  using p unfolding pathin-def .
  have h-cont: continuous-map ?A X ?h
  proof (rule continuous-map-cases-le)
  show continuous-map ?A euclideanreal snd
    by (rule continuous-map-into-fulltopology[OF continuous-map-snd])
  show continuous-map ?A euclideanreal (λ-. 1 / 2)
    by simp
  have left-param-cont-eu:
    continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) euclidean-
    real
    (λt. fst t * (2 * snd t))
  by (intro continuous-intros
    continuous-map-into-fulltopology[OF continuous-map-subtopology-fst]
    continuous-map-into-fulltopology[OF continuous-map-subtopology-snd])
  have left-param-cont:
    continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) ?T01
    (λt. fst t * (2 * snd t))
  proof -
  have left-param-range:
    (λt. fst t * (2 * snd t)) ∈
    topspace (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) → {0..1}
  proof

```

```

fix t :: real × real
assume t-in: t ∈ topspace (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 /
2})
then have tA: t ∈ topspace ?A and t-half: snd t ≤ 1 / 2
  by auto
have fst01: fst t ∈ {0..1} and snd01: snd t ∈ {0..1}
  using tA by (auto simp: topspace-prod-topology)
from snd01 have snd-ge0: 0 ≤ snd t and snd-le1: snd t ≤ 1
  by auto
have scale01: 2 * (snd t :: real) ∈ {0..1}
proof –
  have lower: 0 ≤ 2 * (snd t :: real)
    using snd-ge0 by linarith
  have upper: 2 * (snd t :: real) ≤ 1
    using t-half by linarith
  show 2 * (snd t :: real) ∈ {0..1}
    using lower upper by auto
qed
show fst t * (2 * (snd t :: real)) ∈ {0..1}
proof –
  from fst01 have fst-ge0: 0 ≤ fst t and fst-le1: fst t ≤ 1
    by auto
  from scale01 have scale-ge0: 0 ≤ 2 * (snd t :: real) and scale-le1: 2 *
(snd t :: real) ≤ 1
    by auto
  have lower: 0 ≤ fst t * (2 * (snd t :: real))
    by (rule mult-nonneg-nonneg[OF fst-ge0 scale-ge0])
  have upper: fst t * (2 * (snd t :: real)) ≤ 1
    by (rule mult-le-one[OF fst-le1 scale-ge0 scale-le1])
  have bounds:
    0 ≤ fst t * (2 * (snd t :: real)) ∧ fst t * (2 * (snd t :: real)) ≤ 1
    using lower upper by blast
  show fst t * (2 * (snd t :: real)) ∈ {0..1}
proof –
  from bounds have lower': 0 ≤ fst t * (2 * (snd t :: real))
    and upper': fst t * (2 * (snd t :: real)) ≤ 1
    by auto
  show ?thesis
    using lower' upper' by (auto simp: atLeastAtMost-iff)
qed
qed
show ?thesis
  by (rule continuous-map-into-subtopology[OF left-param-cont-eu
left-param-range])
qed
show continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) X
(λt. p (fst t * (2 * snd t)))
proof –

```

```

have left-comp:
  continuous-map (subtopology ?A {y ∈ topspace ?A. snd y ≤ 1 / 2}) X
    (p ∘ (λt. fst t * (2 * snd t)))
  by (rule continuous-map-compose[OF left-param-cont p-cont])
have left-eq: (λt. p (fst t * (2 * snd t))) = p ∘ (λt. fst t * (2 * snd t))
  by (auto simp: fun-eq-iff)
show ?thesis
  using left-comp left-eq by simp
qed
have right-param-cont-eu:
  continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) euclidean-
real
  (λt. fst t * (1 - (2 * snd t - 1)))
  by (intro continuous-intros
    continuous-map-into-fulltopology[OF continuous-map-subtopology-fst]
    continuous-map-into-fulltopology[OF continuous-map-subtopology-snd])
have right-param-cont:
  continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) ?T01
  (λt. fst t * (1 - (2 * snd t - 1)))
proof -
  have right-param-range:
  (λt. fst t * (1 - (2 * snd t - 1))) ∈
    topspace (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) → {0..1}
proof
  fix t :: real × real
  assume t-in: t ∈ topspace (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd
y})
  then have tA: t ∈ topspace ?A and t-half: 1 / 2 ≤ snd t
  by auto
  have fst01: fst t ∈ {0..1} and snd01: snd t ∈ {0..1}
  using tA by (auto simp: topspace-prod-topology)
  from snd01 have snd-ge0: 0 ≤ snd t and snd-le1: snd t ≤ 1
  by auto
  have scale01: 1 - (2 * (snd t :: real) - 1) ∈ {0..1}
  proof -
  have lower: 0 ≤ 1 - (2 * (snd t :: real) - 1)
  using snd-le1 by linarith
  have upper: 1 - (2 * (snd t :: real) - 1) ≤ 1
  using t-half by linarith
  show 1 - (2 * (snd t :: real) - 1) ∈ {0..1}
  using lower upper by force
  qed
  show fst t * (1 - (2 * (snd t :: real) - 1)) ∈ {0..1}
  proof -
  from fst01 have fst-ge0: 0 ≤ fst t and fst-le1: fst t ≤ 1
  by auto
  from scale01 have scale-ge0: 0 ≤ 1 - (2 * (snd t :: real) - 1)
  and scale-le1: 1 - (2 * (snd t :: real) - 1) ≤ 1
  by auto

```

```

    have lower:  $0 \leq \text{fst } t * (1 - (2 * (\text{snd } t :: \text{real}) - 1))$ 
      by (rule mult-nonneg-nonneg[OF fst-ge0 scale-ge0])
    have upper:  $\text{fst } t * (1 - (2 * (\text{snd } t :: \text{real}) - 1)) \leq 1$ 
      by (rule mult-le-one[OF fst-le1 scale-ge0 scale-le1])
    show  $\text{fst } t * (1 - (2 * (\text{snd } t :: \text{real}) - 1)) \in \{0..1\}$ 
      using lower upper by force
  qed
  qed
  show ?thesis
    using right-param-cont-eu right-param-range by (simp add: continuous-map-in-subtopology)
  qed
  show continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) X
    (λt. p (fst t * (1 - (2 * snd t - 1))))
  proof -
    have right-comp:
      continuous-map (subtopology ?A {y ∈ topspace ?A. 1 / 2 ≤ snd y}) X
        (p ∘ (λt. fst t * (1 - (2 * snd t - 1))))
      by (rule continuous-map-compose[OF right-param-cont p-cont])
    have right-eq:
      (λt. p (fst t * (1 - (2 * snd t - 1)))) =
        p ∘ (λt. fst t * (1 - (2 * snd t - 1)))
      by (auto simp: fun-eq-iff)
    show ?thesis
      using right-comp right-eq by simp
  qed
  show p (fst y * (2 * snd y)) = p (fst y * (1 - (2 * snd y - 1)))
    if y ∈ topspace ?A snd y = 1 / 2 for y
  proof -
    have mid1:  $2 * \text{snd } y = (1::\text{real})$ 
      using that(2) by simp
    have mid0:  $2 * \text{snd } y - 1 = (0::\text{real})$ 
      using that(2) by simp
    show ?thesis
      using mid1 mid0 by simp
  qed
  qed
  have hom-const-join: homotopic-pathsin X (λ-. p 0) (joinpathin p (reversepathin p))
  unfolding homotopic-pathsin-def homotopic-with-def
  proof (rule exI[where x = ?h], intro conjI)
    show continuous-map ?A X ?h
      by (rule h-cont)
    show  $\forall x::\text{real}. ?h (0, x) = (\lambda-. p 0) x$ 
      by simp
    show  $\forall x::\text{real}. ?h (1, x) = \text{joinpath}_{in} p (\text{reversepath}_{in} p) x$ 
      by (auto simp: joinpathin-def reversepathin-def)
    show  $\forall t \in \{0..1\}. (\lambda r. r 0 = p 0 \wedge r 1 = p 0) (\lambda x. ?h (t, x))$ 
      by simp
  end

```

**qed**  
**show** *?thesis*  
**by** (rule *homotopic-pathsin-sym*[*OF hom-const-join*])  
**qed**

**lemma** *homotopic-pathsin-linv-const*:  
**assumes** *p*: *pathin X p*  
**shows** *homotopic-pathsin X (joinpathin (reversepathin p) p) (λ-. p 1)*  
**proof** –  
**have** *homotopic-pathsin X (joinpathin (reversepathin p) (reversepathin (reversepathin p))) (λ-. reversepathin p 0)*  
**using** *pathin-reversepathin*[*OF p*] **by** (rule *homotopic-pathsin-rinv-const*)  
**then show** *?thesis*  
**by** (*simp add: reversepathin-def*)  
**qed**

**lemma** *homotopic-loopsin-imp-pathin*:  
**assumes** *homotopic-loopsin X p q*  
**shows** *pathin X p pathin X q*  
**using** *assms unfolding homotopic-loopsin-def pathin-def*  
**by** (*auto dest: homotopic-with-imp-continuous-maps*)

**lemma** *homotopic-loopsin-imp-loop*:  
**assumes** *homotopic-loopsin X p q*  
**shows** *p 1 = p 0 q 1 = q 0*  
**using** *assms unfolding homotopic-loopsin-def*  
**by** (*auto dest: homotopic-with-imp-property*)

**lemma** *homotopic-loopsin-refl* [*simp*]:  
*homotopic-loopsin X p p*  $\longleftrightarrow$  *pathin X p*  $\wedge$  *p 1 = p 0*  
**unfolding** *homotopic-loopsin-def pathin-def*  
**by** (*simp add: homotopic-with-refl*)

**lemma** *homotopic-loopsin-sym*:  
**assumes** *homotopic-loopsin X p q*  
**shows** *homotopic-loopsin X q p*  
**using** *assms unfolding homotopic-loopsin-def*  
**by** (*simp add: homotopic-with-sym*)

**lemma** *homotopic-loopsin-trans*:  
**assumes** *homotopic-loopsin X p q*  
**and** *homotopic-loopsin X q r*  
**shows** *homotopic-loopsin X p r*  
**using** *assms unfolding homotopic-loopsin-def*  
**by** (*blast intro: homotopic-with-trans*)

**lemma** *loopin-space-reversepathin*:  
**assumes** *p*  $\in$  *loopin-space X x*

```

shows reversepathin  $p \in \text{loopin-space } X \ x$ 
proof –
  from assms obtain  $p\text{-in}$  where  $p\text{-in}: \text{pathin } X \ p \ p \ 0 = x \ p \ 1 = x$ 
    by (rule loopin-spaceE)
  have  $\text{path-rev}: \text{pathin } X \ (\text{reversepathin } p)$ 
    using  $p\text{-in}(1)$  by (rule pathin-reversepathin)
  have  $\text{start-rev}: \text{reversepathin } p \ 0 = x$ 
    using  $p\text{-in}(3)$  by (simp add: reversepathin-def)
  have  $\text{finish-rev}: \text{reversepathin } p \ 1 = x$ 
    using  $p\text{-in}(2)$  by (simp add: reversepathin-def)
  show ?thesis
    unfolding loopin-space-def using  $\text{path-rev}$   $\text{start-rev}$   $\text{finish-rev}$  by blast
qed

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* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{loop-space } S \ x =$

$\{p. \text{path } p \wedge \text{path-image } p \subseteq S \wedge \text{pathstart } p = x \wedge \text{pathfinish } p = x\}$

**definition** *loop-class* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{loop-class } S \ x \ p = \{q. q \in \text{loop-space } S \ x \wedge \text{homotopic-paths } S \ q \ p\}$

**definition** *fundamental-group-space* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow ((\text{real} \Rightarrow 'a) \text{ set}) \text{ set}$

**where**

$\text{fundamental-group-space } S \ x = \text{loop-class } S \ x \ \text{'loop-space } S \ x$

**definition** *some-loop* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a)$

**where**

$\text{some-loop } S \ x \ Q = (\text{SOME } p. p \in \text{loop-space } S \ x \wedge Q = \text{loop-class } S \ x \ p)$

**definition** *fundamental-group-one* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

*fundamental-group-one*  $S x = \text{loop-class } S x (\lambda-. x)$

**definition** *fundamental-group-mult* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow$   
 $(\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

*fundamental-group-mult*  $S x A B =$   
*loop-class*  $S x (\text{some-loop } S x A \text{ +++ some-loop } S x B)$

**definition** *fundamental-group-inv* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow$   
 $(\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

*fundamental-group-inv*  $S x A = \text{loop-class } S x (\text{reversepath } (\text{some-loop } S x A))$

**definition** *loop-image* ::

$('a::\text{topological-space} \Rightarrow 'b::\text{topological-space}) \Rightarrow$   
 $(\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'b)$

**where**

*loop-image*  $h p = h \circ p$

**definition** *fundamental-group-map* ::

$'a::\text{topological-space set} \Rightarrow 'a \Rightarrow$   
 $'b::\text{topological-space set} \Rightarrow 'b \Rightarrow$   
 $('a \Rightarrow 'b) \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'b) \text{ set}$

**where**

*fundamental-group-map*  $S x T y h A =$   
*loop-class*  $T y (\text{loop-image } h (\text{some-loop } S x A))$

**lemma** *loop-spaceI*:

**assumes** *path*  $p$   
**and** *path-image*  $p \subseteq S$   
**and** *pathstart*  $p = x$   
**and** *pathfinish*  $p = x$   
**shows**  $p \in \text{loop-space } S x$   
**using** *assms* **unfolding** *loop-space-def* **by** *blast*

**lemma** *constant-loop-in-space*:

**assumes**  $x \in S$   
**shows**  $(\lambda-. x) \in \text{loop-space } S x$

**proof** (*rule* *loop-spaceI*)

**show** *path*  $(\lambda-. x)$   
**by** (*simp* *add*: *path-def*)  
**show** *path-image*  $(\lambda-. x) \subseteq S$   
**using** *assms* **by** (*auto* *simp*: *path-image-def*)  
**show** *pathstart*  $(\lambda-. x) = x$   
**by** (*simp* *add*: *pathstart-def*)  
**show** *pathfinish*  $(\lambda-. x) = x$

by (*simp add: pathfinish-def*)  
qed

**lemma** *loop-class-in-space*:  
assumes  $p \in \text{loop-space } S \ x$   
shows  $\text{loop-class } S \ x \ p \in \text{fundamental-group-space } S \ x$   
using *assms unfolding fundamental-group-space-def* by *blast*

**lemma** *fundamental-group-spaceE*:  
assumes  $Q \in \text{fundamental-group-space } S \ x$   
obtains  $p$  where  $p \in \text{loop-space } S \ x \ Q = \text{loop-class } S \ x \ p$   
using *assms unfolding fundamental-group-space-def* by *blast*

**lemma** *some-loop-spec*:  
assumes  $Q \in \text{fundamental-group-space } S \ x$   
shows  $\text{some-loop } S \ x \ Q \in \text{loop-space } S \ x$   
and  $Q = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ Q)$   
**proof** –  
from *assms* obtain  $p$  where  $p: p \in \text{loop-space } S \ x \ Q = \text{loop-class } S \ x \ p$   
by (*rule fundamental-group-spaceE*)  
have  $ex: \exists p. p \in \text{loop-space } S \ x \ \wedge \ Q = \text{loop-class } S \ x \ p$   
using  $p$  by *blast*  
have  $\text{some-loop } S \ x \ Q \in \text{loop-space } S \ x \ \wedge \ Q = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ Q)$   
unfolding *some-loop-def* by (*rule someI-ex[OF ex]*)  
then show  $\text{some-loop } S \ x \ Q \in \text{loop-space } S \ x$   
and  $Q = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ Q)$   
by *auto*  
qed

**lemma** *loop-class-eqI*:  
assumes  $p: p \in \text{loop-space } S \ x$   
and  $q: q \in \text{loop-space } S \ x$   
and  $pq: \text{homotopic-paths } S \ p \ q$   
shows  $\text{loop-class } S \ x \ p = \text{loop-class } S \ x \ q$   
**proof** (*auto simp: loop-class-def*)  
fix  $r$   
assume  $\text{homotopic-paths } S \ r \ p$   
then show  $\text{homotopic-paths } S \ r \ q$   
using  $pq$  by (*rule homotopic-paths-trans*)  
next  
fix  $r$   
assume  $\text{homotopic-paths } S \ r \ q$   
moreover have  $\text{homotopic-paths } S \ q \ p$   
using  $pq$  by (*rule homotopic-paths-sym*)  
ultimately show  $\text{homotopic-paths } S \ r \ p$   
by (*rule homotopic-paths-trans*)  
qed

**lemma** *loop-class-eq-iff*:

```

assumes  $p: p \in \text{loop-space } S \ x$ 
  and  $q: q \in \text{loop-space } S \ x$ 
shows  $\text{loop-class } S \ x \ p = \text{loop-class } S \ x \ q \longleftrightarrow \text{homotopic-paths } S \ p \ q$ 
proof
  assume  $h: \text{loop-class } S \ x \ p = \text{loop-class } S \ x \ q$ 
  have  $p \in \text{loop-class } S \ x \ p$ 
  proof –
    from  $p$  have  $p\text{-props}: \text{path } p \ \text{path-image } p \subseteq S$ 
    unfolding  $\text{loop-space-def}$  by  $\text{auto}$ 
    have  $\text{homotopic-paths } S \ p \ p$ 
    using  $p\text{-props}$  by  $\text{simp}$ 
    with  $p$  show  $?thesis$ 
    unfolding  $\text{loop-class-def}$  by  $\text{auto}$ 
  qed
  with  $h$  show  $\text{homotopic-paths } S \ p \ q$ 
  unfolding  $\text{loop-class-def}$  by  $\text{blast}$ 
next
  assume  $\text{homotopic-paths } S \ p \ q$ 
  then show  $\text{loop-class } S \ x \ p = \text{loop-class } S \ x \ q$ 
    by  $(\text{rule } \text{loop-class-eqI}[\text{OF } p \ q])$ 
qed

```

```

lemma  $\text{loop-space-join}$ :
  assumes  $p: p \in \text{loop-space } S \ x$ 
    and  $q: q \in \text{loop-space } S \ x$ 
  shows  $p \ +++ \ q \in \text{loop-space } S \ x$ 
proof –
  from  $p$  have  $p\text{-props}$ :
     $\text{path } p \ \text{path-image } p \subseteq S \ \text{pathstart } p = x \ \text{pathfinish } p = x$ 
    unfolding  $\text{loop-space-def}$  by  $\text{auto}$ 
  from  $q$  have  $q\text{-props}$ :
     $\text{path } q \ \text{path-image } q \subseteq S \ \text{pathstart } q = x \ \text{pathfinish } q = x$ 
    unfolding  $\text{loop-space-def}$  by  $\text{auto}$ 
  show  $?thesis$ 
proof  $(\text{rule } \text{loop-spaceI})$ 
    show  $\text{path } (p \ +++ \ q)$ 
      using  $p\text{-props}$   $q\text{-props}$  by  $\text{simp}$ 
    show  $\text{path-image } (p \ +++ \ q) \subseteq S$ 
      using  $p\text{-props}(2)$   $q\text{-props}(2)$  by  $(\text{rule } \text{subset-path-image-join})$ 
    show  $\text{pathstart } (p \ +++ \ q) = x$ 
      using  $p\text{-props}$  by  $\text{simp}$ 
    show  $\text{pathfinish } (p \ +++ \ q) = x$ 
      using  $q\text{-props}$  by  $\text{simp}$ 
  qed
qed

```

```

lemma  $\text{loop-space-reversepath}$ :
  assumes  $p \in \text{loop-space } S \ x$ 
  shows  $\text{reversepath } p \in \text{loop-space } S \ x$ 

```

**proof** –  
**from** *assms* **have** *p-props*:  
 $path\ p\ path\text{-}image\ p \subseteq S\ pathstart\ p = x\ pathfinish\ p = x$   
**unfolding** *loop-space-def* **by** *auto*  
**show** *?thesis*  
**proof** (*rule loop-spaceI*)  
**show**  $path\ (reversepath\ p)$   
**using** *p-props* **by** *simp*  
**show**  $path\text{-}image\ (reversepath\ p) \subseteq S$   
**using** *p-props* **by** *simp*  
**show**  $pathstart\ (reversepath\ p) = x$   
**using** *p-props* **by** *simp*  
**show**  $pathfinish\ (reversepath\ p) = x$   
**using** *p-props* **by** *simp*  
**qed**  
**qed**

**lemma** *loop-space-continuous-image*:  
**assumes** *p*:  $p \in loop\text{-}space\ S\ x$   
**and** *cont-h*: *continuous-on S h*  
**and** *map-h*:  $h \in S \rightarrow T$   
**and** *hx*:  $h\ x = y$   
**shows**  $loop\text{-}image\ h\ p \in loop\text{-}space\ T\ y$   
**proof** –  
**from** *p* **have** *p-props*:  
 $path\ p\ path\text{-}image\ p \subseteq S\ pathstart\ p = x\ pathfinish\ p = x$   
**unfolding** *loop-space-def* **by** *auto*  
**show** *?thesis*  
**unfolding** *loop-image-def*  
**proof** (*rule loop-spaceI*)  
**show**  $path\ (h \circ p)$   
**using** *p-props(1)* *continuous-on-subset[OF cont-h p-props(2)]*  
**by** (*rule Path-Connected.path-continuous-image*)  
**show**  $path\text{-}image\ (h \circ p) \subseteq T$   
**using** *p-props(2)* *map-h*  
**by** (*auto simp: path-image-def*)  
**show**  $pathstart\ (h \circ p) = y$   
**using** *p-props(3)* *hx* **by** (*simp add: pathstart-compose*)  
**show**  $pathfinish\ (h \circ p) = y$   
**using** *p-props(4)* *hx* **by** (*simp add: pathfinish-compose*)  
**qed**  
**qed**

**lemma** *loop-image-join*:  
 $loop\text{-}image\ h\ (p\ +++\ q) = loop\text{-}image\ h\ p\ +++\ loop\text{-}image\ h\ q$   
**unfolding** *loop-image-def*  
**by** (*rule ext*) (*simp add: joinpaths-def*)

**lemma** *loop-image-reversepath*:

$loop\text{-}image\ h\ (reversepath\ p) = reversepath\ (loop\text{-}image\ h\ p)$   
**unfolding**  $loop\text{-}image\text{-}def\ reversepath\text{-}def$   
**by**  $(rule\ ext)\ simp$

**lemma**  $loop\text{-}class\text{-}join\text{-}eqI$ :  
**assumes**  $p: p \in loop\text{-}space\ S\ x$   
**and**  $p': p' \in loop\text{-}space\ S\ x$   
**and**  $q: q \in loop\text{-}space\ S\ x$   
**and**  $q': q' \in loop\text{-}space\ S\ x$   
**and**  $pp': homotopic\text{-}paths\ S\ p\ p'$   
**and**  $qq': homotopic\text{-}paths\ S\ q\ q'$   
**shows**  $loop\text{-}class\ S\ x\ (p\ +++\ q) = loop\text{-}class\ S\ x\ (p'\ +++\ q')$   
**proof**  $(rule\ loop\text{-}class\text{-}eqI)$   
**show**  $p\ +++\ q \in loop\text{-}space\ S\ x$   
**using**  $p\ q$  **by**  $(rule\ loop\text{-}space\text{-}join)$   
**show**  $p'\ +++\ q' \in loop\text{-}space\ S\ x$   
**using**  $p'\ q'$  **by**  $(rule\ loop\text{-}space\text{-}join)$   
**from**  $p$  **have**  $px: pathfinish\ p = x$   
**unfolding**  $loop\text{-}space\text{-}def$  **by**  $auto$   
**from**  $q$  **have**  $qx: pathstart\ q = x$   
**unfolding**  $loop\text{-}space\text{-}def$  **by**  $auto$   
**show**  $homotopic\text{-}paths\ S\ (p\ +++\ q)\ (p'\ +++\ q')$   
**proof**  $(rule\ homotopic\text{-}paths\text{-}join)$   
**show**  $homotopic\text{-}paths\ S\ p\ p'$   
**by**  $(rule\ pp')$   
**show**  $homotopic\text{-}paths\ S\ q\ q'$   
**by**  $(rule\ qq')$   
**show**  $pathfinish\ p = pathstart\ q$   
**using**  $px\ qx$  **by**  $simp$   
**qed**  
**qed**

**lemma**  $loop\text{-}class\text{-}reverse\text{-}eqI$ :  
**assumes**  $p: p \in loop\text{-}space\ S\ x$   
**and**  $q: q \in loop\text{-}space\ S\ x$   
**and**  $pq: homotopic\text{-}paths\ S\ p\ q$   
**shows**  $loop\text{-}class\ S\ x\ (reversepath\ p) = loop\text{-}class\ S\ x\ (reversepath\ q)$   
**proof**  $(rule\ loop\text{-}class\text{-}eqI)$   
**show**  $reversepath\ p \in loop\text{-}space\ S\ x$   
**using**  $p$  **by**  $(rule\ loop\text{-}space\text{-}reversepath)$   
**show**  $reversepath\ q \in loop\text{-}space\ S\ x$   
**using**  $q$  **by**  $(rule\ loop\text{-}space\text{-}reversepath)$   
**from**  $pq$  **show**  $homotopic\text{-}paths\ S\ (reversepath\ p)\ (reversepath\ q)$   
**by**  $(rule\ homotopic\text{-}paths\text{-}reversepath\text{-}D)$   
**qed**

**lemma**  $homotopic\text{-}paths\text{-}rid\text{-}const$ :  
**assumes**  $path\ p$   
**and**  $path\text{-}image\ p \subseteq S$

```

shows homotopic-paths  $S$  ( $p$  +++ ( $\lambda$ -. pathfinish  $p$ ))  $p$ 
proof –
  let  $?f = \lambda t::real. \text{if } t \leq 1/2 \text{ then } 2 *_{\mathbb{R}} t \text{ else } 1$ 
  have contf: continuous-on  $\{0..1\}$   $?f$ 
    unfolding split-01
    by (rule continuous-on-cases continuous-intros | auto)+
  have  $f01: ?f \in \{0..1\} \rightarrow \{0..1\}$ 
    by auto
  have homotopic-paths  $S$  ( $p$  +++ ( $\lambda$ -. pathfinish  $p$ ))
  proof (rule homotopic-paths-reparametrize[where  $f = ?f$ ])
    show path  $p$ 
      by (rule assms(1))
    show path-image  $p \subseteq S$ 
      by (rule assms(2))
    show continuous-on  $\{0..1\}$   $?f$ 
      by (rule contf)
    show  $?f \in \{0..1\} \rightarrow \{0..1\}$ 
      by (rule f01)
    show  $?f\ 0 = 0$   $?f\ 1 = 1$ 
      by simp-all
    show ( $p$  +++ ( $\lambda$ -. pathfinish  $p$ ))  $t = p$  ( $?f\ t$ ) if  $t \in \{0..1\}$  for  $t$ 
      using that by (auto simp: joinpaths-def pathfinish-def)
  qed
  then show ?thesis
    by (rule homotopic-paths-sym)
qed

```

```

lemma homotopic-paths-lid-const:
  assumes path  $p$ 
  and path-image  $p \subseteq S$ 
  shows homotopic-paths  $S$  (( $\lambda$ -. pathstart  $p$ ) +++  $p$ )  $p$ 
proof –
  let  $?f = \lambda t::real. \text{if } t \leq 1/2 \text{ then } 0 \text{ else } 2 *_{\mathbb{R}} t - 1$ 
  have contf: continuous-on  $\{0..1\}$   $?f$ 
    unfolding split-01
    by (rule continuous-on-cases continuous-intros | auto)+
  have  $f01: ?f \in \{0..1\} \rightarrow \{0..1\}$ 
    by auto
  have homotopic-paths  $S$  ( $p$  (( $\lambda$ -. pathstart  $p$ ) +++  $p$ ))
  proof (rule homotopic-paths-reparametrize[where  $f = ?f$ ])
    show path  $p$ 
      by (rule assms(1))
    show path-image  $p \subseteq S$ 
      by (rule assms(2))
    show continuous-on  $\{0..1\}$   $?f$ 
      by (rule contf)
    show  $?f \in \{0..1\} \rightarrow \{0..1\}$ 
      by (rule f01)
    show  $?f\ 0 = 0$   $?f\ 1 = 1$ 

```

```

    by simp-all
    show ((λ-. pathstart p) +++ p) t = p (?f t) if t ∈ {0..1} for t
    using that by (auto simp: joinpaths-def pathstart-def)
qed
then show ?thesis
    by (rule homotopic-paths-sym)
qed

lemma homotopic-paths-rinv-const:
  assumes path p
  and path-image p ⊆ S
  shows homotopic-paths S (p +++ reversepath p) (λ-. pathstart p)
proof -
  let ?A = {0..1} × {0..1}
  let ?h =
    λy. if snd y ≤ 1/2
      then p (fst y * (2 * snd y))
      else p (fst y * (1 - (2 * snd y - 1)))
  define H where H = ?h
  have p-cont: continuous-on {0..1} p
    using assms by (auto simp: path-def)
  have h-cont: continuous-on ?A H
    unfolding H-def
  proof (rule continuous-on-cases-le)
    show continuous-on {x ∈ ?A. snd x ≤ 1/2} (λt. p (fst t * (2 * snd t)))
      continuous-on {x ∈ ?A. 1/2 ≤ snd x} (λt. p (fst t * (1 - (2 * snd t -
1))))
      continuous-on ?A snd
    by (intro continuous-on-compose2 [OF p-cont] continuous-intros; auto simp:
mult-le-one)+
  qed (auto simp: algebra-simps)
  have h-subset: H ∈ ?A → S
  proof
    fix y :: real × real
    assume y: y ∈ ?A
    with assms show H y ∈ S
      unfolding H-def
      by (force simp: path-image-def mult-le-one)
  qed
  have h0: ∀ x ∈ {0..1}. H (0, x) = (λ-. pathstart p) x
    unfolding H-def by (simp add: pathstart-def)
  have h1: ∀ x ∈ {0..1}. H (1, x) = (p +++ reversepath p) x
    unfolding H-def by (auto simp: joinpaths-def reversepath-def)
  have hends:
    ∀ t ∈ {0..1}. pathstart (H ∘ Pair t) = pathstart (λ-. pathstart p) ∧
      pathfinish (H ∘ Pair t) = pathfinish (λ-. pathstart p)
  proof
    fix t :: real
    assume t ∈ {0..1}

```

```

show  $\text{pathstart } (H \circ \text{Pair } t) = \text{pathstart } (\lambda\cdot. \text{pathstart } p) \wedge$ 
 $\text{pathfinish } (H \circ \text{Pair } t) = \text{pathfinish } (\lambda\cdot. \text{pathstart } p)$ 
proof
  show  $\text{pathstart } (H \circ \text{Pair } t) = \text{pathstart } (\lambda\cdot. \text{pathstart } p)$ 
    unfolding  $H\text{-def}$  by (simp add: pathstart-def)
  show  $\text{pathfinish } (H \circ \text{Pair } t) = \text{pathfinish } (\lambda\cdot. \text{pathstart } p)$ 
    unfolding  $H\text{-def}$  by (simp add: pathstart-def pathfinish-def)
qed
qed
have hom-const:
  homotopic-paths  $S$  ( $\lambda\cdot. \text{pathstart } p$ ) ( $p \text{ +++ reversepath } p$ )
apply (rule iffD2[OF homotopic-paths])
apply (rule-tac
   $x = (\lambda y. \text{if } \text{snd } y \leq 1/2$ 
     $\text{then } p (\text{fst } y * (2 * \text{snd } y))$ 
     $\text{else } p (\text{fst } y * (1 - (2 * \text{snd } y - 1))))$ 
  in exI)
using h-cont h-subset h0 h1 hends
unfolding  $H\text{-def}$ 
apply blast
done
show ?thesis
  using hom-const by (rule homotopic-paths-sym)
qed

lemma homotopic-paths-linv-const:
assumes path  $p$ 
  and path-image  $p \subseteq S$ 
shows homotopic-paths  $S$  (reversepath  $p \text{ +++ } p$ ) ( $\lambda\cdot. \text{pathfinish } p$ )
using homotopic-paths-rinv-const [of reversepath p S] assms
by simp

lemma fundamental-group-one-in-space:
assumes  $x \in S$ 
shows fundamental-group-one  $S$   $x \in \text{fundamental-group-space } S$   $x$ 
unfolding fundamental-group-one-def
using constant-loop-in-space[OF assms] by (rule loop-class-in-space)

lemma fundamental-group-mult-eqI:
assumes A-in:  $A \in \text{fundamental-group-space } S$   $x$ 
  and B-in:  $B \in \text{fundamental-group-space } S$   $x$ 
  and  $p \in \text{loop-space } S$   $x$ 
  and  $q \in \text{loop-space } S$   $x$ 
  and  $A = \text{loop-class } S$   $x$   $p$ 
  and  $B = \text{loop-class } S$   $x$   $q$ 
shows fundamental-group-mult  $S$   $x$   $A$   $B = \text{loop-class } S$   $x$  ( $p \text{ +++ } q$ )
proof –
  have repA: some-loop  $S$   $x$   $A \in \text{loop-space } S$   $x$   $A = \text{loop-class } S$   $x$  (some-loop  $S$   $x$ 
 $A$ )

```

```

    using some-loop-spec[OF A-in] by auto
  have repB: some-loop S x B ∈ loop-space S x B = loop-class S x (some-loop S x
  B)
    using some-loop-spec[OF B-in] by auto
  have homoA': homotopic-paths S p (some-loop S x A)
    using repA p A by (simp add: loop-class-eq-iff)
  have homoA: homotopic-paths S (some-loop S x A) p
    using homoA' by (rule homotopic-paths-sym)
  have homoB': homotopic-paths S q (some-loop S x B)
    using repB q B by (simp add: loop-class-eq-iff)
  have homoB: homotopic-paths S (some-loop S x B) q
    using homoB' by (rule homotopic-paths-sym)
  have loop-class S x (some-loop S x A +++ some-loop S x B) = loop-class S x (p
  +++ q)
    by (rule loop-class-join-eqI[OF repA(1) p repB(1) q homoA homoB])
  then show ?thesis
    unfolding fundamental-group-mult-def .
qed

```

```

lemma fundamental-group-mult-in-space:
  assumes A ∈ fundamental-group-space S x
    and B ∈ fundamental-group-space S x
  shows fundamental-group-mult S x A B ∈ fundamental-group-space S x
proof -
  have repA: some-loop S x A ∈ loop-space S x
    using some-loop-spec[OF assms(1)] by auto
  have repB: some-loop S x B ∈ loop-space S x
    using some-loop-spec[OF assms(2)] by auto
  show ?thesis
    unfolding fundamental-group-mult-def
    using loop-space-join[OF repA repB] by (rule loop-class-in-space)
qed

```

```

lemma fundamental-group-inv-eqI:
  assumes A-in: A ∈ fundamental-group-space S x
    and p: p ∈ 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 -
  have repA: some-loop S x A ∈ loop-space S x A = loop-class S x (some-loop S x
  A)
    using some-loop-spec[OF A-in] by auto
  have homoA': homotopic-paths S p (some-loop S x A)
    using repA p A by (simp add: loop-class-eq-iff)
  have homoA: homotopic-paths S (some-loop S x A) p
    using homoA' by (rule homotopic-paths-sym)
  have loop-class S x (reversepath (some-loop S x A)) = loop-class S x (reversepath
  p)
    by (rule loop-class-reverse-eqI[OF repA(1) p homoA])

```

**then show** *?thesis*  
**unfolding** *fundamental-group-inv-def* .  
**qed**

**lemma** *fundamental-group-inv-in-space*:  
**assumes**  $A \in \text{fundamental-group-space } S \ x$   
**shows** *fundamental-group-inv*  $S \ x \ A \in \text{fundamental-group-space } S \ x$   
**proof** –  
**have** *repA*: *some-loop*  $S \ x \ A \in \text{loop-space } S \ x$   
**using** *some-loop-spec*[*OF assms*] **by** *auto*  
**show** *?thesis*  
**unfolding** *fundamental-group-inv-def*  
**using** *loop-space-reversepath*[*OF repA*] **by** (*rule loop-class-in-space*)  
**qed**

**lemma** *fundamental-group-map-eqI*:  
**assumes** *A-in*:  $A \in \text{fundamental-group-space } S \ x$   
**and**  $p: p \in \text{loop-space } S \ x$   
**and**  $A: A = \text{loop-class } S \ x \ p$   
**and** *cont-h*: *continuous-on*  $S \ h$   
**and** *map-h*:  $h \in S \rightarrow T$   
**and**  $hx: h \ x = y$   
**shows** *fundamental-group-map*  $S \ x \ T \ y \ h \ A = \text{loop-class } T \ y \ (\text{loop-image } h \ p)$   
**proof** –  
**have** *repA*: *some-loop*  $S \ x \ A \in \text{loop-space } S \ x \ A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**using** *some-loop-spec*[*OF A-in*] **by** *auto*  
**have** *homo'*:  
*homotopic-paths*  $S \ p \ (\text{some-loop } S \ x \ A)$   
**using** *repA(1)*  $p \ \text{repA}(2) \ A$   
**by** (*simp add: loop-class-eq-iff*)  
**have** *homo*:  
*homotopic-paths*  $S \ (\text{some-loop } S \ x \ A) \ p$   
**using** *homo'* **by** (*rule homotopic-paths-sym*)  
**have** *map-rep*: *loop-image*  $h \ (\text{some-loop } S \ x \ A) \in \text{loop-space } T \ y$   
**by** (*rule loop-space-continuous-image*[*OF repA(1) cont-h map-h hx*])  
**have** *map-p*: *loop-image*  $h \ p \in \text{loop-space } T \ y$   
**by** (*rule loop-space-continuous-image*[*OF p cont-h map-h hx*])  
**have** *map-homo*:  
*homotopic-paths*  $T \ (\text{loop-image } h \ (\text{some-loop } S \ x \ A)) \ (\text{loop-image } h \ p)$   
**unfolding** *loop-image-def*  
**by** (*rule homotopic-paths-continuous-image*[*OF homo cont-h map-h*])  
**have**  
*loop-class*  $T \ y \ (\text{loop-image } h \ (\text{some-loop } S \ x \ A)) =$   
*loop-class*  $T \ y \ (\text{loop-image } h \ p)$   
**by** (*rule loop-class-eqI*[*OF map-rep map-p map-homo*])  
**then show** *?thesis*  
**unfolding** *fundamental-group-map-def* .  
**qed**

**lemma** *fundamental-group-map-in-space*:  
**assumes** *A-in*:  $A \in \text{fundamental-group-space } S \ x$   
**and** *cont-h*: *continuous-on*  $S \ h$   
**and** *map-h*:  $h \in S \rightarrow T$   
**and** *hx*:  $h \ x = y$   
**shows** *fundamental-group-map*  $S \ x \ T \ y \ h \ A \in \text{fundamental-group-space } T \ y$   
**proof** –  
**have** *repA*: *some-loop*  $S \ x \ A \in \text{loop-space } S \ x$   
**using** *some-loop-spec*[*OF A-in*] **by** *auto*  
**have** *loop-image*  $h \ (\text{some-loop } S \ x \ A) \in \text{loop-space } T \ y$   
**by** (*rule loop-space-continuous-image*[*OF repA cont-h map-h hx*])  
**then show** *?thesis*  
**unfolding** *fundamental-group-map-def* **by** (*rule loop-class-in-space*)  
**qed**

**lemma** *fundamental-group-map-mult*:  
**assumes** *A-in*:  $A \in \text{fundamental-group-space } S \ x$   
**and** *B-in*:  $B \in \text{fundamental-group-space } S \ x$   
**and** *cont-h*: *continuous-on*  $S \ h$   
**and** *map-h*:  $h \in S \rightarrow T$   
**and** *hx*:  $h \ x = y$   
**shows**  
*fundamental-group-map*  $S \ x \ T \ y \ h \ (\text{fundamental-group-mult } S \ x \ A \ B) =$   
*fundamental-group-mult*  $T \ y$   
(*fundamental-group-map*  $S \ x \ T \ y \ h \ A$ )  
(*fundamental-group-map*  $S \ x \ T \ y \ h \ B$ )  
**proof** –  
**have** *repA*: *some-loop*  $S \ x \ A \in \text{loop-space } S \ x \ A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**using** *some-loop-spec*[*OF A-in*] **by** *auto*  
**have** *repB*: *some-loop*  $S \ x \ B \in \text{loop-space } S \ x \ B = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ B)$   
**using** *some-loop-spec*[*OF B-in*] **by** *auto*  
**have** *AB-in*: *fundamental-group-mult*  $S \ x \ A \ B \in \text{fundamental-group-space } S \ x$   
**by** (*rule fundamental-group-mult-in-space*[*OF A-in B-in*])  
**have** *AB-eq*:  
*fundamental-group-mult*  $S \ x \ A \ B =$   
*loop-class*  $S \ x \ (\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B)$   
**by** (*rule fundamental-group-mult-eqI*[*OF A-in B-in repA(1) repB(1) repA(2) repB(2)*])  
**have** *map-A-in*:  
*fundamental-group-map*  $S \ x \ T \ y \ h \ A \in \text{fundamental-group-space } T \ y$   
**by** (*rule fundamental-group-map-in-space*[*OF A-in cont-h map-h hx*])  
**have** *map-B-in*:  
*fundamental-group-map*  $S \ x \ T \ y \ h \ B \in \text{fundamental-group-space } T \ y$   
**by** (*rule fundamental-group-map-in-space*[*OF B-in cont-h map-h hx*])  
**have** *map-A-eq*:  
*fundamental-group-map*  $S \ x \ T \ y \ h \ A =$

$loop\text{-}class\ T\ y\ (loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A))$   
**by** (rule  $fundamental\text{-}group\text{-}map\text{-}eqI[OF\ A\text{-}in\ repA(1)\ repA(2)\ cont\text{-}h\ map\text{-}h\ hx]$ )  
**have**  $map\text{-}B\text{-}eq$ :  
 $fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ B =$   
 $loop\text{-}class\ T\ y\ (loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ B))$   
**by** (rule  $fundamental\text{-}group\text{-}map\text{-}eqI[OF\ B\text{-}in\ repB(1)\ repB(2)\ cont\text{-}h\ map\text{-}h\ hx]$ )  
**have**  $left\text{-}eq$ :  
 $fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ (fundamental\text{-}group\text{-}mult\ S\ x\ A\ B) =$   
 $loop\text{-}class\ T\ y\ (loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A\ +++\ some\text{-}loop\ S\ x\ B))$   
**by** (rule  $fundamental\text{-}group\text{-}map\text{-}eqI[OF\ AB\text{-}in\ loop\text{-}space\text{-}join[OF\ repA(1)\ repB(1)]\ AB\text{-}eq\ cont\text{-}h\ map\text{-}h\ hx]$ )  
**have**  $map\text{-}loop\text{-}A$ :  $loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A) \in loop\text{-}space\ T\ y$   
**by** (rule  $loop\text{-}space\text{-}continuous\text{-}image[OF\ repA(1)\ cont\text{-}h\ map\text{-}h\ hx]$ )  
**have**  $map\text{-}loop\text{-}B$ :  $loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ B) \in loop\text{-}space\ T\ y$   
**by** (rule  $loop\text{-}space\text{-}continuous\text{-}image[OF\ repB(1)\ cont\text{-}h\ map\text{-}h\ hx]$ )  
**have**  $right\text{-}eq$ :  
 $fundamental\text{-}group\text{-}mult\ T\ y$   
 $(fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ A)$   
 $(fundamental\text{-}group\text{-}map\ S\ x\ T\ y\ h\ B) =$   
 $loop\text{-}class\ T\ y$   
 $(loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A)\ +++\ loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ B))$   
**by** (rule  $fundamental\text{-}group\text{-}mult\text{-}eqI[OF\ map\text{-}A\text{-}in\ map\text{-}B\text{-}in\ map\text{-}loop\text{-}A\ map\text{-}loop\text{-}B\ map\text{-}A\text{-}eq\ map\text{-}B\text{-}eq]$ )  
**have**  
 $loop\text{-}class\ T\ y\ (loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A\ +++\ some\text{-}loop\ S\ x\ B)) =$   
 $loop\text{-}class\ T\ y\ (loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ A)\ +++\ loop\text{-}image\ h\ (some\text{-}loop\ S\ x\ B))$   
**by** (rule  $arg\text{-}cong[where\ f = loop\text{-}class\ T\ y]$ ,  $simp\ add: loop\text{-}image\text{-}join$ )  
**then show**  $?thesis$   
**using**  $left\text{-}eq\ right\text{-}eq$  **by**  $simp$   
**qed**

**lemma**  $fundamental\text{-}group\text{-}map\text{-}id$ :

**assumes**  $A\text{-}in$ :  $A \in fundamental\text{-}group\text{-}space\ S\ x$

**shows**  $fundamental\text{-}group\text{-}map\ S\ x\ S\ x\ id\ A = A$

**proof** –

**have**  $repA$ :  $some\text{-}loop\ S\ x\ A \in loop\text{-}space\ S\ x\ A = loop\text{-}class\ S\ x\ (some\text{-}loop\ S\ x\ A)$

**using**  $some\text{-}loop\text{-}spec[OF\ A\text{-}in]$  **by**  $auto$

**have**  $fundamental\text{-}group\text{-}map\ S\ x\ S\ x\ id\ A =$

$loop\text{-}class\ S\ x\ (loop\text{-}image\ id\ (some\text{-}loop\ S\ x\ A))$

**by** (rule  $fundamental\text{-}group\text{-}map\text{-}eqI[OF\ A\text{-}in\ repA(1)\ repA(2)]$ )  $simp\text{-}all$

**then show**  $?thesis$

**using**  $repA(2)$  **by** ( $simp\ add: loop\text{-}image\text{-}def$ )

**qed**

**lemma**  $fundamental\text{-}group\text{-}map\text{-}one$ :

**assumes**  $x\text{-in}: x \in S$   
**and**  $\text{cont-h}: \text{continuous-on } S \ h$   
**and**  $\text{map-h}: h \in S \rightarrow T$   
**and**  $\text{hx}: h \ x = y$   
**shows**  
 $\text{fundamental-group-map } S \ x \ T \ y \ h \ (\text{fundamental-group-one } S \ x) =$   
 $\text{fundamental-group-one } T \ y$   
**proof** –  
**have**  $\text{one-in}: \text{fundamental-group-one } S \ x \in \text{fundamental-group-space } S \ x$   
**by** (rule  $\text{fundamental-group-one-in-space}[OF \ x\text{-in}]$ )  
**have**  $\text{const-in}: (\lambda\cdot. x) \in \text{loop-space } S \ x$   
**by** (rule  $\text{constant-loop-in-space}[OF \ x\text{-in}]$ )  
**have**  $\text{fundamental-group-map } S \ x \ T \ y \ h \ (\text{fundamental-group-one } S \ x) =$   
 $\text{loop-class } T \ y \ (\text{loop-image } h \ (\lambda\cdot. x))$   
**by** (rule  $\text{fundamental-group-map-eqI}[OF \ \text{one-in } \ \text{const-in}]$ ) (simp-all add:  $\text{fundamental-group-one-def } \text{cont-h } \text{map-h } \text{hx}$ )  
**then show**  $?thesis$   
**by** (simp add:  $\text{fundamental-group-one-def } \text{loop-image-def } \text{hx}$ )  
**qed**

**lemma**  $\text{fundamental-group-map-compose}$ :

**assumes**  $A\text{-in}: A \in \text{fundamental-group-space } S \ x$   
**and**  $\text{cont-h}: \text{continuous-on } S \ h$   
**and**  $\text{map-h}: h \in S \rightarrow T$   
**and**  $\text{hx}: h \ x = y$   
**and**  $\text{cont-k}: \text{continuous-on } T \ k$   
**and**  $\text{map-k}: k \in T \rightarrow U$   
**and**  $\text{ky}: k \ y = z$   
**shows**  
 $\text{fundamental-group-map } T \ y \ U \ z \ k \ (\text{fundamental-group-map } S \ x \ T \ y \ h \ A) =$   
 $\text{fundamental-group-map } S \ x \ U \ z \ (k \circ h) \ A$   
**proof** –  
**have**  $\text{repA}: \text{some-loop } S \ x \ A \in \text{loop-space } S \ x \ A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**using**  $\text{some-loop-spec}[OF \ A\text{-in}]$  **by** *auto*  
**have**  $\text{map-A-in}: \text{fundamental-group-map } S \ x \ T \ y \ h \ A \in \text{fundamental-group-space } T \ y$   
**by** (rule  $\text{fundamental-group-map-in-space}[OF \ A\text{-in } \ \text{cont-h } \ \text{map-h } \ \text{hx}]$ )  
**have**  $\text{map-A-eq}: \text{fundamental-group-map } S \ x \ T \ y \ h \ A =$   
 $\text{loop-class } T \ y \ (\text{loop-image } h \ (\text{some-loop } S \ x \ A))$   
**by** (rule  $\text{fundamental-group-map-eqI}[OF \ A\text{-in } \ \text{repA}(1) \ \text{repA}(2) \ \text{cont-h } \ \text{map-h } \ \text{hx}]$ )  
**have**  $\text{left-eq}: \text{fundamental-group-map } T \ y \ U \ z \ k \ (\text{fundamental-group-map } S \ x \ T \ y \ h \ A) =$   
 $\text{loop-class } U \ z \ (\text{loop-image } k \ (\text{loop-image } h \ (\text{some-loop } S \ x \ A)))$   
**by** (rule  $\text{fundamental-group-map-eqI}[OF \ \text{map-A-in } \ \text{loop-space-continuous-image}[OF \ \text{repA}(1) \ \text{cont-h } \ \text{map-h } \ \text{hx}] \ \text{map-A-eq } \ \text{cont-k } \ \text{map-k } \ \text{ky}]$ )

```

have comp-cont: continuous-on  $S$   $(k \circ h)$ 
proof –
  have hs:  $h \text{ ' } S \subseteq T$ 
    using map-h by auto
  have hk: continuous-on  $(h \text{ ' } S)$   $k$ 
    using cont-k hs by (rule continuous-on-subset)
  from cont-h hk show ?thesis
    by (rule continuous-on-compose)
qed
have comp-map:  $(k \circ h) \in S \rightarrow U$ 
  using map-h map-k by auto
have right-eq:
  fundamental-group-map  $S$   $x$   $U$   $z$   $(k \circ h)$   $A =$ 
  loop-class  $U$   $z$  (loop-image  $(k \circ h)$  (some-loop  $S$   $x$   $A$ ))
  by (rule fundamental-group-map-eqI[OF A-in repA(1) repA(2) comp-cont
comp-map])
  (simp add: hx ky)
have
  loop-class  $U$   $z$  (loop-image  $k$  (loop-image  $h$  (some-loop  $S$   $x$   $A$ ))) =
  loop-class  $U$   $z$  (loop-image  $(k \circ h)$  (some-loop  $S$   $x$   $A$ ))
  by (rule arg-cong[where  $f = \text{loop-class } U z$ ]) (simp add: loop-image-def o-assoc)
then show ?thesis
  using left-eq right-eq by simp
qed

lemma fundamental-group-mult-one-left:
  assumes  $x \in S$ 
  and  $A \in \text{fundamental-group-space } S$   $x$ 
  shows fundamental-group-mult  $S$   $x$  (fundamental-group-one  $S$   $x$ )  $A = A$ 
proof –
  have const:  $(\lambda \cdot. x) \in \text{loop-space } S$   $x$ 
    using constant-loop-in-space[OF assms(1)] .
  have repA: some-loop  $S$   $x$   $A \in \text{loop-space } S$   $x$   $A = \text{loop-class } S$   $x$  (some-loop  $S$   $x$ 
 $A$ )
    using some-loop-spec[OF assms(2)] by auto
  have loopA:
    path (some-loop  $S$   $x$   $A$ )
    path-image (some-loop  $S$   $x$   $A$ )  $\subseteq S$ 
    pathstart (some-loop  $S$   $x$   $A$ ) =  $x$ 
    pathfinish (some-loop  $S$   $x$   $A$ ) =  $x$ 
    using repA(1) unfolding loop-space-def by auto
  have mult-eq:
    fundamental-group-mult  $S$   $x$  (fundamental-group-one  $S$   $x$ )  $A =$ 
    loop-class  $S$   $x$  (( $\lambda \cdot. x$ ) +++ some-loop  $S$   $x$   $A$ )
proof (rule fundamental-group-mult-eqI)
  show fundamental-group-one  $S$   $x \in \text{fundamental-group-space } S$   $x$ 
    by (rule fundamental-group-one-in-space[OF assms(1)])
  show  $A \in \text{fundamental-group-space } S$   $x$ 
    by (rule assms(2))

```

```

show  $(\lambda-. x) \in \text{loop-space } S x$ 
  by (rule const)
show  $\text{some-loop } S x A \in \text{loop-space } S x$ 
  by (rule repA(1))
show  $\text{fundamental-group-one } S x = \text{loop-class } S x (\lambda-. x)$ 
  by (simp add: fundamental-group-one-def)
show  $A = \text{loop-class } S x (\text{some-loop } S x A)$ 
  by (rule repA(2))
qed
have join-loop:  $(\lambda-. x) \text{+++ some-loop } S x A \in \text{loop-space } S x$ 
  using const repA(1) by (rule loop-space-join)
have hom:  $\text{homotopic-paths } S ((\lambda-. x) \text{+++ some-loop } S x A) (\text{some-loop } S x A)$ 
  using homotopic-paths-lid-const[OF loopA(1) loopA(2)] loopA(3) by simp
have class-eq:
   $\text{loop-class } S x ((\lambda-. x) \text{+++ some-loop } S x A) = \text{loop-class } S x (\text{some-loop } S x A)$ 
  by (rule loop-class-eqI[OF join-loop repA(1) hom])
show ?thesis
  using mult-eq class-eq repA(2) by simp
qed

```

**lemma** *fundamental-group-mult-one-right*:

```

assumes  $x \in S$ 
  and  $A \in \text{fundamental-group-space } S x$ 
shows  $\text{fundamental-group-mult } S x A (\text{fundamental-group-one } S x) = A$ 
proof –
have const:  $(\lambda-. x) \in \text{loop-space } S x$ 
  using constant-loop-in-space[OF assms(1)] .
have repA:  $\text{some-loop } S x A \in \text{loop-space } S x A = \text{loop-class } S x (\text{some-loop } S x A)$ 
  using some-loop-spec[OF assms(2)] by auto
have loopA:
  path (some-loop  $S x A$ )
  path-image (some-loop  $S x A$ )  $\subseteq S$ 
  pathstart (some-loop  $S x A$ ) =  $x$ 
  pathfinish (some-loop  $S x A$ ) =  $x$ 
  using repA(1) unfolding loop-space-def by auto
have mult-eq:
   $\text{fundamental-group-mult } S x A (\text{fundamental-group-one } S x) =$ 
   $\text{loop-class } S x (\text{some-loop } S x A \text{+++ } (\lambda-. x))$ 
proof (rule fundamental-group-mult-eqI)
show  $A \in \text{fundamental-group-space } S x$ 
  by (rule assms(2))
show  $\text{fundamental-group-one } S x \in \text{fundamental-group-space } S x$ 
  by (rule fundamental-group-one-in-space[OF assms(1)])
show  $\text{some-loop } S x A \in \text{loop-space } S x$ 
  by (rule repA(1))
show  $(\lambda-. x) \in \text{loop-space } S x$ 
  by (rule const)

```

**show**  $A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**by**  $(\text{rule repA}(2))$   
**show**  $\text{fundamental-group-one } S \ x = \text{loop-class } S \ x \ (\lambda-. \ x)$   
**by**  $(\text{simp add: fundamental-group-one-def})$   
**qed**  
**have**  $\text{join-loop: some-loop } S \ x \ A \ +++ \ (\lambda-. \ x) \in \text{loop-space } S \ x$   
**using**  $\text{repA}(1) \ \text{const}$  **by**  $(\text{rule loop-space-join})$   
**have**  $\text{hom: homotopic-paths } S \ (\text{some-loop } S \ x \ A \ +++ \ (\lambda-. \ x)) \ (\text{some-loop } S \ x \ A)$   
**using**  $\text{homotopic-paths-rid-const}[OF \ \text{loopA}(1) \ \text{loopA}(2)] \ \text{loopA}(4)$  **by**  $\text{simp}$   
**have**  $\text{class-eq:}$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ A \ +++ \ (\lambda-. \ x)) = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**by**  $(\text{rule loop-class-eqI}[OF \ \text{join-loop} \ \text{repA}(1) \ \text{hom}])$   
**show**  $?thesis$   
**using**  $\text{mult-eq class-eq repA}(2)$  **by**  $\text{simp}$   
**qed**

**lemma**  $\text{fundamental-group-mult-assoc:}$

**assumes**  $A\text{-in: } A \in \text{fundamental-group-space } S \ x$   
**and**  $B\text{-in: } B \in \text{fundamental-group-space } S \ x$   
**and**  $C\text{-in: } C \in \text{fundamental-group-space } S \ x$   
**shows**  $\text{fundamental-group-mult } S \ x \ A \ (\text{fundamental-group-mult } S \ x \ B \ C) =$   
 $\text{fundamental-group-mult } S \ x \ (\text{fundamental-group-mult } S \ x \ A \ B) \ C$   
**proof** –  
**have**  $\text{repA: some-loop } S \ x \ A \in \text{loop-space } S \ x \ A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**using**  $\text{some-loop-spec}[OF \ A\text{-in}]$  **by**  $\text{auto}$   
**have**  $\text{repB: some-loop } S \ x \ B \in \text{loop-space } S \ x \ B = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ B)$   
**using**  $\text{some-loop-spec}[OF \ B\text{-in}]$  **by**  $\text{auto}$   
**have**  $\text{repC: some-loop } S \ x \ C \in \text{loop-space } S \ x \ C = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ C)$   
**using**  $\text{some-loop-spec}[OF \ C\text{-in}]$  **by**  $\text{auto}$   
**have**  $BC\text{-eq:}$   
 $\text{fundamental-group-mult } S \ x \ B \ C =$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C)$   
**proof**  $(\text{rule fundamental-group-mult-eqI})$   
**show**  $B \in \text{fundamental-group-space } S \ x$   
**by**  $(\text{rule B-in})$   
**show**  $C \in \text{fundamental-group-space } S \ x$   
**by**  $(\text{rule C-in})$   
**show**  $\text{some-loop } S \ x \ B \in \text{loop-space } S \ x$   
**by**  $(\text{rule repB}(1))$   
**show**  $\text{some-loop } S \ x \ C \in \text{loop-space } S \ x$   
**by**  $(\text{rule repC}(1))$   
**show**  $B = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ B)$   
**by**  $(\text{rule repB}(2))$   
**show**  $C = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ C)$   
**by**  $(\text{rule repC}(2))$

**qed**  
**have** *AB-eq*:  
 $\text{fundamental-group-mult } S \ x \ A \ B =$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B)$   
**proof** (*rule fundamental-group-mult-eqI*)  
**show**  $A \in \text{fundamental-group-space } S \ x$   
**by** (*rule A-in*)  
**show**  $B \in \text{fundamental-group-space } S \ x$   
**by** (*rule B-in*)  
**show**  $\text{some-loop } S \ x \ A \in \text{loop-space } S \ x$   
**by** (*rule repA(1)*)  
**show**  $\text{some-loop } S \ x \ B \in \text{loop-space } S \ x$   
**by** (*rule repB(1)*)  
**show**  $A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**by** (*rule repA(2)*)  
**show**  $B = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ B)$   
**by** (*rule repB(2)*)  
**qed**  
**have** *join-BC*:  $\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C \in \text{loop-space } S \ x$   
**using** *repB(1) repC(1)* **by** (*rule loop-space-join*)  
**have** *join-AB*:  $\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B \in \text{loop-space } S \ x$   
**using** *repA(1) repB(1)* **by** (*rule loop-space-join*)  
**have** *left-eq*:  
 $\text{fundamental-group-mult } S \ x \ A \ (\text{fundamental-group-mult } S \ x \ B \ C) =$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ A \ +++ \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x$   
*C))*  
**proof** (*rule fundamental-group-mult-eqI*)  
**show**  $A \in \text{fundamental-group-space } S \ x$   
**by** (*rule A-in*)  
**show**  $\text{fundamental-group-mult } S \ x \ B \ C \in \text{fundamental-group-space } S \ x$   
**by** (*rule fundamental-group-mult-in-space[OF B-in C-in]*)  
**show**  $\text{some-loop } S \ x \ A \in \text{loop-space } S \ x$   
**by** (*rule repA(1)*)  
**show**  $\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C \in \text{loop-space } S \ x$   
**by** (*rule join-BC*)  
**show**  $A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**by** (*rule repA(2)*)  
**show**  $\text{fundamental-group-mult } S \ x \ B \ C =$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C)$   
**by** (*rule BC-eq*)  
**qed**  
**have** *right-eq*:  
 $\text{fundamental-group-mult } S \ x \ (\text{fundamental-group-mult } S \ x \ A \ B) \ C =$   
 $\text{loop-class } S \ x \ ((\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B) \ +++ \ \text{some-loop } S \ x$   
*C)*  
**proof** (*rule fundamental-group-mult-eqI*)  
**show**  $\text{fundamental-group-mult } S \ x \ A \ B \in \text{fundamental-group-space } S \ x$   
**by** (*rule fundamental-group-mult-in-space[OF A-in B-in]*)  
**show**  $C \in \text{fundamental-group-space } S \ x$

**by** (rule *C-in*)  
**show**  $\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B \in \text{loop-space } S \ x$   
**by** (rule *join-AB*)  
**show**  $\text{some-loop } S \ x \ C \in \text{loop-space } S \ x$   
**by** (rule *repC(1)*)  
**show**  $\text{fundamental-group-mult } S \ x \ A \ B =$   
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B)$   
**by** (rule *AB-eq*)  
**show**  $C = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ C)$   
**by** (rule *repC(2)*)  
**qed**  
**have** *left-loop*:  $\text{some-loop } S \ x \ A \ +++ \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C)$   
 $\in \text{loop-space } S \ x$   
**using** *repA(1) join-BC* **by** (rule *loop-space-join*)  
**have** *right-loop*:  $(\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B) \ +++ \ \text{some-loop } S \ x \ C$   
 $\in \text{loop-space } S \ x$   
**using** *join-AB repC(1)* **by** (rule *loop-space-join*)  
**have** *hom-assoc*:  
 $\text{homotopic-paths } S \ (\text{some-loop } S \ x \ A \ +++ \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C))$   
 $(\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B) \ +++ \ \text{some-loop } S \ x \ C)$   
**using** *repA(1) repB(1) repC(1)*  
**unfolding** *loop-space-def*  
**by** (auto *intro: homotopic-paths-assoc*)  
**have** *class-eq*:  
 $\text{loop-class } S \ x \ (\text{some-loop } S \ x \ A \ +++ \ (\text{some-loop } S \ x \ B \ +++ \ \text{some-loop } S \ x \ C)) =$   
 $\text{loop-class } S \ x \ ((\text{some-loop } S \ x \ A \ +++ \ \text{some-loop } S \ x \ B) \ +++ \ \text{some-loop } S \ x \ C)$   
**by** (rule *loop-class-eqI[OF left-loop right-loop hom-assoc]*)  
**show** *?thesis*  
**using** *left-eq right-eq class-eq* **by** *simp*  
**qed**

**lemma** *fundamental-group-mult-inv-right*:  
**assumes**  $x \in S$   
**and**  $A \in \text{fundamental-group-space } S \ x$   
**shows**  $\text{fundamental-group-mult } S \ x \ A \ (\text{fundamental-group-inv } S \ x \ A) = \text{fundamental-group-one } S \ x$   
**proof** –  
**have** *const*:  $(\lambda-. x) \in \text{loop-space } S \ x$   
**using** *constant-loop-in-space[OF assms(1)]* .  
**have** *repA*:  $\text{some-loop } S \ x \ A \in \text{loop-space } S \ x \ A = \text{loop-class } S \ x \ (\text{some-loop } S \ x \ A)$   
**using** *some-loop-spec[OF assms(2)]* **by** *auto*  
**have** *loopA*:  
 $\text{path } (\text{some-loop } S \ x \ A)$   
 $\text{path-image } (\text{some-loop } S \ x \ A) \subseteq S$   
 $\text{pathstart } (\text{some-loop } S \ x \ A) = x$

$pathfinish (some-loop S x A) = x$   
**using**  $repA(1)$  **unfolding**  $loop-space-def$  **by**  $auto$   
**have**  $inv-eq$ :  
 $fundamental-group-inv S x A = loop-class S x (reversepath (some-loop S x A))$   
**by** (rule  $fundamental-group-inv-eqI[OF assms(2) repA(1) repA(2)]$ )  
**have**  $mult-eq$ :  
 $fundamental-group-mult S x A (fundamental-group-inv S x A) =$   
 $loop-class S x (some-loop S x A +++ reversepath (some-loop S x A))$   
**proof** (rule  $fundamental-group-mult-eqI$ )  
**show**  $A \in fundamental-group-space S x$   
**by** (rule  $assms(2)$ )  
**show**  $fundamental-group-inv S x A \in fundamental-group-space S x$   
**by** (rule  $fundamental-group-inv-in-space[OF assms(2)]$ )  
**show**  $some-loop S x A \in loop-space S x$   
**by** (rule  $repA(1)$ )  
**show**  $reversepath (some-loop S x A) \in loop-space S x$   
**by** (rule  $loop-space-reversepath[OF repA(1)]$ )  
**show**  $A = loop-class S x (some-loop S x A)$   
**by** (rule  $repA(2)$ )  
**show**  $fundamental-group-inv S x A = loop-class S x (reversepath (some-loop S$   
 $x A))$   
**by** (rule  $inv-eq$ )  
**qed**  
**have**  $join-loop: some-loop S x A +++ reversepath (some-loop S x A) \in loop-space$   
 $S x$   
**using**  $repA(1)$   $loop-space-reversepath[OF repA(1)]$  **by** (rule  $loop-space-join$ )  
**have**  $hom: homotopic-paths S (some-loop S x A +++ reversepath (some-loop S$   
 $x A)) (\lambda-. x)$   
**using**  $homotopic-paths-rinv-const[OF loopA(1) loopA(2)] loopA(3)$  **by**  $simp$   
**have**  $class-eq$ :  
 $loop-class S x (some-loop S x A +++ reversepath (some-loop S x A)) = loop-class$   
 $S x (\lambda-. x)$   
**by** (rule  $loop-class-eqI[OF join-loop const hom]$ )  
**show**  $?thesis$   
**using**  $mult-eq class-eq$  **unfolding**  $fundamental-group-one-def$  **by**  $simp$   
**qed**

**lemma**  $fundamental-group-mult-inv-left$ :  
**assumes**  $x \in S$   
**and**  $A \in fundamental-group-space S x$   
**shows**  $fundamental-group-mult S x (fundamental-group-inv S x A) A = funda-$   
 $mental-group-one S x$   
**proof** –  
**have**  $const: (\lambda-. x) \in loop-space S x$   
**using**  $constant-loop-in-space[OF assms(1)]$  .  
**have**  $repA: some-loop S x A \in loop-space S x A = loop-class S x (some-loop S x$   
 $A)$   
**using**  $some-loop-spec[OF assms(2)]$  **by**  $auto$   
**have**  $loopA$ :

$path (some-loop S x A)$   
 $path-image (some-loop S x A) \subseteq S$   
 $pathstart (some-loop S x A) = x$   
 $pathfinish (some-loop S x A) = x$   
**using**  $repA(1)$  **unfolding**  $loop-space-def$  **by**  $auto$   
**have**  $inv-eq$ :  
 $fundamental-group-inv S x A = loop-class S x (reversepath (some-loop S x A))$   
**by** (rule  $fundamental-group-inv-eqI[OF assms(2) repA(1) repA(2)]$ )  
**have**  $mult-eq$ :  
 $fundamental-group-mult S x (fundamental-group-inv S x A) A =$   
 $loop-class S x (reversepath (some-loop S x A) +++ some-loop S x A)$   
**proof** (rule  $fundamental-group-mult-eqI$ )  
**show**  $fundamental-group-inv S x A \in fundamental-group-space S x$   
**by** (rule  $fundamental-group-inv-in-space[OF assms(2)]$ )  
**show**  $A \in fundamental-group-space S x$   
**by** (rule  $assms(2)$ )  
**show**  $reversepath (some-loop S x A) \in loop-space S x$   
**by** (rule  $loop-space-reversepath[OF repA(1)]$ )  
**show**  $some-loop S x A \in loop-space S x$   
**by** (rule  $repA(1)$ )  
**show**  $fundamental-group-inv S x A = loop-class S x (reversepath (some-loop S$   
 $x A))$   
**by** (rule  $inv-eq$ )  
**show**  $A = loop-class S x (some-loop S x A)$   
**by** (rule  $repA(2)$ )  
**qed**  
**have**  $join-loop: reversepath (some-loop S x A) +++ some-loop S x A \in loop-space$   
 $S x$   
**using**  $loop-space-reversepath[OF repA(1)] repA(1)$  **by** (rule  $loop-space-join$ )  
**have**  $hom: homotopic-paths S (reversepath (some-loop S x A) +++ some-loop S$   
 $x A) (\lambda-. x)$   
**using**  $homotopic-paths-linv-const[OF loopA(1) loopA(2)] loopA(4)$  **by**  $simp$   
**have**  $class-eq$ :  
 $loop-class S x (reversepath (some-loop S x A) +++ some-loop S x A) = loop-class$   
 $S x (\lambda-. x)$   
**by** (rule  $loop-class-eqI[OF join-loop const hom]$ )  
**show**  $?thesis$   
**using**  $mult-eq class-eq$  **unfolding**  $fundamental-group-one-def$  **by**  $simp$   
**qed**

**lemma**  $trivial-loop-class-in-space$ :

**assumes**  $x \in S$   
**shows**  $loop-class S x (\lambda-. x) \in fundamental-group-space S x$   
**using**  $constant-loop-in-space[OF assms]$  **by** (rule  $loop-class-in-space$ )

**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
  show fundamental-group-one S x ∈ fundamental-group-space S x
    using x-in by (rule fundamental-group-one-in-space)
next
  fix A B
  assume A ∈ fundamental-group-space S x B ∈ fundamental-group-space S x
  then show fundamental-group-mult S x A B ∈ fundamental-group-space S x
    by (rule fundamental-group-mult-in-space)
next
  fix A
  assume A ∈ fundamental-group-space S x
  then show fundamental-group-inv S x A ∈ fundamental-group-space S x
    by (rule fundamental-group-inv-in-space)
next
  fix A B C
  assume A-in: A ∈ fundamental-group-space S x
    and B-in: B ∈ fundamental-group-space S x
    and C-in: C ∈ fundamental-group-space S x
  show
    fundamental-group-mult S x (fundamental-group-mult S x A B) C =
      fundamental-group-mult S x A (fundamental-group-mult S x B C)
    by (rule sym, rule fundamental-group-mult-assoc[OF A-in B-in C-in])
next
  fix A
  assume A-in: A ∈ fundamental-group-space S x
  show fundamental-group-mult S x (fundamental-group-one S x) A = A
    by (rule fundamental-group-mult-one-left[OF x-in A-in])
next
  fix A
  assume A-in: A ∈ fundamental-group-space S x
  show fundamental-group-mult S x A (fundamental-group-one S x) = A
    by (rule fundamental-group-mult-one-right[OF x-in A-in])
next
  fix A
  assume A-in: A ∈ fundamental-group-space S x
  show fundamental-group-mult S x (fundamental-group-inv S x A) A = funda-
    mental-group-one S x
    by (rule fundamental-group-mult-inv-left[OF x-in A-in])
next
  fix A
  assume A-in: A ∈ fundamental-group-space S x
  show fundamental-group-mult S x A (fundamental-group-inv S x A) = funda-
    mental-group-one S x
    by (rule fundamental-group-mult-inv-right[OF x-in A-in])
qed

```

```

lemma fundamental-group-map-carrier-group-hom:
  assumes x-in:  $x \in S$ 
    and cont-h: continuous-on  $S$   $h$ 
    and map-h:  $h \in S \rightarrow T$ 
    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 –
  have y-in:  $y \in T$ 
    using x-in map-h hx by auto
  show ?thesis
proof unfold-locales
  show fundamental-group-one  $S$   $x \in$  fundamental-group-space  $S$   $x$ 
    by (rule fundamental-group-one-in-space[OF x-in])
next
  fix  $A$   $B$ 
  assume A-in:  $A \in$  fundamental-group-space  $S$   $x$ 
    and B-in:  $B \in$  fundamental-group-space  $S$   $x$ 
  show fundamental-group-mult  $S$   $x$   $A$   $B \in$  fundamental-group-space  $S$   $x$ 
    by (rule fundamental-group-mult-in-space[OF A-in B-in])
next
  fix  $A$ 
  assume A-in:  $A \in$  fundamental-group-space  $S$   $x$ 
  show fundamental-group-inv  $S$   $x$   $A \in$  fundamental-group-space  $S$   $x$ 
    by (rule fundamental-group-inv-in-space[OF A-in])
next
  fix  $A$   $B$   $C$ 
  assume 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  $S$   $x$ 
  show fundamental-group-mult  $S$   $x$  (fundamental-group-mult  $S$   $x$   $A$   $B$ )  $C =$ 
    fundamental-group-mult  $S$   $x$   $A$  (fundamental-group-mult  $S$   $x$   $B$   $C$ )
    by (rule sym, rule fundamental-group-mult-assoc[OF A-in B-in C-in])
next
  fix  $A$ 
  assume A-in:  $A \in$  fundamental-group-space  $S$   $x$ 
  show fundamental-group-mult  $S$   $x$  (fundamental-group-one  $S$   $x$ )  $A = A$ 
    by (rule fundamental-group-mult-one-left[OF x-in A-in])
next
  fix  $A$ 
  assume A-in:  $A \in$  fundamental-group-space  $S$   $x$ 

```

```

show fundamental-group-mult  $S$   $x$   $A$  (fundamental-group-one  $S$   $x$ ) =  $A$ 
  by (rule fundamental-group-mult-one-right[OF x-in A-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $S$   $x$ 
show fundamental-group-mult  $S$   $x$  (fundamental-group-inv  $S$   $x$   $A$ )  $A$  =
  fundamental-group-one  $S$   $x$ 
  by (rule fundamental-group-mult-inv-left[OF x-in A-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $S$   $x$ 
show fundamental-group-mult  $S$   $x$   $A$  (fundamental-group-inv  $S$   $x$   $A$ ) =
  fundamental-group-one  $S$   $x$ 
  by (rule fundamental-group-mult-inv-right[OF x-in A-in])
next
show fundamental-group-one  $T$   $y \in$  fundamental-group-space  $T$   $y$ 
  by (rule fundamental-group-one-in-space[OF y-in])
next
fix  $A$   $B$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 
  and  $B$ -in:  $B \in$  fundamental-group-space  $T$   $y$ 
show fundamental-group-mult  $T$   $y$   $A$   $B \in$  fundamental-group-space  $T$   $y$ 
  by (rule fundamental-group-mult-in-space[OF A-in B-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 
show fundamental-group-inv  $T$   $y$   $A \in$  fundamental-group-space  $T$   $y$ 
  by (rule fundamental-group-inv-in-space[OF A-in])
next
fix  $A$   $B$   $C$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 
  and  $B$ -in:  $B \in$  fundamental-group-space  $T$   $y$ 
  and  $C$ -in:  $C \in$  fundamental-group-space  $T$   $y$ 
show fundamental-group-mult  $T$   $y$  (fundamental-group-mult  $T$   $y$   $A$   $B$ )  $C$  =
  fundamental-group-mult  $T$   $y$   $A$  (fundamental-group-mult  $T$   $y$   $B$   $C$ )
  by (rule sym, rule fundamental-group-mult-assoc[OF A-in B-in C-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 
show fundamental-group-mult  $T$   $y$  (fundamental-group-one  $T$   $y$ )  $A$  =  $A$ 
  by (rule fundamental-group-mult-one-left[OF y-in A-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 
show fundamental-group-mult  $T$   $y$   $A$  (fundamental-group-one  $T$   $y$ ) =  $A$ 
  by (rule fundamental-group-mult-one-right[OF y-in A-in])
next
fix  $A$ 
assume  $A$ -in:  $A \in$  fundamental-group-space  $T$   $y$ 

```

```

show fundamental-group-mult  $T y$  (fundamental-group-inv  $T y A$ )  $A =$ 
  fundamental-group-one  $T y$ 
  by (rule fundamental-group-mult-inv-left[OF y-in A-in])
next
fix  $A$ 
assume  $A-in: A \in \text{fundamental-group-space } T y$ 
show fundamental-group-mult  $T y A$  (fundamental-group-inv  $T y A$ ) =
  fundamental-group-one  $T y$ 
  by (rule fundamental-group-mult-inv-right[OF y-in A-in])
next
fix  $A$ 
assume  $A-in: A \in \text{fundamental-group-space } S x$ 
show fundamental-group-map  $S x T y h A \in \text{fundamental-group-space } T y$ 
  by (rule fundamental-group-map-in-space[OF A-in cont-h map-h hx])
next
fix  $A B$ 
assume  $A-in: A \in \text{fundamental-group-space } S x$ 
  and  $B-in: B \in \text{fundamental-group-space } S x$ 
show fundamental-group-map  $S x T y h$  (fundamental-group-mult  $S x A B$ ) =
  fundamental-group-mult  $T y$ 
  (fundamental-group-map  $S x T y h A$ )
  (fundamental-group-map  $S x T y h B$ )
  by (rule fundamental-group-map-mult[OF A-in B-in cont-h map-h hx])
qed
qed

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* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{loopin-class } X x p = \{q. q \in \text{loopin-space } X x \wedge \text{homotopic-pathsin } X q p\}$

**definition** *fundamental-groupin-space* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow ((\text{real} \Rightarrow 'a) \text{ set}) \text{ set}$

**where**

$\text{fundamental-groupin-space } X x = \text{loopin-class } X x ' \text{loopin-space } X x$

**definition** *some-loopin* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a)$

**where**

$\text{some-loopin } X \ x \ Q = (\text{SOME } p. p \in \text{loopin-space } X \ x \wedge Q = \text{loopin-class } X \ x \ p)$

**definition** *fundamental-groupin-one* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{fundamental-groupin-one } X \ x = \text{loopin-class } X \ x \ (\lambda-. x)$

**definition** *fundamental-groupin-mult* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow$

$(\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{fundamental-groupin-mult } X \ x \ A \ B =$

$\text{loopin-class } X \ x \ (\text{joinpathin } (\text{some-loopin } X \ x \ A) \ (\text{some-loopin } X \ x \ B))$

**definition** *fundamental-groupin-inv* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'a) \text{ set}$

**where**

$\text{fundamental-groupin-inv } X \ x \ A = \text{loopin-class } X \ x \ (\text{reversepathin } (\text{some-loopin } X \ x \ A))$

**definition** *loopin-image* ::

$('a \Rightarrow 'b) \Rightarrow (\text{real} \Rightarrow 'a) \Rightarrow (\text{real} \Rightarrow 'b)$

**where**

$\text{loopin-image } h \ p = h \circ p$

**definition** *fundamental-groupin-map* ::

$'a \text{ topology} \Rightarrow 'a \Rightarrow 'b \text{ topology} \Rightarrow 'b \Rightarrow$

$('a \Rightarrow 'b) \Rightarrow (\text{real} \Rightarrow 'a) \text{ set} \Rightarrow (\text{real} \Rightarrow 'b) \text{ set}$

**where**

$\text{fundamental-groupin-map } X \ x \ Y \ y \ h \ A =$

$\text{loopin-class } Y \ y \ (\text{loopin-image } h \ (\text{some-loopin } X \ x \ A))$

**lemma** *loopin-class-in-space*:

**assumes**  $p \in \text{loopin-space } X \ x$

**shows**  $\text{loopin-class } X \ x \ p \in \text{fundamental-groupin-space } X \ x$

**using** *assms* **unfolding** *fundamental-groupin-space-def* **by** *blast*

**lemma** *fundamental-groupin-spaceE*:

**assumes**  $Q \in \text{fundamental-groupin-space } X \ x$

**obtains**  $p$  **where**  $p \in \text{loopin-space } X \ x \ Q = \text{loopin-class } X \ x \ p$

**using** *assms* **unfolding** *fundamental-groupin-space-def* **by** *blast*

**lemma** *some-loopin-spec*:

**assumes**  $Q \in \text{fundamental-groupin-space } X \ x$

**shows**  $\text{some-loopin } X \ x \ Q \in \text{loopin-space } X \ x$

**and**  $Q = \text{loopin-class } X \ x \ (\text{some-loopin } X \ x \ Q)$   
**proof** –  
**from** *assms* **obtain**  $p$  **where**  $p: p \in \text{loopin-space } X \ x \ Q = \text{loopin-class } X \ x \ p$   
**by** (*rule fundamental-groupin-spaceE*)  
**have**  $ex: \exists p. p \in \text{loopin-space } X \ x \ \wedge \ Q = \text{loopin-class } X \ x \ p$   
**using**  $p$  **by** *blast*  
**have**  $\text{some-loopin } X \ x \ Q \in \text{loopin-space } X \ x \ \wedge \ Q = \text{loopin-class } X \ x \ (\text{some-loopin } X \ x \ Q)$   
**unfolding** *some-loopin-def* **by** (*rule someI-ex[OF ex]*)  
**then show**  $\text{some-loopin } X \ x \ Q \in \text{loopin-space } X \ x$   
**and**  $Q = \text{loopin-class } X \ x \ (\text{some-loopin } X \ x \ Q)$   
**by** *auto*  
**qed**

**lemma** *loopin-class-eqI*:  
**assumes**  $p: p \in \text{loopin-space } X \ x$   
**and**  $q: q \in \text{loopin-space } X \ x$   
**and**  $pq: \text{homotopic-pathsin } X \ p \ q$   
**shows**  $\text{loopin-class } X \ x \ p = \text{loopin-class } X \ x \ q$   
**proof** (*auto simp: loopin-class-def*)  
**fix**  $r$   
**assume**  $\text{homotopic-pathsin } X \ r \ p$   
**then show**  $\text{homotopic-pathsin } X \ r \ q$   
**using**  $pq$  **by** (*rule homotopic-pathsin-trans*)  
**next**  
**fix**  $r$   
**assume**  $\text{homotopic-pathsin } X \ r \ q$   
**moreover have**  $\text{homotopic-pathsin } X \ q \ p$   
**using**  $pq$  **by** (*rule homotopic-pathsin-sym*)  
**ultimately show**  $\text{homotopic-pathsin } X \ r \ p$   
**by** (*rule homotopic-pathsin-trans*)  
**qed**

**lemma** *loopin-class-eq-iff*:  
**assumes**  $p: p \in \text{loopin-space } X \ x$   
**and**  $q: q \in \text{loopin-space } X \ x$   
**shows**  $\text{loopin-class } X \ x \ p = \text{loopin-class } X \ x \ q \iff \text{homotopic-pathsin } X \ p \ q$   
**proof**  
**assume**  $h: \text{loopin-class } X \ x \ p = \text{loopin-class } X \ x \ q$   
**have**  $p \in \text{loopin-class } X \ x \ p$   
**using**  $p$  **by** (*auto simp: loopin-class-def elim: loopin-spaceE*)  
**then have**  $p \in \text{loopin-class } X \ x \ q$   
**using**  $h$  **by** *simp*  
**then show**  $\text{homotopic-pathsin } X \ p \ q$   
**unfolding** *loopin-class-def* **using**  $p$  **by** *auto*  
**next**  
**assume**  $pq: \text{homotopic-pathsin } X \ p \ q$   
**show**  $\text{loopin-class } X \ x \ p = \text{loopin-class } X \ x \ q$   
**by** (*rule loopin-class-eqI[OF p q pq]*)

qed

**lemma** *fundamental-groupin-one-in-space*:

**assumes**  $x \in \text{topspace } X$

**shows**  $\text{fundamental-groupin-one } X x \in \text{fundamental-groupin-space } X x$

**unfolding** *fundamental-groupin-one-def*

**by** (*rule loopin-class-in-space, rule constant-loopin-in-space, fact assms*)

**lemma** *fundamental-groupin-mult-eqI*:

**assumes**  $A\text{-in}: A \in \text{fundamental-groupin-space } X x$

**and**  $B\text{-in}: B \in \text{fundamental-groupin-space } X x$

**and**  $p: p \in \text{loopin-space } X x$

**and**  $q: q \in \text{loopin-space } X x$

**and**  $A: A = \text{loopin-class } X x p$

**and**  $B: B = \text{loopin-class } X x q$

**shows**  $\text{fundamental-groupin-mult } X x A B = \text{loopin-class } X x (\text{joinpathin } p q)$

**proof** –

**have**  $\text{rep}A: \text{some-loopin } X x A \in \text{loopin-space } X x A = \text{loopin-class } X x$   
(*some-loopin } X x A*)

**using** *some-loopin-spec[OF A-in]* **by** *auto*

**have**  $\text{rep}B: \text{some-loopin } X x B \in \text{loopin-space } X x B = \text{loopin-class } X x$   
(*some-loopin } X x B*)

**using** *some-loopin-spec[OF B-in]* **by** *auto*

**have**  $\text{homo}A': \text{homotopic-pathsin } X p (\text{some-loopin } X x A)$

**using**  $\text{rep}A p A$  **by** (*simp add: loopin-class-eq-iff*)

**have**  $\text{homo}A: \text{homotopic-pathsin } X (\text{some-loopin } X x A) p$

**using**  $\text{homo}A'$  **by** (*rule homotopic-pathsin-sym*)

**have**  $\text{homo}B': \text{homotopic-pathsin } X q (\text{some-loopin } X x B)$

**using**  $\text{rep}B q B$  **by** (*simp add: loopin-class-eq-iff*)

**have**  $\text{homo}B: \text{homotopic-pathsin } X (\text{some-loopin } X x B) q$

**using**  $\text{homo}B'$  **by** (*rule homotopic-pathsin-sym*)

**have** *join-hom*:

*homotopic-pathsin } X*

(*joinpathin } (some-loopin } X x A) (some-loopin } X x B)*)

(*joinpathin } p q*)

**proof** –

**have**  $\text{rep}A\text{-end}: \text{some-loopin } X x A 1 = x$

**using**  $\text{rep}A(1)$  **by** (*auto elim: loopin-spaceE*)

**have**  $\text{rep}B\text{-start}: \text{some-loopin } X x B 0 = x$

**using**  $\text{rep}B(1)$  **by** (*auto elim: loopin-spaceE*)

**show** *?thesis*

**by** (*rule homotopic-pathsin-joinpathin[OF homoA homoB]*) (*simp add:*  
*repA-end repB-start*)

**qed**

**have**  $\text{loopin-class } X x (\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B)) =$   
 $\text{loopin-class } X x (\text{joinpathin } p q)$

**by** (*rule loopin-class-eqI[OF loopin-space-joinpathin[OF repA(1) repB(1)]*  
*loopin-space-joinpathin[OF p q] join-hom*)

**then show** *?thesis*

**unfolding** *fundamental-groupin-mult-def* .  
**qed**

**lemma** *fundamental-groupin-mult-in-space*:  
**assumes**  $A \in \text{fundamental-groupin-space } X \ x$   
**and**  $B \in \text{fundamental-groupin-space } X \ x$   
**shows**  $\text{fundamental-groupin-mult } X \ x \ A \ B \in \text{fundamental-groupin-space } X \ x$   
**proof** –  
**have**  $\text{rep}A: \text{some-loopin } X \ x \ A \in \text{loopin-space } X \ x$   
**using**  $\text{some-loopin-spec}[OF \ \text{assms}(1)]$  **by** *auto*  
**have**  $\text{rep}B: \text{some-loopin } X \ x \ B \in \text{loopin-space } X \ x$   
**using**  $\text{some-loopin-spec}[OF \ \text{assms}(2)]$  **by** *auto*  
**show** *?thesis*  
**unfolding** *fundamental-groupin-mult-def*  
**using**  $\text{loopin-space-joinpathin}[OF \ \text{rep}A \ \text{rep}B]$  **by** (*rule loopin-class-in-space*)  
**qed**

**lemma** *fundamental-groupin-inv-eqI*:  
**assumes**  $A\text{-in}: A \in \text{fundamental-groupin-space } X \ x$   
**and**  $p: p \in \text{loopin-space } X \ x$   
**and**  $A: A = \text{loopin-class } X \ x \ p$   
**shows**  $\text{fundamental-groupin-inv } X \ x \ A = \text{loopin-class } X \ x \ (\text{reversepathin } p)$   
**proof** –  
**have**  $\text{rep}A: \text{some-loopin } X \ x \ A \in \text{loopin-space } X \ x \ A = \text{loopin-class } X \ x$   
*(some-loopin X x A)*  
**using**  $\text{some-loopin-spec}[OF \ A\text{-in}]$  **by** *auto*  
**have**  $\text{homo}A': \text{homotopic-pathsin } X \ p \ (\text{some-loopin } X \ x \ A)$   
**using**  $\text{rep}A \ p \ A$  **by** (*simp add: loopin-class-eq-iff*)  
**have**  $\text{homo}A: \text{homotopic-pathsin } X \ (\text{some-loopin } X \ x \ A) \ p$   
**using**  $\text{homo}A'$  **by** (*rule homotopic-pathsin-sym*)  
**have**  $\text{rev-hom}: \text{homotopic-pathsin } X \ (\text{reversepathin } (\text{some-loopin } X \ x \ A)) \ (\text{reversepathin } p)$   
**using**  $\text{homo}A$  **by** (*rule homotopic-pathsin-reversepathin-D*)  
**have**  $\text{loopin-class } X \ x \ (\text{reversepathin } (\text{some-loopin } X \ x \ A)) = \text{loopin-class } X \ x$   
*(reversepathin p)*  
**by** (*rule loopin-class-eqI[OF loopin-space-reversepathin[OF repA(1)]*  
*loopin-space-reversepathin[OF p] rev-hom]*)  
**then show** *?thesis*  
**unfolding** *fundamental-groupin-inv-def* .  
**qed**

**lemma** *fundamental-groupin-inv-in-space*:  
**assumes**  $A \in \text{fundamental-groupin-space } X \ x$   
**shows**  $\text{fundamental-groupin-inv } X \ x \ A \in \text{fundamental-groupin-space } X \ x$   
**proof** –  
**have**  $\text{rep}A: \text{some-loopin } X \ x \ A \in \text{loopin-space } X \ x$   
**using**  $\text{some-loopin-spec}[OF \ \text{assms}]$  **by** *auto*  
**show** *?thesis*  
**unfolding** *fundamental-groupin-inv-def*

using *loopin-space-reversepathin*[*OF repA*] by (rule *loopin-class-in-space*)  
qed

**lemma** *loopin-image-in-space*:

assumes *p*:  $p \in \text{loopin-space } X \ x$   
and *h*: *continuous-map*  $X \ Y \ f$   
and *fx*:  $f \ x = y$

shows *loopin-image*  $f \ p \in \text{loopin-space } Y \ y$

**proof** –

from *p* obtain *p-in* where *p-in*: *pathin*  $X \ p \ p \ 0 = x \ p \ 1 = x$   
by (rule *loopin-spaceE*)

have *pathin*  $Y \ (\text{loopin-image } f \ p)$

using *p-in*(1) *h* unfolding *loopin-image-def* by (rule *pathin-compose*)

moreover have *loopin-image*  $f \ p \ 0 = y$  *loopin-image*  $f \ p \ 1 = y$

using *p-in* *fx* unfolding *loopin-image-def* by *simp-all*

ultimately show *?thesis*

unfolding *loopin-space-def* by *blast*

qed

**lemma** *fundamental-groupin-map-rep*:

assumes *A*:  $A \in \text{fundamental-groupin-space } X \ x$

and *p*:  $p \in \text{loopin-space } X \ x$

and *rep*:  $A = \text{loopin-class } X \ x \ p$

and *h*: *continuous-map*  $X \ Y \ f$

and *fx*:  $f \ x = y$

shows *fundamental-groupin-map*  $X \ x \ Y \ y \ f \ A = \text{loopin-class } Y \ y \ (\text{loopin-image } f \ p)$

**proof** –

have *sp*: *some-loopin*  $X \ x \ A \in \text{loopin-space } X \ x$

using *A* by (rule *some-loopin-spec*)

have *A'*:  $A = \text{loopin-class } X \ x \ (\text{some-loopin } X \ x \ A)$

using *A* by (rule *some-loopin-spec*)

have *class-eq*: *loopin-class*  $X \ x \ (\text{some-loopin } X \ x \ A) = \text{loopin-class } X \ x \ p$

using *A'* *rep* by *simp*

have *hom*: *homotopic-paths**in*  $X \ (\text{some-loopin } X \ x \ A) \ p$

using *sp* *p* *class-eq* by (*simp* *add*: *loopin-class-eq-iff*)

have *img-some*: *loopin-image*  $f \ (\text{some-loopin } X \ x \ A) \in \text{loopin-space } Y \ y$

by (rule *loopin-image-in-space*[*OF sp h fx*])

have *img-p*: *loopin-image*  $f \ p \in \text{loopin-space } Y \ y$

by (rule *loopin-image-in-space*[*OF p h fx*])

have *img-hom*: *homotopic-paths**in*  $Y \ (\text{loopin-image } f \ (\text{some-loopin } X \ x \ A))$   
(*loopin-image*  $f \ p$ )

using *hom* *h* unfolding *loopin-image-def* by (rule *homotopic-paths**in*-*continuous-image*)

have *loopin-class*  $Y \ y \ (\text{loopin-image } f \ (\text{some-loopin } X \ x \ A)) = \text{loopin-class } Y \ y$   
(*loopin-image*  $f \ p$ )

by (rule *loopin-class-eqI*[*OF img-some img-p img-hom*])

then show *?thesis*

unfolding *fundamental-groupin-map-def* by *simp*

qed

**lemma** *fundamental-groupin-map-in-space*:

**assumes** *A*:  $A \in \text{fundamental-groupin-space } X \ x$

**and** *h*: *continuous-map*  $X \ Y \ f$

**and** *fx*:  $f \ x = y$

**shows** *fundamental-groupin-map*  $X \ x \ Y \ y \ f \ A \in \text{fundamental-groupin-space } Y \ y$

**proof** –

**have** *sp*: *some-loopin*  $X \ x \ A \in \text{loopin-space } X \ x$

**using** *A* **by** (*rule some-loopin-spec*)

**have** *img*: *loopin-image*  $f \ (\text{some-loopin } X \ x \ A) \in \text{loopin-space } Y \ y$

**by** (*rule loopin-image-in-space*[*OF sp h fx*])

**show** *?thesis*

**unfolding** *fundamental-groupin-map-def* **by** (*rule loopin-class-in-space*[*OF img*])

qed

**lemma** *loopin-image-joinpathin* [*simp*]:

*loopin-image*  $h \ (\text{joinpathin } p \ q) = \text{joinpathin} \ (\text{loopin-image } h \ p) \ (\text{loopin-image } h \ q)$

**unfolding** *loopin-image-def* *joinpathin-def* **by** (*rule ext*) *simp*

**lemma** *loopin-image-reversepathin* [*simp*]:

*loopin-image*  $h \ (\text{reversepathin } p) = \text{reversepathin} \ (\text{loopin-image } h \ p)$

**unfolding** *loopin-image-def* *reversepathin-def* **by** (*rule ext*) *simp*

**lemma** *fundamental-groupin-map-mult*:

**assumes** *A-in*:  $A \in \text{fundamental-groupin-space } X \ x$

**and** *B-in*:  $B \in \text{fundamental-groupin-space } X \ x$

**and** *h*: *continuous-map*  $X \ Y \ f$

**and** *fx*:  $f \ x = y$

**shows** *fundamental-groupin-map*  $X \ x \ Y \ y \ f \ (\text{fundamental-groupin-mult } X \ x \ A \ B)$

=

*fundamental-groupin-mult*  $Y \ y$

(*fundamental-groupin-map*  $X \ x \ Y \ y \ f \ A$ )

(*fundamental-groupin-map*  $X \ x \ Y \ y \ f \ B$ )

**proof** –

**have** *repA*: *some-loopin*  $X \ x \ A \in \text{loopin-space } X \ x \ A = \text{loopin-class } X \ x$   
(*some-loopin*  $X \ x \ A$ )

**using** *some-loopin-spec*[*OF A-in*] **by** *auto*

**have** *repB*: *some-loopin*  $X \ x \ B \in \text{loopin-space } X \ x \ B = \text{loopin-class } X \ x$   
(*some-loopin*  $X \ x \ B$ )

**using** *some-loopin-spec*[*OF B-in*] **by** *auto*

**have** *AB-in*: *fundamental-groupin-mult*  $X \ x \ A \ B \in \text{fundamental-groupin-space } X \ x$

**by** (*rule fundamental-groupin-mult-in-space*[*OF A-in B-in*])

**have** *AB-eq*:

*fundamental-groupin-mult*  $X \ x \ A \ B =$

*loopin-class*  $X \ x \ (\text{joinpathin} \ (\text{some-loopin } X \ x \ A) \ (\text{some-loopin } X \ x \ B))$

**by** (rule *fundamental-groupin-mult-eqI*[*OF A-in B-in repA(1) repB(1) repA(2) repB(2)*])  
**have** *map-A-in*:  
*fundamental-groupin-map X x Y y f A ∈ fundamental-groupin-space Y y*  
**by** (rule *fundamental-groupin-map-in-space*[*OF A-in h fx*])  
**have** *map-B-in*:  
*fundamental-groupin-map X x Y y f B ∈ fundamental-groupin-space Y y*  
**by** (rule *fundamental-groupin-map-in-space*[*OF B-in h fx*])  
**have** *map-A-eq*:  
*fundamental-groupin-map X x Y y f A =*  
*loopin-class Y y (loopin-image f (some-loopin X x A))*  
**by** (rule *fundamental-groupin-map-rep*[*OF A-in repA(1) repA(2) h fx*])  
**have** *map-B-eq*:  
*fundamental-groupin-map X x Y y f B =*  
*loopin-class Y y (loopin-image f (some-loopin X x B))*  
**by** (rule *fundamental-groupin-map-rep*[*OF B-in repB(1) repB(2) h fx*])  
**have** *left-eq*:  
*fundamental-groupin-map X x Y y f (fundamental-groupin-mult X x A B) =*  
*loopin-class Y y (loopin-image f (joinpathin (some-loopin X x A) (some-loopin X x B)))*  
**by** (rule *fundamental-groupin-map-rep*[*OF AB-in loopin-space-joinpathin*[*OF repA(1) repB(1) AB-eq h fx*])  
**have** *map-loop-A*: *loopin-image f (some-loopin X x A) ∈ loopin-space Y y*  
**by** (rule *loopin-image-in-space*[*OF repA(1) h fx*])  
**have** *map-loop-B*: *loopin-image f (some-loopin X x B) ∈ loopin-space Y y*  
**by** (rule *loopin-image-in-space*[*OF repB(1) h fx*])  
**have** *right-eq*:  
*fundamental-groupin-mult Y y*  
*(fundamental-groupin-map X x Y y f A)*  
*(fundamental-groupin-map X x Y y f B) =*  
*loopin-class Y y*  
*(joinpathin (loopin-image f (some-loopin X x A))*  
*(loopin-image f (some-loopin X x B)))*  
**by** (rule *fundamental-groupin-mult-eqI*[*OF map-A-in map-B-in map-loop-A*  
*map-loop-B map-A-eq map-B-eq*])  
**then show** *?thesis*  
**using** *left-eq* **by** *simp*  
**qed**

**lemma** *fundamental-groupin-mult-one-left*:

**assumes** *x-in*:  $x \in \text{topspace } X$   
**and** *A-in*:  $A \in \text{fundamental-groupin-space } X x$   
**shows** *fundamental-groupin-mult X x (fundamental-groupin-one X x) A = A*  
**proof** –  
**have** *const*:  $(\lambda-. x) \in \text{loopin-space } X x$   
**by** (rule *constant-loopin-in-space*[*OF x-in*])  
**have** *repA*:  $\text{some-loopin } X x A \in \text{loopin-space } X x A = \text{loopin-class } X x$   
*(some-loopin X x A)*  
**using** *some-loopin-spec*[*OF A-in*] **by** *auto*

```

have loopA: pathin X (some-loopin X x A) some-loopin X x A 0 = x some-loopin
X x A 1 = x
  using repA(1) by (auto elim: loopin-spaceE)
have mult-eq:
  fundamental-groupin-mult X x (fundamental-groupin-one X x) A =
  loopin-class X x (joinpathin (λ-. x) (some-loopin X x A))
proof (rule fundamental-groupin-mult-eqI)
show fundamental-groupin-one X x ∈ fundamental-groupin-space X x
  by (rule fundamental-groupin-one-in-space[OF x-in])
show A ∈ fundamental-groupin-space X x
  by (rule A-in)
show (λ-. x) ∈ loopin-space X x
  by (rule const)
show some-loopin X x A ∈ loopin-space X x
  by (rule repA(1))
show fundamental-groupin-one X x = loopin-class X x (λ-. x)
  by (simp add: fundamental-groupin-one-def)
show A = loopin-class X x (some-loopin X x A)
  by (rule repA(2))
qed
have join-loop: joinpathin (λ-. x) (some-loopin X x A) ∈ loopin-space X x
  by (rule loopin-space-joinpathin[OF const repA(1)])
have hom: homotopic-pathsin X (joinpathin (λ-. x) (some-loopin X x A))
(some-loopin X x A)
  using homotopic-pathsin-lid-const[OF loopA(1)] loopA(2) by simp
have class-eq:
  loopin-class X x (joinpathin (λ-. x) (some-loopin X x A)) = loopin-class X x
(some-loopin X x A)
  by (rule loopin-class-eqI[OF join-loop repA(1) hom])
show ?thesis
  using mult-eq class-eq repA(2) by simp
qed

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 –
have const: (λ-. x) ∈ loopin-space X x
  by (rule constant-loopin-in-space[OF x-in])
have repA: some-loopin X x A ∈ loopin-space X x A = loopin-class X x
(some-loopin X x A)
  using some-loopin-spec[OF A-in] by auto
have loopA: pathin X (some-loopin X x A) some-loopin X x A 0 = x some-loopin
X x A 1 = x
  using repA(1) by (auto elim: loopin-spaceE)
have mult-eq:
  fundamental-groupin-mult X x A (fundamental-groupin-one X x) =
  loopin-class X x (joinpathin (some-loopin X x A) (λ-. x))

```

**proof** (rule *fundamental-groupin-mult-eqI*)  
**show**  $A \in \text{fundamental-groupin-space } X \ x$   
**by** (rule *A-in*)  
**show**  $\text{fundamental-groupin-one } X \ x \in \text{fundamental-groupin-space } X \ x$   
**by** (rule *fundamental-groupin-one-in-space*[*OF x-in*])  
**show**  $\text{some-loopin } X \ x \ A \in \text{loopin-space } X \ x$   
**by** (rule *repA(1)*)  
**show**  $(\lambda-. \ x) \in \text{loopin-space } X \ x$   
**by** (rule *const*)  
**show**  $A = \text{loopin-class } X \ x \ (\text{some-loopin } X \ x \ A)$   
**by** (rule *repA(2)*)  
**show**  $\text{fundamental-groupin-one } X \ x = \text{loopin-class } X \ x \ (\lambda-. \ x)$   
**by** (*simp add: fundamental-groupin-one-def*)  
**qed**  
**have** *join-loop*:  $\text{joinpathin } (\text{some-loopin } X \ x \ A) \ (\lambda-. \ x) \in \text{loopin-space } X \ x$   
**by** (rule *loopin-space-joinpathin*[*OF repA(1) const*])  
**have** *hom*:  $\text{homotopic-pathsin } X \ (\text{joinpathin } (\text{some-loopin } X \ x \ A) \ (\lambda-. \ x))$   
(*some-loopin } X \ x \ A*)  
**using** *homotopic-pathsin-rid-const*[*OF loopA(1)*] *loopA(3)* **by** *simp*  
**have** *class-eq*:  
 $\text{loopin-class } X \ x \ (\text{joinpathin } (\text{some-loopin } X \ x \ A) \ (\lambda-. \ x)) = \text{loopin-class } X \ x$   
(*some-loopin } X \ x \ A*)  
**by** (rule *loopin-class-eqI*[*OF join-loop repA(1) hom*])  
**show** *?thesis*  
**using** *mult-eq class-eq repA(2)* **by** *simp*  
**qed**

**lemma** *fundamental-groupin-mult-assoc*:  
**assumes** *A-in*:  $A \in \text{fundamental-groupin-space } X \ x$   
**and** *B-in*:  $B \in \text{fundamental-groupin-space } X \ x$   
**and** *C-in*:  $C \in \text{fundamental-groupin-space } X \ x$   
**shows**  $\text{fundamental-groupin-mult } X \ x \ A \ (\text{fundamental-groupin-mult } X \ x \ B \ C) =$   
 $\text{fundamental-groupin-mult } X \ x \ (\text{fundamental-groupin-mult } X \ x \ A \ B) \ C$   
**proof** –  
**have** *repA*:  $\text{some-loopin } X \ x \ A \in \text{loopin-space } X \ x \ A = \text{loopin-class } X \ x$   
(*some-loopin } X \ x \ A*)  
**using** *some-loopin-spec*[*OF A-in*] **by** *auto*  
**have** *repB*:  $\text{some-loopin } X \ x \ B \in \text{loopin-space } X \ x \ B = \text{loopin-class } X \ x$   
(*some-loopin } X \ x \ B*)  
**using** *some-loopin-spec*[*OF B-in*] **by** *auto*  
**have** *repC*:  $\text{some-loopin } X \ x \ C \in \text{loopin-space } X \ x \ C = \text{loopin-class } X \ x$   
(*some-loopin } X \ x \ C*)  
**using** *some-loopin-spec*[*OF C-in*] **by** *auto*  
**have** *loopA*:  $\text{pathin } X \ (\text{some-loopin } X \ x \ A) \ \text{some-loopin } X \ x \ A \ 0 = x \ \text{some-loopin}$   
 $X \ x \ A \ 1 = x$   
**using** *repA(1)* **by** (*auto elim: loopin-spaceE*)  
**have** *loopB*:  $\text{pathin } X \ (\text{some-loopin } X \ x \ B) \ \text{some-loopin } X \ x \ B \ 0 = x \ \text{some-loopin}$   
 $X \ x \ B \ 1 = x$   
**using** *repB(1)* **by** (*auto elim: loopin-spaceE*)

**have** *loopC*: *pathin*  $X$  (*some-loopin*  $X$   $x$   $C$ ) *some-loopin*  $X$   $x$   $C$   $0 = x$  *some-loopin*  $X$   $x$   $C$   $1 = x$   
**using** *repC(1)* **by** (*auto elim: loopin-spaceE*)  
**have** *BC-eq*:  
*fundamental-groupin-mult*  $X$   $x$   $B$   $C =$   
*loopin-class*  $X$   $x$  (*joinpathin* (*some-loopin*  $X$   $x$   $B$ ) (*some-loopin*  $X$   $x$   $C$ ))  
**by** (*rule fundamental-groupin-mult-eqI*[*OF B-in C-in repB(1) repC(1) repB(2) repC(2)*])  
**have** *AB-eq*:  
*fundamental-groupin-mult*  $X$   $x$   $A$   $B =$   
*loopin-class*  $X$   $x$  (*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*some-loopin*  $X$   $x$   $B$ ))  
**by** (*rule fundamental-groupin-mult-eqI*[*OF A-in B-in repA(1) repB(1) repA(2) repB(2)*])  
**have** *join-BC*: *joinpathin* (*some-loopin*  $X$   $x$   $B$ ) (*some-loopin*  $X$   $x$   $C$ )  $\in$  *loopin-space*  $X$   $x$   
**by** (*rule loopin-space-joinpathin*[*OF repB(1) repC(1)*])  
**have** *join-AB*: *joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*some-loopin*  $X$   $x$   $B$ )  $\in$  *loopin-space*  $X$   $x$   
**by** (*rule loopin-space-joinpathin*[*OF repA(1) repB(1)*])  
**have** *left-eq*:  
*fundamental-groupin-mult*  $X$   $x$   $A$  (*fundamental-groupin-mult*  $X$   $x$   $B$   $C$ ) =  
*loopin-class*  $X$   $x$   
(*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*joinpathin* (*some-loopin*  $X$   $x$   $B$ ) (*some-loopin*  $X$   $x$   $C$ )))  
**proof** (*rule fundamental-groupin-mult-eqI*)  
**show**  $A \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule A-in*)  
**show** *fundamental-groupin-mult*  $X$   $x$   $B$   $C \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule fundamental-groupin-mult-in-space*[*OF B-in C-in*])  
**show** *some-loopin*  $X$   $x$   $A \in$  *loopin-space*  $X$   $x$   
**by** (*rule repA(1)*)  
**show** *joinpathin* (*some-loopin*  $X$   $x$   $B$ ) (*some-loopin*  $X$   $x$   $C$ )  $\in$  *loopin-space*  $X$   $x$   
**by** (*rule join-BC*)  
**show**  $A =$  *loopin-class*  $X$   $x$  (*some-loopin*  $X$   $x$   $A$ )  
**by** (*rule repA(2)*)  
**show** *fundamental-groupin-mult*  $X$   $x$   $B$   $C =$   
*loopin-class*  $X$   $x$  (*joinpathin* (*some-loopin*  $X$   $x$   $B$ ) (*some-loopin*  $X$   $x$   $C$ ))  
**by** (*rule BC-eq*)  
**qed**  
**have** *right-eq*:  
*fundamental-groupin-mult*  $X$   $x$  (*fundamental-groupin-mult*  $X$   $x$   $A$   $B$ )  $C =$   
*loopin-class*  $X$   $x$   
(*joinpathin* (*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*some-loopin*  $X$   $x$   $B$ )) (*some-loopin*  $X$   $x$   $C$ ))  
**proof** (*rule fundamental-groupin-mult-eqI*)  
**show** *fundamental-groupin-mult*  $X$   $x$   $A$   $B \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule fundamental-groupin-mult-in-space*[*OF A-in B-in*])  
**show**  $C \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule C-in*)

**show**  $\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B) \in \text{loopin-space } X x$   
**by** (rule *join-AB*)  
**show**  $\text{some-loopin } X x C \in \text{loopin-space } X x$   
**by** (rule *repC(1)*)  
**show**  $\text{fundamental-groupin-mult } X x A B =$   
 $\text{loopin-class } X x (\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B))$   
**by** (rule *AB-eq*)  
**show**  $C = \text{loopin-class } X x (\text{some-loopin } X x C)$   
**by** (rule *repC(2)*)  
**qed**  
**have** *left-loop*:  
 $\text{joinpathin } (\text{some-loopin } X x A) (\text{joinpathin } (\text{some-loopin } X x B) (\text{some-loopin } X x C)) \in \text{loopin-space } X x$   
**by** (rule *loopin-space-joinpathin[OF repA(1) join-BC]*)  
**have** *right-loop*:  
 $\text{joinpathin } (\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B)) (\text{some-loopin } X x C) \in \text{loopin-space } X x$   
**by** (rule *loopin-space-joinpathin[OF join-AB repC(1)]*)  
**have** *hom-assoc*:  
 $\text{homotopic-pathsin } X$   
 $(\text{joinpathin } (\text{some-loopin } X x A) (\text{joinpathin } (\text{some-loopin } X x B) (\text{some-loopin } X x C)))$   
 $(\text{joinpathin } (\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B)) (\text{some-loopin } X x C))$   
**by** (rule *homotopic-pathsin-assoc[OF loopA(1) loopB(1) loopC(1)]*) (*simp-all add: loopA loopB loopC*)  
**have** *class-eq*:  
 $\text{loopin-class } X x$   
 $(\text{joinpathin } (\text{some-loopin } X x A) (\text{joinpathin } (\text{some-loopin } X x B) (\text{some-loopin } X x C))) =$   
 $\text{loopin-class } X x$   
 $(\text{joinpathin } (\text{joinpathin } (\text{some-loopin } X x A) (\text{some-loopin } X x B)) (\text{some-loopin } X x C))$   
**by** (rule *loopin-class-eqI[OF left-loop right-loop hom-assoc]*)  
**show** *?thesis*  
**using** *left-eq right-eq class-eq* **by** *simp*  
**qed**

**lemma** *fundamental-groupin-mult-inv-right*:  
**assumes** *x-in*:  $x \in \text{topspace } X$   
**and** *A-in*:  $A \in \text{fundamental-groupin-space } X x$   
**shows**  $\text{fundamental-groupin-mult } X x A (\text{fundamental-groupin-inv } X x A) = \text{fundamental-groupin-one } X x$   
**proof** –  
**have** *const*:  $(\lambda-. x) \in \text{loopin-space } X x$   
**by** (rule *constant-loopin-in-space[OF x-in]*)  
**have** *repA*:  $\text{some-loopin } X x A \in \text{loopin-space } X x A = \text{loopin-class } X x$   
 $(\text{some-loopin } X x A)$   
**using** *some-loopin-spec[OF A-in]* **by** *auto*

**have** *loopA*: *pathin*  $X$  (*some-loopin*  $X$   $x$   $A$ ) *some-loopin*  $X$   $x$   $A$   $0 = x$  *some-loopin*  $X$   $x$   $A$   $1 = x$   
**using** *repA*(1) **by** (*auto elim*: *loopin-spaceE*)  
**have** *inv-eq*:  
*fundamental-groupin-inv*  $X$   $x$   $A =$  *loopin-class*  $X$   $x$  (*reversepathin* (*some-loopin*  $X$   $x$   $A$ ))  
**by** (*rule fundamental-groupin-inv-eqI*[*OF A-in repA*(1) *repA*(2)])  
**have** *mult-eq*:  
*fundamental-groupin-mult*  $X$   $x$   $A$  (*fundamental-groupin-inv*  $X$   $x$   $A$ ) =  
*loopin-class*  $X$   $x$  (*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*reversepathin* (*some-loopin*  $X$   $x$   $A$ )))  
**proof** (*rule fundamental-groupin-mult-eqI*)  
**show**  $A \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule A-in*)  
**show** *fundamental-groupin-inv*  $X$   $x$   $A \in$  *fundamental-groupin-space*  $X$   $x$   
**by** (*rule fundamental-groupin-inv-in-space*[*OF A-in*])  
**show** *some-loopin*  $X$   $x$   $A \in$  *loopin-space*  $X$   $x$   
**by** (*rule repA*(1))  
**show** *reversepathin* (*some-loopin*  $X$   $x$   $A$ )  $\in$  *loopin-space*  $X$   $x$   
**by** (*rule loopin-space-reversepathin*[*OF repA*(1)])  
**show**  $A =$  *loopin-class*  $X$   $x$  (*some-loopin*  $X$   $x$   $A$ )  
**by** (*rule repA*(2))  
**show** *fundamental-groupin-inv*  $X$   $x$   $A =$  *loopin-class*  $X$   $x$  (*reversepathin* (*some-loopin*  $X$   $x$   $A$ ))  
**by** (*rule inv-eq*)  
**qed**  
**have** *join-loop*: *joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*reversepathin* (*some-loopin*  $X$   $x$   $A$ ))  $\in$  *loopin-space*  $X$   $x$   
**by** (*rule loopin-space-joinpathin*[*OF repA*(1) *loopin-space-reversepathin*[*OF repA*(1)]]])  
**have** *hom*:  
*homotopic-paths*  $X$   
(*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*reversepathin* (*some-loopin*  $X$   $x$   $A$ )))  
( $\lambda$ -.  $x$ )  
**using** *homotopic-paths-in-const*[*OF loopA*(1)] *loopA*(2) **by** *simp*  
**have** *class-eq*:  
*loopin-class*  $X$   $x$  (*joinpathin* (*some-loopin*  $X$   $x$   $A$ ) (*reversepathin* (*some-loopin*  $X$   $x$   $A$ ))) =  
*loopin-class*  $X$   $x$  ( $\lambda$ -.  $x$ )  
**by** (*rule loopin-class-eqI*[*OF join-loop const hom*])  
**show** *?thesis*  
**using** *mult-eq class-eq unfolding fundamental-groupin-one-def* **by** *simp*  
**qed**

**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 –
  have const:  $(\lambda-. x) \in \text{loopin-space } X x$ 
    by (rule constant-loopin-in-space[OF x-in])
  have repA:  $\text{some-loopin } X x A \in \text{loopin-space } X x A = \text{loopin-class } X x$ 
    ( $\text{some-loopin } X x A$ )
    using some-loopin-spec[OF A-in] by auto
  have loopA:  $\text{pathin } X (\text{some-loopin } X x A) \text{ some-loopin } X x A 0 = x \text{ some-loopin}$ 
     $X x A 1 = x$ 
    using repA(1) by (auto elim: loopin-spaceE)
  have inv-eq:
     $\text{fundamental-groupin-inv } X x A = \text{loopin-class } X x (\text{reversepathin } (\text{some-loopin}$ 
     $X x A))$ 
    by (rule fundamental-groupin-inv-eqI[OF A-in repA(1) repA(2)])
  have mult-eq:
     $\text{fundamental-groupin-mult } X x (\text{fundamental-groupin-inv } X x A) A =$ 
     $\text{loopin-class } X x (\text{joinpathin } (\text{reversepathin } (\text{some-loopin } X x A)) (\text{some-loopin}$ 
     $X x A))$ 
  proof (rule fundamental-groupin-mult-eqI)
  show  $\text{fundamental-groupin-inv } X x A \in \text{fundamental-groupin-space } X x$ 
    by (rule fundamental-groupin-inv-in-space[OF A-in])
  show  $A \in \text{fundamental-groupin-space } X x$ 
    by (rule A-in)
  show  $\text{reversepathin } (\text{some-loopin } X x A) \in \text{loopin-space } X x$ 
    by (rule loopin-space-reversepathin[OF repA(1)])
  show  $\text{some-loopin } X x A \in \text{loopin-space } X x$ 
    by (rule repA(1))
  show  $\text{fundamental-groupin-inv } X x A = \text{loopin-class } X x (\text{reversepathin}$ 
    ( $\text{some-loopin } X x A$ ))
    by (rule inv-eq)
  show  $A = \text{loopin-class } X x (\text{some-loopin } X x A)$ 
    by (rule repA(2))
  qed
  have join-loop:  $\text{joinpathin } (\text{reversepathin } (\text{some-loopin } X x A)) (\text{some-loopin } X x$ 
     $A) \in \text{loopin-space } X x$ 
    by (rule loopin-space-joinpathin[OF loopin-space-reversepathin[OF repA(1)]
    repA(1)])
  have hom:
     $\text{homotopic-pathsin } X$ 
    ( $\text{joinpathin } (\text{reversepathin } (\text{some-loopin } X x A)) (\text{some-loopin } X x A)$ )
    ( $\lambda-. x$ )
    using homotopic-pathsin-linv-const[OF loopA(1)] loopA(3) by simp
  have class-eq:
     $\text{loopin-class } X x (\text{joinpathin } (\text{reversepathin } (\text{some-loopin } X x A)) (\text{some-loopin}$ 
     $X x A)) =$ 
     $\text{loopin-class } X x (\lambda-. x)$ 
    by (rule loopin-class-eqI[OF join-loop const hom])
  show ?thesis
    using mult-eq class-eq unfolding fundamental-groupin-one-def by simp
  qed

```

```

lemma fundamental-groupin-carrier-group:
  assumes x-in:  $x \in \text{topspace } X$ 
  shows carrier-group
    (fundamental-groupin-space  $X x$ )
    (fundamental-groupin-mult  $X x$ )
    (fundamental-groupin-one  $X x$ )
    (fundamental-groupin-inv  $X x$ )
proof
  show fundamental-groupin-one  $X x \in \text{fundamental-groupin-space } X x$ 
    by (rule fundamental-groupin-one-in-space[OF x-in])
next
  fix  $A B$ 
  assume  $A \in \text{fundamental-groupin-space } X x$   $B \in \text{fundamental-groupin-space } X x$ 
  then show fundamental-groupin-mult  $X x A B \in \text{fundamental-groupin-space } X x$ 
    by (rule fundamental-groupin-mult-in-space)
next
  fix  $A$ 
  assume  $A \in \text{fundamental-groupin-space } X x$ 
  then show fundamental-groupin-inv  $X x A \in \text{fundamental-groupin-space } X x$ 
    by (rule fundamental-groupin-inv-in-space)
next
  fix  $A B C$ 
  assume A-in:  $A \in \text{fundamental-groupin-space } X x$ 
    and B-in:  $B \in \text{fundamental-groupin-space } X x$ 
    and C-in:  $C \in \text{fundamental-groupin-space } X x$ 
  show
    fundamental-groupin-mult  $X x (\text{fundamental-groupin-mult } X x A B) C =$ 
    fundamental-groupin-mult  $X x A (\text{fundamental-groupin-mult } X x B C)$ 
    by (rule sym, rule fundamental-groupin-mult-assoc[OF A-in B-in C-in])
next
  fix  $A$ 
  assume A-in:  $A \in \text{fundamental-groupin-space } X x$ 
  show fundamental-groupin-mult  $X x (\text{fundamental-groupin-one } X x) A = A$ 
    by (rule fundamental-groupin-mult-one-left[OF x-in A-in])
next
  fix  $A$ 
  assume A-in:  $A \in \text{fundamental-groupin-space } X x$ 
  show fundamental-groupin-mult  $X x A (\text{fundamental-groupin-one } X x) = A$ 
    by (rule fundamental-groupin-mult-one-right[OF x-in A-in])
next
  fix  $A$ 
  assume A-in:  $A \in \text{fundamental-groupin-space } X x$ 
  show fundamental-groupin-mult  $X x (\text{fundamental-groupin-inv } X x A) A = \text{fundamental-groupin-one } X x$ 
    by (rule fundamental-groupin-mult-inv-left[OF x-in A-in])
next
  fix  $A$ 

```

**assume**  $A\text{-in}$ :  $A \in \text{fundamental-groupin-space } X \ x$   
**show**  $\text{fundamental-groupin-mult } X \ x \ A \ (\text{fundamental-groupin-inv } X \ x \ A) = \text{fundamental-groupin-one } X \ x$   
**by** ( $\text{rule fundamental-groupin-mult-inv-right}[OF \ x\text{-in } A\text{-in}]$ )  
**qed**

**lemma**  $\text{fundamental-groupin-map-carrier-group-hom}$ :

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

**proof** –

**have**  $y\text{-in}$ :  $y \in \text{topspace } Y$   
**using**  $x\text{-in}$   $h$   $fx$  **unfolding**  $\text{continuous-map-def}$  **by**  $\text{blast}$   
**show**  $?thesis$

**proof** ( $\text{rule carrier-group-hom.intro}$ )

**show**  $\text{carrier-group}$   
 $(\text{fundamental-groupin-space } X \ x)$   
 $(\text{fundamental-groupin-mult } X \ x)$   
 $(\text{fundamental-groupin-one } X \ x)$   
 $(\text{fundamental-groupin-inv } X \ x)$   
**by** ( $\text{rule fundamental-groupin-carrier-group}[OF \ x\text{-in}]$ )

**show**  $\text{carrier-group}$   
 $(\text{fundamental-groupin-space } Y \ y)$   
 $(\text{fundamental-groupin-mult } Y \ y)$   
 $(\text{fundamental-groupin-one } Y \ y)$   
 $(\text{fundamental-groupin-inv } Y \ y)$   
**by** ( $\text{rule fundamental-groupin-carrier-group}[OF \ y\text{-in}]$ )

**show**  $\text{carrier-group-hom-axioms}$   
 $(\text{fundamental-groupin-space } X \ x)$   
 $(\text{fundamental-groupin-mult } X \ x)$   
 $(\text{fundamental-groupin-space } Y \ y)$   
 $(\text{fundamental-groupin-mult } Y \ y)$   
 $(\text{fundamental-groupin-map } X \ x \ Y \ y \ f)$

**proof** ( $\text{rule carrier-group-hom-axioms.intro}$ )

**fix**  $A$

**assume**  $A\text{-in}$ :  $A \in \text{fundamental-groupin-space } X \ x$

**show**  $\text{fundamental-groupin-map } X \ x \ Y \ y \ f \ A \in \text{fundamental-groupin-space } Y$

$y$

**by** ( $\text{rule fundamental-groupin-map-in-space}[OF \ A\text{-in } h \ fx]$ )

```

next
  fix A B
  assume A-in: A ∈ fundamental-groupin-space X x
  and B-in: B ∈ fundamental-groupin-space X x
  show 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)
  by (rule fundamental-groupin-map-mult[OF A-in B-in h fx])
qed
qed
qed

```

```

lemma loopin-space-top-of-set [simp]:
  loopin-space (top-of-set S) x = loop-space S x
  unfolding loopin-space-def loop-space-def
  by (auto simp: pathin-canon-iff pathstart-def pathfinish-def path-image-def im-
age-subset-iff-funcset)

```

```

lemma loopin-class-top-of-set [simp]:
  loopin-class (top-of-set S) x p = loop-class S x p
  unfolding loopin-class-def loop-class-def by simp

```

```

lemma fundamental-groupin-space-top-of-set [simp]:
  fundamental-groupin-space (top-of-set S) x = fundamental-group-space S x
  unfolding fundamental-groupin-space-def fundamental-group-space-def by simp

```

```

lemma fundamental-groupin-one-top-of-set [simp]:
  fundamental-groupin-one (top-of-set S) x = fundamental-group-one S x
  unfolding fundamental-groupin-one-def fundamental-group-one-def by simp

```

```

lemma loopin-image-eq-loop-image [simp]:
  loopin-image h p = loop-image h p
  unfolding loopin-image-def loop-image-def by simp

```

```

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
  show pushout-rel f g x x for x
    by (rule pushout-rel.refl)
next
  show pushout-rel f g x y ==> pushout-rel f g y x for x y
    by (rule pushout-rel.sym)
next
  show pushout-rel f g x y ==> pushout-rel f g y z ==> pushout-rel f g x z for x
    y z
    by (rule pushout-rel.trans)
qed

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  $\longleftrightarrow$  pushout-rel f g x y
unfolding pushout-class-def
by (rule pushout-equiv.equiv-class-eq-iff)

lemma pushout-inl-in-space [intro]:
  pushout-inl f g a  $\in$  pushout-space f g
unfolding pushout-inl-def pushout-space-def pushout-class-def quotient-space-def

```

by *blast*

**lemma** *pushout-inr-in-space* [*intro*]:

*pushout-inr f g b ∈ pushout-space f g*

**unfolding** *pushout-inr-def pushout-space-def pushout-class-def quotient-space-def*

by *blast*

**lemma** *pushout-glue*:

*pushout-inl f g (f c) = pushout-inr f g (g c)*

by (*simp add: pushout-class-eq-iff pushout-inl-def pushout-inr-def pushout-rel.glue*)

**definition** *pushout-case-compatible* ::

$(c \Rightarrow a) \Rightarrow (c \Rightarrow b) \Rightarrow (a \Rightarrow d) \Rightarrow (b \Rightarrow d) \Rightarrow \text{bool}$

where

*pushout-case-compatible f g left right*  $\longleftrightarrow (\forall c. \text{left } (f c) = \text{right } (g c))$

**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*

**using** *rel*

**proof** (*induction rule: pushout-rel.induct*)

**case** (*refl x*)

**then show** *?case* by *simp*

**next**

**case** (*sym x y*)

**then show** *?case* by *simp*

**next**

**case** (*trans x y z*)

**then show** *?case* by *simp*

**next**

**case** (*glue c*)

**then show** *?case*

**using** *compat unfolding pushout-case-compatible-def* by *simp*

**qed**

**definition** *pushout-rec* ::

$(c \Rightarrow a) \Rightarrow (c \Rightarrow b) \Rightarrow (a \Rightarrow d) \Rightarrow (b \Rightarrow d) \Rightarrow (a + b) \text{ set} \Rightarrow d$

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**  $\exists! z. \exists x. X = \text{pushout-class } f g x \wedge z = \text{case-sum left right } x$

**proof** –

```

from  $X$ -in obtain  $rep$  where  $X$ -rep:  $X = pushout-class\ f\ g\ rep$ 
  unfolding  $pushout-space-def\ quotient-space-def\ pushout-class-def$  by  $blast$ 
show  $?thesis$ 
proof (rule  $exI[of - case-sum\ left\ right\ rep]$ )
  show  $\exists x. X = pushout-class\ f\ g\ x \wedge case-sum\ left\ right\ rep = case-sum\ left$ 
 $right\ x$ 
  by (rule  $exI[where\ x = rep]$ ,  $simp\ add: X-rep$ )
next
fix  $z$ 
assume  $hz: \exists x. X = pushout-class\ f\ g\ x \wedge z = case-sum\ left\ right\ x$ 
from  $hz$  obtain  $x$  where
   $x: X = pushout-class\ f\ g\ x\ z = case-sum\ left\ right\ x$ 
  by  $blast$ 
have  $pushout-rel\ f\ g\ x\ rep$ 
  using  $X-rep\ x(1)$  by ( $simp\ add: pushout-class-eq-iff$ )
then have  $rel-rep-x: pushout-rel\ f\ g\ rep\ x$ 
  by (rule  $pushout-rel.sym$ )
from  $compat\ rel-rep-x$  have  $case-sum\ left\ right\ rep = case-sum\ left\ right\ x$ 
  by (rule  $pushout-rel-case-cong$ )
with  $x$  show  $z = case-sum\ left\ right\ rep$ 
  by  $simp$ 
qed
qed

```

```

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 -
  have  $uniq$ :
     $\exists! z. \exists x. pushout-inl\ f\ g\ a = pushout-class\ f\ g\ x \wedge z = case-sum\ left\ right\ x$ 
    using  $pushout-rec-well-defined[OF\ compat\ pushout-inl-in-space[of\ f\ g\ a]]$  .
  have  $witness$ :
     $\exists x. pushout-inl\ f\ g\ a = pushout-class\ f\ g\ x \wedge left\ a = case-sum\ left\ right\ x$ 
    unfolding  $pushout-inl-def$  by ( $intro\ exI[of - Inl\ a]$ )  $simp$ 
  show  $?thesis$ 
  unfolding  $pushout-rec-def$ 
  by (rule  $the1-equality[OF\ uniq\ witness]$ )
qed

```

```

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 -
  have  $uniq$ :
     $\exists! z. \exists x. pushout-inr\ f\ g\ b = pushout-class\ f\ g\ x \wedge z = case-sum\ left\ right\ x$ 
    using  $pushout-rec-well-defined[OF\ compat\ pushout-inr-in-space[of\ f\ g\ b]]$  .
  have  $witness$ :
     $\exists x. pushout-inr\ f\ g\ b = pushout-class\ f\ g\ x \wedge right\ b = case-sum\ left\ right\ x$ 
    unfolding  $pushout-inr-def$  by ( $intro\ exI[of - Inr\ b]$ )  $simp$ 

```

```

show ?thesis
  unfolding pushout-rec-def
  by (rule the1-equality[OF uniq witness])
qed

```

```

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
  unfolding svk-quotient-map-def by (rule full-amalg-class-in-space)

```

```

lemma svk-quotient-map-injective:
  assumes svk-quotient-map x = svk-quotient-map y
  shows x = y
proof –
  have full-amalg-equiv i1 i2 (encode x) (encode y)
  using assms
  unfolding svk-quotient-map-def
  by (simp add: full-amalg-class-eq-iff)
  then have decode (encode x) = decode (encode y)

```

```

    by (rule decode-respects)
  then show ?thesis
    by (simp add: decode-encode)
qed

```

**lemma** *svk-quotient-map-surjective*:

```

  assumes  $Q \in \text{full-amalgamated-free-product-space } i1 \ i2$ 
  shows  $\exists x. \text{svk-quotient-map } x = Q$ 
proof -
  from assms obtain  $w$  where  $Q\text{-rep}: Q = \text{full-amalg-class } i1 \ i2 \ w$ 
    unfolding full-amalgamated-free-product-space-def quotient-space-def
    full-amalg-class-def
    by blast
  have  $\text{svk-quotient-map } (\text{decode } w) = Q$ 
    unfolding svk-quotient-map-def  $Q\text{-rep}$ 
    by (simp add: full-amalg-class-eq-iff encode-decode)
  then show ?thesis
    by blast
qed

```

**theorem** *seifert-van-kampen-bij-betw*:

```

  bij-betw svk-quotient-map UNIV (full-amalgamated-free-product-space  $i1 \ i2$ )
  unfolding bij-betw-def
proof
  show inj-on svk-quotient-map UNIV
    unfolding inj-on-def
    using svk-quotient-map-injective by blast
next
  show svk-quotient-map ' $UNIV = \text{full-amalgamated-free-product-space } i1 \ i2$ '
  proof
    show svk-quotient-map ' $UNIV \subseteq \text{full-amalgamated-free-product-space } i1 \ i2$ '
      using svk-quotient-map-in-space by blast
    next
    show  $\text{full-amalgamated-free-product-space } i1 \ i2 \subseteq \text{svk-quotient-map } 'UNIV$ 
      using svk-quotient-map-surjective by blast
  qed
qed

```

**end**

**locale** *carrier-svk-encode-decode-interface* =

```

  fixes  $G1 :: 'a \text{ set}$ 
    and  $G2 :: 'b \text{ set}$ 
    and  $H :: 'h \text{ set}$ 
    and  $i1 :: 'h \Rightarrow 'a$ 
    and  $i2 :: 'h \Rightarrow 'b$ 
    and  $\text{mult1} :: 'a \Rightarrow 'a \Rightarrow 'a$ 
    and  $\text{one1} :: 'a$ 
    and  $\text{mult2} :: 'b \Rightarrow 'b \Rightarrow 'b$ 

```

```

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
unfolding carrier-svk-quotient-map-def
by (rule carrier-full-amalg-class-in-space[OF encode-in-space])

```

```

lemma carrier-svk-quotient-map-injective:
assumes carrier-svk-quotient-map x = carrier-svk-quotient-map y
shows x = y
proof –
  have carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (encode x)
(encode y)
  using assms
  unfolding carrier-svk-quotient-map-def
  by (simp add: carrier-full-amalg-class-eq-iff)
  then have decode (encode x) = decode (encode y)
  by (rule decode-respects)
  then show ?thesis
  by (simp add: decode-encode)
qed

```

```

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 –

```

```

from assms obtain w where
  w ∈ carrier-fpw-space G1 G2
  and Q-rep: Q = carrier-full-amalg-class G1 G2 H i1 i2 mult1 one1 mult2 one2
w
  unfolding carrier-full-amalgamated-free-product-space-def
  by blast
  have carrier-svk-quotient-map (decode w) = Q
  unfolding carrier-svk-quotient-map-def Q-rep
  by (simp add: carrier-full-amalg-class-eq-iff encode-decode)
  then show ?thesis
  by blast
qed

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)
  unfolding bij-betw-def
proof
  show inj-on carrier-svk-quotient-map UNIV
  unfolding inj-on-def
  using carrier-svk-quotient-map-injective by blast
next
  show carrier-svk-quotient-map ‘ UNIV =
  carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2
one2
  proof
  show carrier-svk-quotient-map ‘ UNIV ⊆
  carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2
one2
  using carrier-svk-quotient-map-in-space by blast
  next
  show carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1
mult2 one2
   $\subseteq$  carrier-svk-quotient-map ‘ UNIV
  using carrier-svk-quotient-map-surjective by blast
  qed
qed

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*  $p \subseteq S$   
**shows** *pathin* (*top-of-set*  $S$ )  $p$   
**using** *assms*  
**by** (*auto simp: pathin-canon-iff path-image-def image-subset-iff-funcset*)

**locale** *classical-svk-setup* =

**fixes**  $U :: 'a::\text{topological-space set}$   
**and**  $V :: 'a \text{ set}$   
**and**  $x0 :: 'a$   
**assumes** *U-open*: *open*  $U$   
**and** *V-open*: *open*  $V$   
**and** *x0-in*:  $x0 \in U \cap V$   
**and** *UV-path-connected*: *path-connected* ( $U \cap V$ )

**begin**

**abbreviation**  $W$  **where**  $W \equiv U \cup V$

**abbreviation**  $G1$  **where**  $G1 \equiv \text{fundamental-group-space } U \ x0$

**abbreviation**  $G2$  **where**  $G2 \equiv \text{fundamental-group-space } V \ x0$

**abbreviation**  $H$  **where**  $H \equiv \text{fundamental-group-space } (U \cap V) \ x0$

**abbreviation**  $\text{mult}1$  **where**  $\text{mult}1 \equiv \text{fundamental-group-mult } U \ x0$

**abbreviation**  $\text{one}1$  **where**  $\text{one}1 \equiv \text{fundamental-group-one } U \ x0$

**abbreviation**  $\text{mult}2$  **where**  $\text{mult}2 \equiv \text{fundamental-group-mult } V \ x0$

**abbreviation**  $\text{one}2$  **where**  $\text{one}2 \equiv \text{fundamental-group-one } V \ x0$

**abbreviation**  $\text{mult}W$  **where**  $\text{mult}W \equiv \text{fundamental-group-mult } W \ x0$

**abbreviation**  $\text{one}W$  **where**  $\text{one}W \equiv \text{fundamental-group-one } W \ x0$

**abbreviation**  $i1$  **where**  $i1 \equiv \text{fundamental-group-map } (U \cap V) \ x0 \ U \ x0 \ \text{id}$

**abbreviation**  $i2$  **where**  $i2 \equiv \text{fundamental-group-map } (U \cap V) \ x0 \ V \ x0 \ \text{id}$

**abbreviation**  $j1$  **where**  $j1 \equiv \text{fundamental-group-map } U \ x0 \ W \ x0 \ \text{id}$

**abbreviation**  $j2$  **where**  $j2 \equiv \text{fundamental-group-map } V \ x0 \ W \ x0 \ \text{id}$

### 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*]:  $x0 \in U$   
using *x0-in* by *blast*

**lemma** *x0-in-V* [*simp*]:  $x0 \in V$   
using *x0-in* by *blast*

**lemma** *x0-in-W* [*simp*]:  $x0 \in W$   
using *x0-in* by *blast*

**lemma** *x0-in-UV* [*simp*]:  $x0 \in U \cap V$   
using *x0-in* by *blast*

**lemma** *i1-in-G1*:  
assumes  $h \in H$   
shows  $i1\ h \in G1$   
by (rule *fundamental-group-map-in-space*[*OF assms*]) *auto*

**lemma** *i2-in-G2*:  
assumes  $h \in H$   
shows  $i2\ h \in G2$   
by (rule *fundamental-group-map-in-space*[*OF assms*]) *auto*

**lemma** *union-fundamental-group-maps-agree*:

assumes *h-in*:  $h \in H$   
shows  $j1\ (i1\ h) = j2\ (i2\ h)$

**proof** –

have *left-comp*:  
*fundamental-group-map*  $U\ x0\ W\ x0\ id$   
 $(\text{fundamental-group-map } (U \cap V)\ x0\ U\ x0\ id\ h) =$   
 $\text{fundamental-group-map } (U \cap V)\ x0\ W\ x0\ (id \circ id)\ h$   
by (rule *fundamental-group-map-compose*[*OF h-in*]) *auto*

have *right-comp*:  
*fundamental-group-map*  $V\ x0\ W\ x0\ id$   
 $(\text{fundamental-group-map } (U \cap V)\ x0\ V\ x0\ id\ h) =$   
 $\text{fundamental-group-map } (U \cap V)\ x0\ W\ x0\ (id \circ id)\ h$   
by (rule *fundamental-group-map-compose*[*OF h-in*]) *auto*

show *?thesis*  
using *left-comp right-comp* by *simp*

**qed**

**lemma** *decode-locale*:

*carrier-full-amalg-word-eval*

$G1\ mult1\ one1\ (\text{fundamental-group-inv } U\ x0)$

$G2\ mult2\ one2\ (\text{fundamental-group-inv } V\ x0)$

$H\ i1\ i2$

$(\text{fundamental-group-space } W\ x0)\ multW\ oneW\ (\text{fundamental-group-inv } W\ x0)$

$j1\ j2$

**proof** (rule *carrier-full-amalg-word-eval.intro*)

```

show carrier-group
  (fundamental-group-space U x0)
  (fundamental-group-mult U x0)
  (fundamental-group-one U x0)
  (fundamental-group-inv U x0)
  by (rule fundamental-group-carrier-group[OF x0-in-U])
show carrier-group
  (fundamental-group-space V x0)
  (fundamental-group-mult V x0)
  (fundamental-group-one V x0)
  (fundamental-group-inv V x0)
  by (rule fundamental-group-carrier-group[OF x0-in-V])
show carrier-group
  (fundamental-group-space W x0)
  (fundamental-group-mult W x0)
  (fundamental-group-one W x0)
  (fundamental-group-inv W x0)
  by (rule fundamental-group-carrier-group[OF x0-in-W])
show carrier-group-hom
  (fundamental-group-space U x0)
  (fundamental-group-mult U x0)
  (fundamental-group-one U x0)
  (fundamental-group-inv U x0)
  (fundamental-group-space W x0)
  (fundamental-group-mult W x0)
  (fundamental-group-one W x0)
  (fundamental-group-inv W x0)
  (fundamental-group-map U x0 W x0 id)
  by (rule fundamental-group-map-carrier-group-hom[OF x0-in-U]) auto
show carrier-group-hom
  (fundamental-group-space V x0)
  (fundamental-group-mult V x0)
  (fundamental-group-one V x0)
  (fundamental-group-inv V x0)
  (fundamental-group-space W x0)
  (fundamental-group-mult W x0)
  (fundamental-group-one W x0)
  (fundamental-group-inv W x0)
  (fundamental-group-map V x0 W x0 id)
  by (rule fundamental-group-map-carrier-group-hom[OF x0-in-V]) auto
show carrier-full-amalg-word-eval-axioms G1 G2 H i1 i2 j1 j2
proof (rule carrier-full-amalg-word-eval-axioms.intro)
  show  $h \in H \implies i1\ h \in G1$  for  $h$ 
    by (rule i1-in-G1)
  show  $h \in H \implies i2\ h \in G2$  for  $h$ 
    by (rule i2-in-G2)
  show  $h \in H \implies j1\ (i1\ h) = j2\ (i2\ h)$  for  $h$ 
    by (rule union-fundamental-group-maps-agree)
qed

```

qed

**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*

**by** (*rule decode-locale*)

**abbreviation** *svk-word-eval* **where** *svk-word-eval*  $\equiv$  *decode.carrier-full-amalg-eval*

**abbreviation** *svk-decode* **where** *svk-decode*  $\equiv$  *decode.carrier-full-amalg-decode*

**lemma** *svk-decode-in-space*:

*svk-decode w*  $\in$  *fundamental-group-space W x0*

**by** (*rule decode.carrier-full-amalg-decode-in-carrier*)

**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*

**using** *assms* **by** (*rule decode.carrier-full-amalg-decode-respects*)

**lemma** *svk-decode-eq-eval*:

**assumes** *fpw-in-space G1 G2 w*

**shows** *svk-decode w* = *svk-word-eval w*

**using** *assms* **by** (*rule decode.carrier-full-amalg-decode-eq-eval*)

**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*  $\in$  *G1*

**shows** *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*

(*WordLeft a u*) (*WordLeft a v*)

**using** *rel a-in*

**proof** (*induction rule: carrier-full-amalg-equiv.induct*)

**case** (*refl w*)

**then show** *?case* **by** *simp*

**next**

**case** (*sym u v*)

**then show** *?case*

**by** (*meson carrier-full-amalg-equiv.sym*)

**next**

**case** (*trans u v w*)

**then show** *?case*

**by** (*meson carrier-full-amalg-equiv.trans*)

**next**

**case** (*of-amalg u v*)

**then show** *?case*

**by** (*meson carrier-amalgam-equiv.left-context carrier-full-amalg-equiv.of-amalg*)

**next**  
**case** (*of-reduction*  $u\ v$ )  
**then show** *?case*  
**by** (*meson carrier-fpw-reduction-left-context carrier-full-amalg-equiv.of-reduction*)  
**qed**

**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)*  
**using** *rel b-in*  
**proof** (*induction rule: carrier-full-amalg-equiv.induct*)  
**case** (*refl*  $w$ )  
**then show** *?case* **by** *simp*  
**next**  
**case** (*sym*  $u\ v$ )  
**then show** *?case*  
**by** (*meson carrier-full-amalg-equiv.sym*)  
**next**  
**case** (*trans*  $u\ v\ w$ )  
**then show** *?case*  
**by** (*meson carrier-full-amalg-equiv.trans*)  
**next**  
**case** (*of-amalg*  $u\ v$ )  
**then show** *?case*  
**by** (*meson carrier-amalgam-equiv.right-context carrier-full-amalg-equiv.of-amalg*)  
**next**  
**case** (*of-reduction*  $u\ v$ )  
**then show** *?case*  
**by** (*meson carrier-fpw-reduction-right-context carrier-full-amalg-equiv.of-reduction*)  
**qed**

**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** –

**have** *step-left*:  
*carrier-fpw-reduction-step*  $G1\ G2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $a\ (WordLeft\ b\ rest)$ )  
(*WordLeft*  $(mult1\ a\ b)\ rest$ )  
**proof** (*rule carrier-fpw-reduction-step.combine-left*)  
**show**  $a \in G1$  **by** (*rule a-in*)  
**show**  $b \in G1$  **by** (*rule b-in*)  
**show**  $mult1\ a\ b \in G1$  **by** (*rule ab-in*)  
**show** *fpw-in-space*  $G1\ G2\ rest$  **by** (*rule rest-in*)  
**qed**  
**have** *red-left*:  
*carrier-fpw-reduction*  $G1\ G2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $a\ (WordLeft\ b\ rest)$ )  
(*WordLeft*  $(mult1\ a\ b)\ rest$ )  
**by** (*rule carrier-fpw-reduction.step[OF step-left]*)  
**have** *step-right*:  
*carrier-fpw-reduction-step*  $G1\ G2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $c\ (WordLeft\ d\ rest)$ )  
(*WordLeft*  $(mult1\ c\ d)\ rest$ )  
**proof** (*rule carrier-fpw-reduction-step.combine-left*)  
**show**  $c \in G1$  **by** (*rule c-in*)  
**show**  $d \in G1$  **by** (*rule d-in*)  
**show**  $mult1\ c\ d \in G1$  **by** (*rule cd-in*)  
**show** *fpw-in-space*  $G1\ G2\ rest$  **by** (*rule rest-in*)  
**qed**  
**have** *red-right*:  
*carrier-fpw-reduction*  $G1\ G2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $c\ (WordLeft\ d\ rest)$ )  
(*WordLeft*  $(mult1\ c\ d)\ rest$ )  
**by** (*rule carrier-fpw-reduction.step[OF step-right]*)  
**have** *rel-left*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $a\ (WordLeft\ b\ rest)$ )  
(*WordLeft*  $(mult1\ a\ b)\ rest$ )  
**by** (*rule carrier-full-amalg-equiv.of-reduction[OF red-left]*)  
**have** *rel-right*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
(*WordLeft*  $c\ (WordLeft\ d\ rest)$ )  
(*WordLeft*  $(mult1\ a\ b)\ rest$ )  
**using** *eq* **by** (*simp add: carrier-full-amalg-equiv.of-reduction[OF red-right]*)  
**show** *?thesis*  
**by** (*rule carrier-full-amalg-equiv.trans[OF rel-left]*)  
(*rule carrier-full-amalg-equiv.sym[OF rel-right]*)  
**qed**  
**lemma** *carrier-full-amalg-equiv-right-pair-eq*:  
**assumes** *a-in*:  $a \in G2$   
**and** *b-in*:  $b \in G2$   
**and** *ab-in*:  $mult2\ a\ b \in G2$

```

and c-in:  $c \in G2$ 
and d-in:  $d \in G2$ 
and cd-in:  $\text{mult2 } c \ d \in G2$ 
and rest-in: fpw-in-space  $G1 \ G2 \ \text{rest}$ 
and eq:  $\text{mult2 } a \ b = \text{mult2 } c \ d$ 
shows carrier-full-amalg-equiv  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
  (WordRight  $a \ (\text{WordRight } b \ \text{rest})$ )
  (WordRight  $c \ (\text{WordRight } d \ \text{rest})$ )
proof –
  have step-left:
    carrier-fpw-reduction-step  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $a \ (\text{WordRight } b \ \text{rest})$ )
      (WordRight  $(\text{mult2 } a \ b) \ \text{rest}$ )
  proof (rule carrier-fpw-reduction-step.combine-right)
    show  $a \in G2$  by (rule a-in)
    show  $b \in G2$  by (rule b-in)
    show  $\text{mult2 } a \ b \in G2$  by (rule ab-in)
    show fpw-in-space  $G1 \ G2 \ \text{rest}$  by (rule rest-in)
  qed
  have red-left:
    carrier-fpw-reduction  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $a \ (\text{WordRight } b \ \text{rest})$ )
      (WordRight  $(\text{mult2 } a \ b) \ \text{rest}$ )
    by (rule carrier-fpw-reduction.step[OF step-left])
  have step-right:
    carrier-fpw-reduction-step  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $c \ (\text{WordRight } d \ \text{rest})$ )
      (WordRight  $(\text{mult2 } c \ d) \ \text{rest}$ )
  proof (rule carrier-fpw-reduction-step.combine-right)
    show  $c \in G2$  by (rule c-in)
    show  $d \in G2$  by (rule d-in)
    show  $\text{mult2 } c \ d \in G2$  by (rule cd-in)
    show fpw-in-space  $G1 \ G2 \ \text{rest}$  by (rule rest-in)
  qed
  have red-right:
    carrier-fpw-reduction  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $c \ (\text{WordRight } d \ \text{rest})$ )
      (WordRight  $(\text{mult2 } c \ d) \ \text{rest}$ )
    by (rule carrier-fpw-reduction.step[OF step-right])
  have rel-left:
    carrier-full-amalg-equiv  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $a \ (\text{WordRight } b \ \text{rest})$ )
      (WordRight  $(\text{mult2 } a \ b) \ \text{rest}$ )
    by (rule carrier-full-amalg-equiv.of-reduction[OF red-left])
  have rel-right:
    carrier-full-amalg-equiv  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$ 
      (WordRight  $c \ (\text{WordRight } d \ \text{rest})$ )
      (WordRight  $(\text{mult2 } a \ b) \ \text{rest}$ )
    using eq by (simp add: carrier-full-amalg-equiv.of-reduction[OF red-right])

```

**show** ?thesis  
**by** (rule carrier-full-amalg-equiv.trans[OF rel-left])  
(rule carrier-full-amalg-equiv.sym[OF rel-right])  
**qed**

**lemma** loop-subdivision-by-cover:  
**assumes** p-loop:  $p \in \text{loop-space } W \ x0$   
**shows**  $\exists n::\text{nat. } 0 < n \wedge$   
 $(\forall i < n.$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) p \text{ ' } \{0..1\} \subseteq U \vee$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) p \text{ ' } \{0..1\} \subseteq V)$

**proof** –  
**have** p-path: path p  
**using** p-loop unfolding loop-space-def **by** auto  
**have** p-img: path-image p  $\subseteq W$   
**using** p-loop unfolding loop-space-def **by** auto  
**have** p-pathin: pathin (top-of-set W) p  
**by** (rule path-top-of-setI[OF p-path p-img])  
**have** cover:  $p \text{ ' } \{0..1\} \subseteq \bigcup \{U, V\}$   
**using** p-img **by** (auto simp: path-image-def)  
**have** open-cover: openin (top-of-set W) S **if**  $S \in \{U, V\}$  **for** S  
**using** that U-open V-open **by** (auto simp: openin-open)  
**from** pathin-subdivision-open-cover[OF p-pathin cover open-cover]  
**obtain** n :: nat **where** n-pos:  $0 < n$   
**and** n-cover:  
 $\forall i < n. \exists S \in \{U, V\}.$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) p \text{ ' } \{0..1\} \subseteq S$   
**by** blast  
**show** ?thesis  
**using** n-pos n-cover **by** auto  
**qed**

**definition** connector :: 'a  $\Rightarrow$  real  $\Rightarrow$  'a **where**  
connector a =  
(if a = x0 then ( $\lambda$ -. x0)  
else (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p = x0  $\wedge$  pathfinish p = a))

**lemma** connector-x0 [simp]:  
connector x0 = ( $\lambda$ -. x0)  
**unfolding** connector-def **by** simp

**lemma** connector-witness:  
**assumes** a-in:  $a \in U \cap V$   
**shows**  $\exists p. \text{path } p \wedge \text{path-image } p \subseteq U \cap V \wedge \text{pathstart } p = x0 \wedge \text{pathfinish } p = a$   
**proof** (cases a = x0)  
**case** True  
**have** path ( $\lambda$ -. x0)  $\wedge$

```

    path-image ( $\lambda-. x0$ )  $\subseteq U \cap V \wedge$ 
    pathstart ( $\lambda-. x0$ ) =  $x0 \wedge$ 
    pathfinish ( $\lambda-. x0$ ) =  $a$ 
    using a-in True by (auto simp: path-def path-image-def pathstart-def pathfin-
ish-def)
    then show ?thesis by blast
next
case False
then show ?thesis
using UV-path-connected x0-in-UV a-in unfolding path-connected-def by blast
qed

```

```

lemma connector-path:
  assumes a-in:  $a \in U \cap V$ 
  shows path (connector a)
proof (cases a = x0)
  case True
  then show ?thesis
    by (simp add: connector-def path-def)
next
  case False
  have some:
    path (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$  pathfinish
p = a)  $\wedge$ 
    path-image (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$ 
pathfinish p = a)  $\subseteq U \cap V \wedge$ 
    pathstart (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$ 
pathfinish p = a) =  $x0 \wedge$ 
    pathfinish (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$ 
pathfinish p = a) = a
  by (rule someI-ex[OF connector-witness[OF a-in]])
  then show ?thesis
    using False by (simp add: connector-def)
qed

```

```

lemma connector-image-subset:
  assumes a-in:  $a \in U \cap V$ 
  shows path-image (connector a)  $\subseteq U \cap V$ 
proof (cases a = x0)
  case True
  then show ?thesis
    using a-in by (auto simp: connector-def path-image-def)
next
  case False
  have some:
    path (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$  pathfinish
p = a)  $\wedge$ 
    path-image (SOME p. path p  $\wedge$  path-image p  $\subseteq U \cap V \wedge$  pathstart p =  $x0 \wedge$ 
pathfinish p = a)  $\subseteq U \cap V \wedge$ 

```

```

      pathstart (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = x0 ∧
      pathfinish (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = a
    by (rule someI-ex[OF connector-witness[OF a-in]])
    then show ?thesis
    using False by (simp add: connector-def)
qed

```

**lemma** *connector-start*:

```

  assumes a-in: a ∈ U ∩ V
  shows pathstart (connector a) = x0
proof (cases a = x0)
  case True
  then show ?thesis
    by (simp add: connector-def pathstart-def)
next
  case False
  have some:
    path (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧ pathfinish
p = a) ∧
    path-image (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) ⊆ U ∩ V ∧
    pathstart (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = x0 ∧
    pathfinish (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = a
    by (rule someI-ex[OF connector-witness[OF a-in]])
  then show ?thesis
  using False by (simp add: connector-def)
qed

```

**lemma** *connector-finish*:

```

  assumes a-in: a ∈ U ∩ V
  shows pathfinish (connector a) = a
proof (cases a = x0)
  case True
  then show ?thesis
    by (simp add: connector-def pathfinish-def)
next
  case False
  have some:
    path (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧ pathfinish
p = a) ∧
    path-image (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) ⊆ U ∩ V ∧
    pathstart (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = x0 ∧
    pathfinish (SOME p. path p ∧ path-image p ⊆ U ∩ V ∧ pathstart p = x0 ∧
pathfinish p = a) = a

```

$\text{pathfinish } p = a) = a$   
**by** (rule *someI-ex*[*OF connector-witness*[*OF a-in*]])  
**then show** *?thesis*  
**using** *False* **by** (*simp add: connector-def*)  
**qed**

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

**definition** *segment-loop* :: (*real*  $\Rightarrow$  'a)  $\Rightarrow$  *real*  $\Rightarrow$  *real*  $\Rightarrow$  *real*  $\Rightarrow$  '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*  $\subseteq$  *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** *conn-u*: *path-image* (*connector* (*p u*))  $\subseteq$  *S*  
**and** *conn-v*: *path-image* (*connector* (*p v*))  $\subseteq$  *S*  
**and** *seg-in*: *subpathin* *u v p* ' {0..1}  $\subseteq$  *S*  
**and** *x0S*: *x0*  $\in$  *S*

**shows** *segment-loop p u v*  $\in$  *loop-space S x0*

**proof** –

**have** *cu-path*: *path* (*connector* (*p u*))  
**using** *puv-in connector-spec(1)*[*of p u*] **by** *blast*  
**have** *cv-path*: *path* (*connector* (*p v*))  
**using** *puv-in connector-spec(1)*[*of p v*] **by** *blast*  
**have** *cu-start*: *pathstart* (*connector* (*p u*)) = *x0*  
**using** *puv-in connector-spec(3)*[*of p u*] **by** *blast*  
**have** *cu-finish*: *pathfinish* (*connector* (*p u*)) = *p u*  
**using** *puv-in connector-spec(4)*[*of p u*] **by** *blast*  
**have** *cv-start*: *pathstart* (*connector* (*p v*)) = *x0*  
**using** *puv-in connector-spec(3)*[*of p v*] **by** *blast*  
**have** *cv-finish*: *pathfinish* (*connector* (*p v*)) = *p v*  
**using** *puv-in connector-spec(4)*[*of p v*] **by** *blast*  
**have** *p-pathin*: *pathin* (*top-of-set W*) *p*  
**by** (rule *path-top-of-setI*[*OF p-path p-image*])  
**have** *subpath-pathin*: *pathin* (*top-of-set W*) (*subpathin u v p*)  
**by** (rule *pathin-subpathin*[*OF p-pathin uv01*])  
**have** *subpath-path*: *path* (*subpathin u v p*)  
**using** *subpath-pathin* **by** (*simp add: pathin-canon-iff*)  
**have** *subpath-start*: *pathstart* (*subpathin u v p*) = *p u*  
**by** (*simp add: pathstart-def subpathin-def*)  
**have** *subpath-finish*: *pathfinish* (*subpathin u v p*) = *p v*  
**by** (*simp add: pathfinish-def subpathin-def*)  
**have** *first-join*: *path* (*connector* (*p u*) +++ *subpathin u v p*)  
**using** *cu-path subpath-path cu-finish subpath-start* **by** *simp*  
**have** *first-finish*: *pathfinish* (*connector* (*p u*) +++ *subpathin u v p*) = *p v*

```

    using first-join subpath-finish by simp
  have rev-cv-path: path (reversepath (connector (p v)))
    using cv-path by simp
  have rev-cv-start: pathstart (reversepath (connector (p v))) = p v
    using cv-finish by simp
  have second-join:
    path ((connector (p u) +++ subpathin u v p) +++ reversepath (connector (p
v)))
    using first-join rev-cv-path first-finish rev-cv-start by simp
  show ?thesis
  proof (rule loop-spaceI)
    show path (segment-loop p u v)
      using second-join unfolding segment-loop-def by simp
    show path-image (segment-loop p u v)  $\subseteq$  S
    proof -
      have seg-img: path-image (subpathin u v p)  $\subseteq$  S
        using seg-in by (simp add: path-image-def)
      have left-img: path-image (connector (p u) +++ subpathin u v p)  $\subseteq$  S
        by (rule subset-path-image-join[OF conn-u seg-img])
      have right-img: path-image (reversepath (connector (p v)))  $\subseteq$  S
        using conn-v by simp
      show ?thesis
        unfolding segment-loop-def by (rule subset-path-image-join[OF left-img
right-img])
    qed
  show pathstart (segment-loop p u v) = x0
    unfolding segment-loop-def using cu-start by simp
  show pathfinish (segment-loop p u v) = x0
    unfolding segment-loop-def using cv-start by simp
  qed
qed

```

```

lemma segment-loop-in-U:
  assumes p-path: path p
    and p-image: path-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}  $\subseteq$  U
  shows segment-loop p u v  $\in$  loop-space U x0
  proof (rule segment-loop-in-set[where S = U])
    show path p
      by (rule p-path)
    show path-image p  $\subseteq$  W
      by (rule p-image)
    show u  $\in$  {0..1} v  $\in$  {0..1}
      by (rule uv01)+
    show p u  $\in$  U  $\cap$  V p v  $\in$  U  $\cap$  V
      by (rule puv-in)+
    show path-image (connector (p u))  $\subseteq$  U

```

**using** *connector-spec(2)*[*OF puv-in(1)*] **by** *blast*  
**show** *path-image* (*connector* (*p v*))  $\subseteq U$   
**using** *connector-spec(2)*[*OF puv-in(2)*] **by** *blast*  
**show** *subpathin* *u v p* ‘  $\{0..1\} \subseteq U$   
**by** (*rule seg-in*)  
**show**  $x0 \in U$   
**by** *simp*  
**qed**

**lemma** *segment-loop-in-V*:  
**assumes** *p-path*: *path p*  
**and** *p-image*: *path-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* ‘  $\{0..1\} \subseteq V$   
**shows** *segment-loop* *p u v*  $\in$  *loop-space* *V x0*  
**proof** (*rule segment-loop-in-set*[**where**  $S = V$ ])  
**show** *path p*  
**by** (*rule p-path*)  
**show** *path-image p*  $\subseteq W$   
**by** (*rule p-image*)  
**show**  $u \in \{0..1\} v \in \{0..1\}$   
**by** (*rule uv01*)  
**show**  $p u \in U \cap V p v \in U \cap V$   
**by** (*rule puv-in*)  
**show** *path-image* (*connector* (*p u*))  $\subseteq V$   
**using** *connector-spec(2)*[*OF puv-in(1)*] **by** *blast*  
**show** *path-image* (*connector* (*p v*))  $\subseteq V$   
**using** *connector-spec(2)*[*OF puv-in(2)*] **by** *blast*  
**show** *subpathin* *u v p* ‘  $\{0..1\} \subseteq V$   
**by** (*rule seg-in*)  
**show**  $x0 \in V$   
**by** *simp*  
**qed**

**lemma** *segment-loop-in-W*:  
**assumes** *p-path*: *path p*  
**and** *p-image*: *path-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-loop* *p u v*  $\in$  *loop-space* *W x0*  
**proof** (*rule segment-loop-in-set*[**where**  $S = W$ ])  
**show** *path p*  
**by** (*rule p-path*)  
**show** *path-image p*  $\subseteq W$   
**by** (*rule p-image*)  
**show**  $u \in \{0..1\} v \in \{0..1\}$   
**by** (*rule uv01*)

```

show  $p\ u \in U \cap V\ p\ v \in U \cap V$ 
  by (rule puv-in)+
show  $\text{path-image } (\text{connector } (p\ u)) \subseteq W$ 
  using connector-spec(2)[OF puv-in(1)] by blast
show  $\text{path-image } (\text{connector } (p\ v)) \subseteq W$ 
  using connector-spec(2)[OF puv-in(2)] by blast
show  $\text{subpathin } u\ v\ p\ \{0..1\} \subseteq W$ 
  by (rule seg-in)
show  $x0 \in W$ 
  by simp
qed

```

```

lemma path-subpathin:
  assumes path p
    and  $u \in \{0..1\}$ 
    and  $v \in \{0..1\}$ 
  shows path (subpathin u v p)
proof –
  have pathin (top-of-set (path-image p)) p
    by (rule path-top-of-setI[OF assms(1)]) auto
  then have pathin (top-of-set (path-image p)) (subpathin u v p)
    by (rule pathin-subpathin[OF - assms(2,3)])
  then show ?thesis
    by (simp add: pathin-canon-iff)
qed

```

```

lemma path-image-subpathin-subset:
  assumes  $u \in \{0..1\}$ 
    and  $v \in \{0..1\}$ 
  shows  $\text{path-image } (\text{subpathin } u\ v\ p) \subseteq \text{path-image } p$ 
proof –
  have  $\text{closed-segment } u\ v \subseteq \{0..1\}$ 
    using assms by (auto simp: closed-segment-eq-real-ivl)
  then show ?thesis
    by (simp add: path-image-def subpathin-image image-mono)
qed

```

```

lemma reversepath-subpathin [simp]:
  reversepath (subpathin u v p) = subpathin v u p
  unfolding reversepath-def subpathin-def by (rule ext) (simp add: algebra-simps)

```

```

lemma subpathin-refl [simp]:
  subpathin u u p = ( $\lambda$ -. p u)
  unfolding subpathin-def by (rule ext) simp

```

```

fun svk-partition :: (real  $\Rightarrow$  'a)  $\Rightarrow$  real list  $\Rightarrow$  bool list  $\Rightarrow$  bool where
  svk-partition p [] bs = False
| svk-partition p [t] [] = (t = 1  $\wedge$  p t  $\in$  U  $\cap$  V)
| svk-partition p [t] (b # bs) = False

```

```

| svk-partition p (t # u # ts) [] = False
| svk-partition p (t # u # ts) (b # bs) =
  (t ∈ {0..1} ∧ p t ∈ U ∩ V ∧ u ∈ {0..1} ∧ t < u ∧
   (if b then subpathin t u p ‘ {0..1} ⊆ U else subpathin t u p ‘ {0..1} ⊆ V) ∧
   svk-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
by (induction p ts bs
  WordNil :: ((real ⇒ 'a) set, (real ⇒ 'a) set) free-product-word
  rule: partition-word-with-tail.induct) simp-all

lemma subpathin-joinpaths-left-half [simp]:
  subpathin 0 (1 / 2) (p +++ q) ' {0..1} = p ' {0..1}
proof
  show subpathin 0 (1 / 2) (p +++ q) ' {0..1} ⊆ p ' {0..1}
  by (auto simp: subpathin-def joinpaths-def)
next
  show p ' {0..1} ⊆ subpathin 0 (1 / 2) (p +++ q) ' {0..1}
  proof
    fix x
    assume x ∈ p ' {0..1}
    then obtain t where t01: t ∈ {0..1} and x-eq: x = p t
    by blast
    have x-eq': x = subpathin 0 (1 / 2) (p +++ q) t
    using x-eq t01 by (simp add: subpathin-def joinpaths-def)
    show x ∈ subpathin 0 (1 / 2) (p +++ q) ' {0..1}
    using t01 x-eq' by blast
  qed
qed

```

**lemma** affine-unit-interval:

```

fixes  $u\ v\ t :: \text{real}$ 
assumes  $u01: u \in \{0..1\}$ 
  and  $v01: v \in \{0..1\}$ 
  and  $t01: t \in \{0..1\}$ 
shows  $(v - u) * t + u \in \{0..1\}$ 
proof -
  have  $t\text{-bounds}: 0 \leq t \ t \leq 1$ 
    using  $t01$  by auto
  have  $u\text{-bounds}: 0 \leq u \ u \leq 1$ 
    using  $u01$  by auto
  have  $v\text{-bounds}: 0 \leq v \ v \leq 1$ 
    using  $v01$  by auto
  have  $one\text{-minus}\text{-}t\text{-nonneg}: 0 \leq 1 - t$ 
    using  $t\text{-bounds}$  by linarith
  have  $lower1: 0 \leq (1 - t) * u$ 
    using  $one\text{-minus}\text{-}t\text{-nonneg}$   $u\text{-bounds}$  by (intro mult-nonneg-nonneg) auto
  have  $lower2: 0 \leq t * v$ 
    using  $t\text{-bounds}$   $v\text{-bounds}$  by (intro mult-nonneg-nonneg) auto
  have  $lower: 0 \leq (1 - t) * u + t * v$ 
    using  $lower1$   $lower2$  by simp
  have  $upper1: (1 - t) * u \leq (1 - t) * 1$ 
    using  $one\text{-minus}\text{-}t\text{-nonneg}$   $u\text{-bounds}$  by (intro mult-left-mono) auto
  have  $upper2: t * v \leq t * 1$ 
    using  $t\text{-bounds}$   $v\text{-bounds}$  by (intro mult-left-mono) auto
  have  $upper: (1 - t) * u + t * v \leq 1$ 
    using  $upper1$   $upper2$  by simp
  have  $(v - u) * t + u = (1 - t) * u + t * v$ 
    by (simp add: algebra-simps)
  then show ?thesis
    using  $lower$   $upper$  by auto
qed

```

```

lemma subpathin-joinpaths-tail-scaled-pointwise:
  assumes  $q0: \text{pathstart } q = \text{pathfinish } p$ 
    and  $u01: u \in \{0..1\}$ 
    and  $v01: v \in \{0..1\}$ 
    and  $t01: t \in \{0..1\}$ 
shows  $\text{subpathin } ((1 + u) / 2) ((1 + v) / 2) (p +++ q) \ t = \text{subpathin } u \ v \ q \ t$ 
proof -
  let  $?s = (v - u) * t + u$ 
  have  $s01: ?s \in \{0..1\}$ 
    by (rule affine-unit-interval[OF u01 v01 t01])
  have param-eq:
     $((1 + v) / 2 - (1 + u) / 2) * t + (1 + u) / 2 = (1 + ?s) / 2$ 
    by (simp add: field-simps algebra-simps)
  show ?thesis
proof (cases ?s = 0)
  case True
    then show ?thesis

```

```

    using q0 by (simp add: subpathin-def joinpaths-def pathstart-def pathfinish-def
param-eq)
  next
  case False
  then have s-pos: 0 < ?s
    using s01 by auto
  then have param-gt:
    1 / 2 < ((1 + v) / 2 - (1 + u) / 2) * t + (1 + u) / 2
    by (simp add: param-eq)
  have param-not-le:
    ¬ (((1 + v) / 2 - (1 + u) / 2) * t + (1 + u) / 2 ≤ 1 / 2)
    using param-gt by simp
  then show ?thesis
  proof -
    have subpathin ((1 + u) / 2) ((1 + v) / 2) (p +++ q) t =
      q (2 * (((1 + v) / 2 - (1 + u) / 2) * t + (1 + u) / 2) - 1)
      using param-not-le by (simp add: subpathin-def joinpaths-def)
    also have ... = q ?s
    proof -
      have 2 * (((1 + v) / 2 - (1 + u) / 2) * t + (1 + u) / 2) - 1 = ?s
        by (simp add: field-simps algebra-simps)
      then show ?thesis
        by simp
    qed
    also have ... = subpathin u v q t
      by (simp add: subpathin-def)
    finally show ?thesis .
  qed
qed
qed

```

**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
    show subpathin ((1 + u) / 2) ((1 + v) / 2) (p +++ q) ‘ {0..1} ⊆ subpathin u
v q ‘ {0..1}
    proof
      fix x
      assume x-in: x ∈ subpathin ((1 + u) / 2) ((1 + v) / 2) (p +++ q) ‘ {0..1}
      then obtain t where t01: t ∈ {0..1}
        and x-eq: x = subpathin ((1 + u) / 2) ((1 + v) / 2) (p +++ q) t
        by blast
      have x-eq!: x = subpathin u v q t
        using x-eq subpathin-joinpaths-tail-scaled-pointwise[OF q0 u01 v01 t01] by
simp

```

```

    show  $x \in \text{subpathin } u \ v \ q \ ' \{0..1\}$ 
      using  $t01 \ x\text{-eq}'$  by blast
  qed
next
  show  $\text{subpathin } u \ v \ q \ ' \{0..1\} \subseteq \text{subpathin } ((1 + u) / 2) \ ((1 + v) / 2) \ (p \ +++ \ q) \ ' \{0..1\}$ 
  proof
    fix  $x$ 
    assume  $x\text{-in}: x \in \text{subpathin } u \ v \ q \ ' \{0..1\}$ 
    then obtain  $t$  where  $t01: t \in \{0..1\}$  and  $x\text{-eq}: x = \text{subpathin } u \ v \ q \ t$ 
      by blast
    have  $x\text{-eq}': x = \text{subpathin } ((1 + u) / 2) \ ((1 + v) / 2) \ (p \ +++ \ q) \ t$ 
      using  $x\text{-eq}$  subpathin-joinpaths-tail-scaled-pointwise[OF  $q0 \ u01 \ v01 \ t01$ ] by
  simp
    show  $x \in \text{subpathin } ((1 + u) / 2) \ ((1 + v) / 2) \ (p \ +++ \ q) \ ' \{0..1\}$ 
      using  $t01 \ x\text{-eq}'$  by blast
  qed
qed

lemma homotopic-paths-join-left:
  assumes  $qr: \text{homotopic-paths } S \ q \ r$ 
    and  $p\text{-path}: \text{path } p$ 
    and  $p\text{-img}: \text{path-image } p \subseteq S$ 
    and  $pq: \text{pathfinish } p = \text{pathstart } q$ 
  shows  $\text{homotopic-paths } S \ (p \ +++ \ q) \ (p \ +++ \ r)$ 
proof (rule homotopic-paths-join)
  show  $\text{homotopic-paths } S \ p \ p$ 
    using  $p\text{-path } p\text{-img}$  by simp
  show  $\text{homotopic-paths } S \ q \ r$ 
    by (rule  $qr$ )
  show  $\text{pathfinish } p = \text{pathstart } q$ 
    by (rule  $pq$ )
qed

lemma homotopic-paths-join-right:
  assumes  $pq: \text{homotopic-paths } S \ p \ q$ 
    and  $r\text{-path}: \text{path } r$ 
    and  $r\text{-img}: \text{path-image } r \subseteq S$ 
    and  $qr: \text{pathfinish } p = \text{pathstart } r$ 
  shows  $\text{homotopic-paths } S \ (p \ +++ \ r) \ (q \ +++ \ r)$ 
proof (rule homotopic-paths-join)
  show  $\text{homotopic-paths } S \ p \ q$ 
    by (rule  $pq$ )
  show  $\text{homotopic-paths } S \ r \ r$ 
    using  $r\text{-path } r\text{-img}$  by simp
  show  $\text{pathfinish } p = \text{pathstart } r$ 
    by (rule  $qr$ )
qed

```

**lemma** *homotopic-paths-cancel-middle-local*:

**assumes** *r-path*: path *r*  
**and** *r-img*: path-image  $r \subseteq S$   
**and** *c-path*: path *c*  
**and** *c-img*: path-image  $c \subseteq S$   
**and** *s-path*: path *s*  
**and** *s-img*: path-image  $s \subseteq S$   
**and** *rc*: pathfinish *r* = pathfinish *c*  
**and** *cs*: pathstart *s* = pathfinish *c*  
**shows** *homotopic-paths*  $S$  (((*r* +++ reversepath *c*) +++ *c*) +++ *s*) (*r* +++ *s*)

**proof** –

**have** *revc-path*: path (reversepath *c*)  
**using** *c-path* **by** simp  
**have** *revc-img*: path-image (reversepath *c*)  $\subseteq S$   
**using** *c-img* **by** simp  
**have** *mid-path*: path (reversepath *c* +++ *c*)  
**using** *revc-path* *c-path* **by** simp  
**have** *mid-img*: path-image (reversepath *c* +++ *c*)  $\subseteq S$   
**by** (rule subset-path-image-join[OF *revc-img* *c-img*])  
**have** *hom-cancel0*: *homotopic-paths*  $S$  (reversepath *c* +++ *c*) ( $\lambda$ -. pathfinish *c*)  
**by** (rule homotopic-paths-linv-const[OF *c-path* *c-img*])  
**have** *hom-cancel1*:  
*homotopic-paths*  $S$  (((reversepath *c* +++ *c*) +++ *s*) (( $\lambda$ -. pathfinish *c*) +++ *s*))

**proof** (rule homotopic-paths-join-right[OF *hom-cancel0* *s-path* *s-img*])  
**show** pathfinish (reversepath *c* +++ *c*) = pathstart *s*  
**using** *cs* **by** (simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def)

**qed**  
**have** *hom-cancel2*: *homotopic-paths*  $S$  (( $\lambda$ -. pathfinish *c*) +++ *s*) *s*  
**using** homotopic-paths-lid-const[OF *s-path* *s-img*] *cs* **by** (simp add: pathstart-def)

**have** *hom-cancel*: *homotopic-paths*  $S$  (((reversepath *c* +++ *c*) +++ *s*) *s*)  
**by** (rule homotopic-paths-trans[OF *hom-cancel1* *hom-cancel2*])  
**have** *hom-left*:  
*homotopic-paths*  $S$  (*r* +++ ((reversepath *c* +++ *c*) +++ *s*)) (*r* +++ *s*)

**proof** (rule homotopic-paths-join-left[OF *hom-cancel* *r-path* *r-img*])  
**show** pathfinish *r* = pathstart (((reversepath *c* +++ *c*) +++ *s*)  
**using** *rc* **by** (simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def)

**qed**  
**have** *assoc1*:  
*homotopic-paths*  $S$  (((*r* +++ reversepath *c*) +++ *c*) (*r* +++ (reversepath *c* +++ *c*)))

**proof** –  
**have** *homotopic-paths*  $S$  (*r* +++ (reversepath *c* +++ *c*)) (((*r* +++ reversepath *c*) +++ *c*)  
**by** (rule homotopic-paths-assoc[OF *r-path* *r-img* *revc-path* *revc-img* *c-path*])

```

c-img] (use rc in simp-all)
  then show ?thesis
    by (rule homotopic-paths-sym)
qed
have assoc1-join:
  homotopic-paths S (((r +++ reversepath c) +++ c) +++ s) (((r +++
(reversepath c +++ c)) +++ s))
  proof (rule homotopic-paths-join-right[OF assoc1 s-path s-img])
    show pathfinish ((r +++ reversepath c) +++ c) = pathstart s
      using rc cs by (simp add: pathstart-def pathfinish-def joinpaths-def re-
versepath-def)
  qed
  have assoc2:
    homotopic-paths S ((r +++ (reversepath c +++ c)) +++ s) (r +++
((reversepath c +++ c) +++ s))
  proof -
    have homotopic-paths S (r +++ ((reversepath c +++ c) +++ s)) (((r +++
(reversepath c +++ c)) +++ s))
      by (rule homotopic-paths-assoc[OF r-path r-img mid-path mid-img s-path
s-img]) (use rc cs in simp-all)
    then show ?thesis
      by (rule homotopic-paths-sym)
  qed
  have homotopic-paths S (((r +++ reversepath c) +++ c) +++ s) (r +++
((reversepath c +++ c) +++ s))
    by (rule homotopic-paths-trans[OF assoc1-join assoc2])
  then show ?thesis
    by (rule homotopic-paths-trans[OF - hom-left])
qed

```

**lemma** *segment-loop-base-full-in-set*:

```

assumes p-loop:  $p \in \text{loop-space } S \ x0$ 
shows homotopic-paths S (segment-loop p 0 1) p
proof -
  have p-path: path p
    and p-img: path-image p  $\subseteq S$ 
    and p0:  $p \ 0 = x0$ 
    and p1:  $p \ 1 = x0$ 
    using p-loop unfolding loop-space-def pathstart-def pathfinish-def by auto
  have x0-in-S:  $x0 \in S$ 
    using p-img p0 by (auto simp: path-image-def)
  have const-path: path ( $\lambda-. x0$ )
    by (simp add: path-def)
  have const-img: path-image ( $\lambda-. x0$ )  $\subseteq S$ 
    using x0-in-S by (auto simp: path-image-def)
  have hom-lid: homotopic-paths S (( $\lambda-. x0$ ) +++ p) p
    using homotopic-paths-lid-const[OF p-path p-img] p0 by (simp add: path-
start-def)
  have hom-join:

```

```

    homotopic-paths S (((λ-. x0) +++ p) +++ (λ-. x0)) (p +++ (λ-. x0))
proof (rule homotopic-paths-join-right[OF hom-lid const-path const-img])
    show pathfinish ((λ-. x0) +++ p) = pathstart (λ-. x0)
    using p0 p1 by (simp add: pathstart-def pathfinish-def joinpaths-def)
qed
have hom-rid: homotopic-paths S (p +++ (λ-. x0)) p
    using homotopic-paths-rid-const[OF p-path p-img] p1 by (simp add: pathfin-
ish-def)
have conn0: connector (p 0) = (λ-. x0)
    using p0 by (simp add: connector-def fun-eq-iff)
have conn1: reversepath (connector (p 1)) = (λ-. x0)
    using p1 by (simp add: connector-def reversepath-def fun-eq-iff)
have seg-eq: segment-loop p 0 1 = (((λ-. x0) +++ p) +++ (λ-. x0))
    unfolding segment-loop-def using conn0 conn1 by simp
have homotopic-paths S (segment-loop p 0 1) (p +++ (λ-. x0))
    unfolding seg-eq by (rule hom-join)
then show ?thesis
    by (rule homotopic-paths-trans[OF - hom-rid])
qed

```

```

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 -
  have p0: p 0 = x0 and p1: p 1 = x0
    using p-loop unfolding loop-space-def pathstart-def pathfinish-def by auto
  show ?thesis
proof
  fix t
  show segment-loop (p +++ q) 0 (1 / 2) t = segment-loop p 0 1 t
  proof (cases t ≤ 1 / 4)
    case True
    then show ?thesis
      unfolding segment-loop-def
      using p0 p1
      by (simp add: connector-def joinpaths-def subpathin-def reversepath-def
field-simps algebra-simps)
    next
    case False
    show ?thesis
    proof (cases t ≤ 1 / 2)
      case True
      have mid-gt: ¬ (2 * t ≤ 1 / 2)
        using False by linarith
      have sub-le: (4 * t - 1) / 2 ≤ 1 / 2
      proof -
      have 4 * t - 1 ≤ 1
        using True by linarith
    
```

```

then show ?thesis
  by (simp add: divide-right-mono)
qed
have start-eq: (p +++ q) 0 = x0
  using p0 by (simp add: joinpaths-def)
have mid-eq: (p +++ q) (1 / 2) = x0
  using p1 by (simp add: joinpaths-def)
have conn-start: connector ((p +++ q) 0) = (λ-. x0)
  using start-eq by (simp add: connector-def fun-eq-iff)
have conn-mid: reversepath (connector ((p +++ q) (1 / 2))) = (λ-. x0)
  using mid-eq by (simp add: connector-def reversepath-def fun-eq-iff)
have arg-eq: ((8 * t - 2) / 2 :: real) = 4 * t - 1
proof -
  have ((8 * t - 2) / 2 :: real) = (8 * t) / 2 - (2 :: real) / 2
    by (simp add: diff-divide-distrib)
  also have ... = 4 * t - 1
    by simp
  finally show ?thesis .
qed
have lhs-eq: segment-loop (p +++ q) 0 (1 / 2) t = p (4 * t - 1)
proof -
  have segment-loop (p +++ q) 0 (1 / 2) t =
    (((λ-. x0) +++ subpathin 0 (1 / 2) (p +++ q)) +++ (λ-. x0)) t
    unfolding segment-loop-def
    using conn-start conn-mid by simp
  also have ... = subpathin 0 (1 / 2) (p +++ q) (4 * t - 1)
    using False True mid-gt
    by (simp add: joinpaths-def field-simps algebra-simps)
  also have ... = (p +++ q) ((4 * t - 1) / 2)
    by (simp add: subpathin-def)
  also have ... = p ((8 * t - 2) / 2)
    using sub-le by (simp add: joinpaths-def field-simps algebra-simps)
  also have ... = p (4 * t - 1)
    by (subst arg-eq) simp
  finally show ?thesis .
qed
have rhs-eq: segment-loop p 0 1 t = p (4 * t - 1)
  unfolding segment-loop-def
  using p0 p1 False True mid-gt
  by (simp add: connector-def joinpaths-def subpathin-def reversepath-def
field-simps algebra-simps)
show ?thesis
  using lhs-eq rhs-eq by simp
next
case False
then show ?thesis
  unfolding segment-loop-def
  using p0 p1
  by (simp add: connector-def joinpaths-def subpathin-def reversepath-def

```

```

field-simps algebra-simps)
  qed
  qed
  qed
  qed

lemma segment-loop-joinpaths-tail-scaled [simp]:
  assumes p-loop:  $p \in \text{loop-space } W \ x0$ 
    and q-loop:  $q \in \text{loop-space } W \ x0$ 
    and u01:  $u \in \{0..1\}$ 
    and v01:  $v \in \{0..1\}$ 
  shows  $\text{segment-loop } (p \ +++ \ q) \ ((1 + u) / 2) \ ((1 + v) / 2) = \text{segment-loop } q$ 
  u v
proof -
  have p1:  $p \ 1 = x0$  and q0:  $q \ 0 = x0$ 
    using p-loop q-loop unfolding loop-space-def pathfinish-def pathstart-def by
  auto
  have pq:  $\text{pathstart } q = \text{pathfinish } p$ 
    using p-loop q-loop unfolding loop-space-def by auto
  have start-eq:  $(p \ +++ \ q) \ ((1 + u) / 2) = q \ u$ 
  proof (cases  $u = 0$ )
    case True
    then show ?thesis
      using p1 q0 by (simp add: joinpaths-def)
  next
    case False
    then have  $(1 / 2 :: \text{real}) < (1 + u) / 2$ 
      using u01 by auto
    then show ?thesis
      by (simp add: joinpaths-def field-simps algebra-simps)
  qed
  have finish-eq:  $(p \ +++ \ q) \ ((1 + v) / 2) = q \ v$ 
  proof (cases  $v = 0$ )
    case True
    then show ?thesis
      using p1 q0 by (simp add: joinpaths-def)
  next
    case False
    then have  $(1 / 2 :: \text{real}) < (1 + v) / 2$ 
      using v01 by auto
    then show ?thesis
      by (simp add: joinpaths-def field-simps algebra-simps)
  qed
  show ?thesis
  proof
    fix  $t :: \text{real}$ 
    show  $\text{segment-loop } (p \ +++ \ q) \ ((1 + u) / 2) \ ((1 + v) / 2) \ t = \text{segment-loop}$ 
    q u v t
    proof (cases  $t \leq 1 / 4$ )

```

```

case True
then show ?thesis
  unfolding segment-loop-def
  using start-eq finish-eq
  by (simp add: joinpaths-def field-simps algebra-simps)
next
case False
show ?thesis
proof (cases t ≤ 1 / 2)
  case True
  have s01: 4 * t - 1 ∈ {0..1}
    using False True by auto
  have sub-eq:
    subpathin ((1 + u) / 2) ((1 + v) / 2) (p +++ q) (4 * t - 1) =
    subpathin u v q (4 * t - 1)
    by (rule subpathin-joinpaths-tail-scaled-pointwise[OF pq u01 v01 s01])
  show ?thesis
    unfolding segment-loop-def
    using False True start-eq finish-eq sub-eq
    by (simp add: joinpaths-def field-simps algebra-simps)
  next
  case False
  then show ?thesis
    unfolding segment-loop-def
    using finish-eq
    by (simp add: joinpaths-def field-simps algebra-simps)
qed
qed
qed
qed

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:  $p \in \text{loop-space } W \ x0$ 
  and q-loop:  $q \in \text{loop-space } W \ x0$ 
  and t01:  $t \in \{0..1\}$ 
  shows  $(p \text{ +++ } q) ((1 + t) / 2) = q \ t$ 
proof (cases  $t = 0$ )
  case True
  then show ?thesis
    using p-loop q-loop
    unfolding loop-space-def pathstart-def pathfinish-def joinpaths-def
    by simp
next
  case False
  have t-pos:  $0 < t$ 
    using False t01 by auto
  then have param-gt:  $(1 + t) / 2 > 1 / 2$ 
    by (simp add: field-simps)
  have param-not-le:  $\neg ((1 + t) / 2 \leq 1 / 2)$ 
    using param-gt by simp
  have arg-eq:  $2 * ((1 + t) / 2) - 1 = (t :: \text{real})$ 
    by (simp add: field-simps algebra-simps)
  have  $(p \text{ +++ } q) ((1 + t) / 2) = q (2 * ((1 + t) / 2) - 1)$ 
    using param-not-le by (simp add: joinpaths-def algebra-simps)
  also have ... =  $q \ t$ 
    by (subst arg-eq) simp
  finally show ?thesis .
qed

```

```

lemma word-loop-in-W:
  assumes w-in: fpw-in-space G1 G2 w
  shows word-loop w ∈ loop-space W x0
  using w-in
proof (induction w)
  case WordNil

```

```

then show ?case
  by (simp add: constant-loop-in-space[OF x0-in-W])
next
case (WordLeft a rest)
then have a-in:  $a \in G1$  and rest-in:  $\text{fpw-in-space } G1 \ G2 \ rest$ 
  by auto
have a-loopU:  $\text{some-loop } U \ x0 \ a \in \text{loop-space } U \ x0$ 
  using some-loop-spec[OF a-in] by auto
have a-loopW:  $\text{some-loop } U \ x0 \ a \in \text{loop-space } W \ x0$ 
  using a-loopU unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
  case True
  then show ?thesis
    using a-loopW by simp
  next
  case False
  have tail-loop:  $\text{word-loop } rest \in \text{loop-space } W \ x0$ 
    by (rule WordLeft.IH[OF rest-in])
  have a-path:  $\text{path } (\text{some-loop } U \ x0 \ a)$ 
    and a-img:  $\text{path-image } (\text{some-loop } U \ x0 \ a) \subseteq W$ 
    and a-start:  $\text{pathstart } (\text{some-loop } U \ x0 \ a) = x0$ 
    and a-finish:  $\text{pathfinish } (\text{some-loop } U \ x0 \ a) = x0$ 
    using a-loopW unfolding loop-space-def by auto
  have tail-path:  $\text{path } (\text{word-loop } rest)$ 
    and tail-img:  $\text{path-image } (\text{word-loop } rest) \subseteq W$ 
    and tail-start:  $\text{pathstart } (\text{word-loop } rest) = x0$ 
    and tail-finish:  $\text{pathfinish } (\text{word-loop } rest) = x0$ 
    using tail-loop unfolding loop-space-def by auto
  have join-loop:  $\text{some-loop } U \ x0 \ a \ +++ \ \text{word-loop } rest \in \text{loop-space } W \ x0$ 
  proof -
    have join-path:  $\text{path } (\text{some-loop } U \ x0 \ a \ +++ \ \text{word-loop } rest)$ 
      using a-path tail-path a-finish tail-start by simp
    have join-img:  $\text{path-image } (\text{some-loop } U \ x0 \ a \ +++ \ \text{word-loop } rest) \subseteq W$ 
      by (rule subset-path-image-join[OF a-img tail-img])
    show ?thesis
      unfolding loop-space-def
      using join-path join-img a-start tail-finish by simp
  qed
  then show ?thesis
    using False by simp
qed
next
case (WordRight b rest)
then have b-in:  $b \in G2$  and rest-in:  $\text{fpw-in-space } G1 \ G2 \ rest$ 
  by auto
have b-loopV:  $\text{some-loop } V \ x0 \ b \in \text{loop-space } V \ x0$ 
  using some-loop-spec[OF b-in] by auto
have b-loopW:  $\text{some-loop } V \ x0 \ b \in \text{loop-space } W \ x0$ 

```

```

    using b-loopV unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
  case True
  then show ?thesis
    using b-loopW by simp
next
  case False
  have tail-loop: word-loop rest ∈ loop-space W x0
    by (rule WordRight.IH[OF rest-in])
  have b-path: path (some-loop V x0 b)
    and b-img: path-image (some-loop V x0 b) ⊆ W
    and b-start: pathstart (some-loop V x0 b) = x0
    and b-finish: pathfinish (some-loop V x0 b) = x0
    using b-loopW unfolding loop-space-def by auto
  have tail-path: path (word-loop rest)
    and tail-img: path-image (word-loop rest) ⊆ W
    and tail-start: pathstart (word-loop rest) = x0
    and tail-finish: pathfinish (word-loop rest) = x0
    using tail-loop unfolding loop-space-def by auto
  have join-loop: some-loop V x0 b +++ word-loop rest ∈ loop-space W x0
proof -
  have join-path: path (some-loop V x0 b +++ word-loop rest)
    using b-path tail-path b-finish tail-start by simp
  have join-img: path-image (some-loop V x0 b +++ word-loop rest) ⊆ W
    by (rule subset-path-image-join[OF b-img tail-img])
  show ?thesis
    unfolding loop-space-def
    using join-path join-img b-start tail-finish by simp
qed
then show ?thesis
  using False by simp
qed
qed

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
  using q-part
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case by simp
next
  case (Cons t ts)
  show ?case
  proof (cases ts)
    case Nil

```

```

have bs-nil:  $bs = []$ 
  using Cons.prems Nil by (cases bs) simp-all
have t1:  $t = 1$ 
  using Cons.prems Nil bs-nil by simp
have qt1UV:  $q \ 1 \in U \cap V$ 
  using Cons.prems Nil bs-nil t1 by simp
have qtUV:  $q \ t \in U \cap V$ 
  using t1 qt1UV by simp
have t01:  $t \in \{0..1\}$ 
  using t1 by simp
have joined-tUV:  $(p \ +++ \ q) \ ((1 + t) / 2) \in U \cap V$ 
  using qtUV joinpaths-tail-scaled-point[OF p-loop q-loop t01] by simp
show ?thesis
  using Nil bs-nil t1 joined-tUV by simp
next
case (Cons u us)
then obtain b bs' where bs:  $bs = b \# \ bs'$ 
  using Cons.prems by (cases bs) auto
have t01:  $t \in \{0..1\}$  and qtUV:  $q \ t \in U \cap V$ 
  and u01:  $u \in \{0..1\}$  and tu:  $t < u$ 
  and seg-side: (if b then subpathin  $t \ u \ q \ \{0..1\} \subseteq U$  else subpathin  $t \ u \ q \ \{0..1\} \subseteq V$ )
  and q-tail: svk-partition  $q \ (u \ \# \ us) \ bs'$ 
  using Cons.prems bs Cons by simp-all
have pq: pathstart  $q = \text{pathfinish } p$ 
  using p-loop q-loop unfolding loop-space-def by auto
have scaled-seg-side:
  (if b
    then subpathin  $((1 + t) / 2) \ ((1 + u) / 2) \ (p \ +++ \ q) \ \{0..1\} \subseteq U$ 
    else subpathin  $((1 + t) / 2) \ ((1 + u) / 2) \ (p \ +++ \ q) \ \{0..1\} \subseteq V$ )
  using seg-side by (simp add: subpathin-joinpaths-tail-scaled[OF pq t01 u01])
have q-tail-ts: svk-partition  $q \ ts \ bs'$ 
  using q-tail Cons by simp
have tail-scaled-ts:
  svk-partition  $(p \ +++ \ q) \ (\text{map } (\lambda t. (1 + t) / 2) \ ts) \ bs'$ 
  by (rule Cons.IH[OF q-tail-ts])
have tail-scaled:
  svk-partition  $(p \ +++ \ q) \ (\text{map } (\lambda t. (1 + t) / 2) \ (u \ \# \ us)) \ bs'$ 
  using tail-scaled-ts Cons by simp
have joined-tUV:  $(p \ +++ \ q) \ ((1 + t) / 2) \in U \cap V$ 
  using qtUV joinpaths-tail-scaled-point[OF p-loop q-loop t01] by simp
show ?thesis
  using bs Cons t01 joined-tUV u01 tu scaled-seg-side tail-scaled
  by simp
qed
qed

```

**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)
using w-in
proof (induction w)
case WordNil
have const-subU:  $(\lambda-. x0) \cdot \{0..1\} \subseteq U$ 
using x0-in-UV by auto
then show ?case
unfolding valid-partition-def
using x0-in-UV const-subU
by (auto simp: path-image-def subpathin-def)
next
case (WordLeft a rest)
then have a-in:  $a \in G1$  and rest-in: fpw-in-space G1 G2 rest
by auto
have a-loopU: some-loop U x0 a  $\in$  loop-space U x0
using some-loop-spec[OF a-in] by auto
have a-loopW: some-loop U x0 a  $\in$  loop-space W x0
using a-loopU unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
case True
then show ?thesis
using a-loopU x0-in-UV
unfolding valid-partition-def loop-space-def
by (auto simp: pathstart-def pathfinish-def path-image-def)
next
case False
have rest-valid:
valid-partition (word-loop rest) (word-partition-times rest) (word-partition-bits rest)
by (rule WordLeft.IH[OF rest-in])
have rest-svk:
svk-partition (word-loop rest) (word-partition-times rest) (word-partition-bits rest)
using rest-valid unfolding valid-partition-def by auto
have tail-scaled:
svk-partition (some-loop U x0 a +++ word-loop rest)
(map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest))
(word-partition-bits rest)
by (rule svk-partition-joinpaths-tail-scaled[OF a-loopW word-loop-in-W[OF rest-in] rest-svk])
have headU:
subpathin 0 (1 / 2) (some-loop U x0 a +++ word-loop rest)  $\cdot \{0..1\} \subseteq U$ 
using a-loopU by (simp add: loop-space-def path-image-def)
have startUV: (some-loop U x0 a +++ word-loop rest)  $0 \in U \cap V$ 
using a-loopU x0-in-UV unfolding loop-space-def pathstart-def joinpaths-def
by simp
have midUV: (some-loop U x0 a +++ word-loop rest)  $(1 / 2) \in U \cap V$ 

```

```

    using a-loopU x0-in-UV unfolding loop-space-def pathfinish-def joinpaths-def
  by simp
  have rest-times-nonempty: word-partition-times rest ≠ []
    using rest-valid unfolding valid-partition-def by auto
  have rest-times-hd: hd (word-partition-times rest) = 0
    using rest-valid unfolding valid-partition-def by auto
  have rest-times: word-partition-times rest = 0 # tl (word-partition-times rest)
  proof (cases word-partition-times rest)
    case Nil
    with rest-times-nonempty show ?thesis
      by simp
  next
    case (Cons s ss)
    have s = 0
      using rest-times-hd Cons by simp
    then show ?thesis
      using Cons by simp
  qed
  have scaled-times:
    map (λt. (1 + t) / 2) (word-partition-times rest) =
      1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest))
  proof -
    have map (λt. (1 + t) / 2) (word-partition-times rest) =
      map (λt. (1 + t) / 2) (0 # tl (word-partition-times rest))
    using rest-times by simp
    also have ... = 1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times
rest))
      by simp
    finally show ?thesis .
  qed
  have tail-scaled':
    svk-partition (some-loop U x0 a +++ word-loop rest)
      (1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
      (word-partition-bits rest)
    using tail-scaled scaled-times by simp
  have step-svk:
    svk-partition (some-loop U x0 a +++ word-loop rest)
      (0 # 1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
      (True # word-partition-bits rest)
    using False tail-scaled' headU startUV midUV
    by simp
  have step-svk':
    svk-partition (some-loop U x0 a +++ word-loop rest)
      (0 # map (λt. (1 + t) / 2) (word-partition-times rest))
      (True # word-partition-bits rest)
    using step-svk scaled-times by simp
  show ?thesis
    unfolding valid-partition-def
    using False step-svk'

```

```

    by simp
  qed
next
case (WordRight b rest)
then have b-in:  $b \in G2$  and rest-in:  $fpw\text{-in-space } G1\ G2\ rest$ 
  by auto
have b-loopV:  $some\text{-loop } V\ x0\ b \in loop\text{-space } V\ x0$ 
  using some-loop-spec[OF b-in] by auto
have b-loopW:  $some\text{-loop } V\ x0\ b \in loop\text{-space } W\ x0$ 
  using b-loopV unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
  case True
  then show ?thesis
    using b-loopV x0-in-UV
    unfolding valid-partition-def loop-space-def
    by (auto simp: pathstart-def pathfinish-def path-image-def)
  next
  case False
  have rest-valid:
     $valid\text{-partition } (word\text{-loop } rest)\ (word\text{-partition-times } rest)\ (word\text{-partition-bits } rest)$ 
  by (rule WordRight.IH[OF rest-in])
  have rest-svk:
     $svk\text{-partition } (word\text{-loop } rest)\ (word\text{-partition-times } rest)\ (word\text{-partition-bits } rest)$ 
  using rest-valid unfolding valid-partition-def by auto
  have tail-scaled:
     $svk\text{-partition } (some\text{-loop } V\ x0\ b\ \text{+++ } word\text{-loop } rest)$ 
     $(map\ (\lambda t. (1 + t) / 2)\ (word\text{-partition-times } rest))$ 
     $(word\text{-partition-bits } rest)$ 
  by (rule svk-partition-joinpaths-tail-scaled[OF b-loopW word-loop-in-W[OF rest-in] rest-svk])
  have headV:
     $subpathin\ 0\ (1 / 2)\ (some\text{-loop } V\ x0\ b\ \text{+++ } word\text{-loop } rest)\ \{0..1\} \subseteq V$ 
  using b-loopV by (simp add: loop-space-def path-image-def)
  have startUV:  $(some\text{-loop } V\ x0\ b\ \text{+++ } word\text{-loop } rest)\ 0 \in U \cap V$ 
  using b-loopV x0-in-UV unfolding loop-space-def pathstart-def joinpaths-def
  by simp
  have midUV:  $(some\text{-loop } V\ x0\ b\ \text{+++ } word\text{-loop } rest)\ (1 / 2) \in U \cap V$ 
  using b-loopV x0-in-UV unfolding loop-space-def pathfinish-def joinpaths-def
  by simp
  have rest-times-nonempty:  $word\text{-partition-times } rest \neq []$ 
  using rest-valid unfolding valid-partition-def by auto
  have rest-times-hd:  $hd\ (word\text{-partition-times } rest) = 0$ 
  using rest-valid unfolding valid-partition-def by auto
  have rest-times:  $word\text{-partition-times } rest = 0 \# tl\ (word\text{-partition-times } rest)$ 
  proof (cases word-partition-times rest)
    case Nil

```

```

with rest-times-nonempty show ?thesis
  by simp
next
  case (Cons s ss)
  have s = 0
    using rest-times-hd Cons by simp
  then show ?thesis
    using Cons by simp
qed
have scaled-times:
  map (λt. (1 + t) / 2) (word-partition-times rest) =
    1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest))
proof -
  have map (λt. (1 + t) / 2) (word-partition-times rest) =
    map (λt. (1 + t) / 2) (0 # tl (word-partition-times rest))
    using rest-times by simp
  also have ... = 1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times
rest))
    by simp
  finally show ?thesis .
qed
have tail-scaled':
  svk-partition (some-loop V x0 b +++ word-loop rest)
    (1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
    (word-partition-bits rest)
  using tail-scaled scaled-times by simp
have step-svk:
  svk-partition (some-loop V x0 b +++ word-loop rest)
    (0 # 1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
    (False # word-partition-bits rest)
  using False tail-scaled' headV startUV midUV
  by simp
have step-svk':
  svk-partition (some-loop V x0 b +++ word-loop rest)
    (0 # map (λt. (1 + t) / 2) (word-partition-times rest))
    (False # word-partition-bits rest)
  using step-svk scaled-times by simp
show ?thesis
  unfolding valid-partition-def
  using False step-svk'
  by simp
qed
qed

lemma partition-word-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 partition-word (p +++ q) (map (λt. (1 + t) / 2) ts) bs = partition-word

```

```

q ts bs
  using q-part
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by simp
next
  case (Cons t ts)
  show ?case
  proof (cases ts)
    case Nil
    then show ?thesis
      using Cons.prem1 by (cases bs) simp-all
  next
  case (Cons u us)
  then obtain b bs' where bs: bs = b # bs'
    using Cons.prem1 by (cases bs) auto
  have tail: svk-partition q (u # us) bs'
    using Cons.prem1 Cons bs by simp
  have t01: t ∈ {0..1} and u01: u ∈ {0..1}
    using Cons.prem1 Cons bs by simp-all
  have tail-ts: svk-partition q ts bs'
    using tail Cons by simp
  have ih0:
    partition-word (p +++ q) (map (λt. (1 + t) / 2) ts) bs' =
      partition-word q ts bs'
    by (rule Cons.IH[of bs', OF tail-ts])
  have ih:
    partition-word (p +++ q) (map (λt. (1 + t) / 2) (u # us)) bs' =
      partition-word q (u # us) bs'
    using ih0 Cons by simp
  have seg-eq:
    segment-loop (p +++ q) ((1 + t) / 2) ((1 + u) / 2) = segment-loop q t u
    by (rule segment-loop-joinpaths-tail-scaled[OF p-loop q-loop t01 u01])
  show ?thesis
    using Cons bs ih seg-eq
    by simp
qed
qed

```

**lemma** *segment-loop-some-loop-left-class*:

**assumes** *a-in*:  $a \in G1$

**shows**  $\text{loop-class } U \ x0 \ (\text{segment-loop } (\text{some-loop } U \ x0 \ a) \ 0 \ 1) = a$

**proof** –

**have** *p-loop*:  $\text{some-loop } U \ x0 \ a \in \text{loop-space } U \ x0$

**and** *a-eq*:  $a = \text{loop-class } U \ x0 \ (\text{some-loop } U \ x0 \ a)$

**using** *some-loop-spec*[OF *a-in*] **by** *auto*

**have** *p-path*:  $\text{path } (\text{some-loop } U \ x0 \ a)$

**and** *p-imgU*:  $\text{path-image } (\text{some-loop } U \ x0 \ a) \subseteq U$

```

and p0: some-loop U x0 a 0 = x0
and p1: some-loop U x0 a 1 = x0
using p-loop unfolding loop-space-def pathstart-def pathfinish-def by auto
have segU: segment-loop (some-loop U x0 a) 0 1 ∈ loop-space U x0
proof (rule segment-loop-in-U[OF p-path])
  show path-image (some-loop U x0 a) ⊆ W
    using p-imgU by auto
  show (0::real) ∈ {0..1}
    by simp
  show (1::real) ∈ {0..1}
    by simp
  show some-loop U x0 a 0 ∈ U ∩ V some-loop U x0 a 1 ∈ U ∩ V
    using p0 p1 x0-in-UV by simp-all
  show subpathin 0 1 (some-loop U x0 a) ‘ {0..1} ⊆ U
    using p-imgU by (simp add: path-image-def)
qed
have seg-eq:
  loop-class U x0 (segment-loop (some-loop U x0 a) 0 1) =
  loop-class U x0 (some-loop U x0 a)
  by (rule loop-class-eqI[OF segU p-loop segment-loop-base-full-in-set[OF p-loop]])
show ?thesis
  using seg-eq a-eq by simp
qed

lemma segment-loop-some-loop-right-class:
  assumes b-in: b ∈ G2
  shows loop-class V x0 (segment-loop (some-loop V x0 b) 0 1) = b
proof –
  have p-loop: some-loop V x0 b ∈ loop-space V x0
    and b-eq: b = loop-class V x0 (some-loop V x0 b)
    using some-loop-spec[OF b-in] by auto
  have p-path: path (some-loop V x0 b)
    and p-imgV: path-image (some-loop V x0 b) ⊆ V
    and p0: some-loop V x0 b 0 = x0
    and p1: some-loop V x0 b 1 = x0
    using p-loop unfolding loop-space-def pathstart-def pathfinish-def by auto
  have segV: segment-loop (some-loop V x0 b) 0 1 ∈ loop-space V x0
proof (rule segment-loop-in-V[OF p-path])
  show path-image (some-loop V x0 b) ⊆ W
    using p-imgV by auto
  show (0::real) ∈ {0..1}
    by simp
  show (1::real) ∈ {0..1}
    by simp
  show some-loop V x0 b 0 ∈ U ∩ V some-loop V x0 b 1 ∈ U ∩ V
    using p0 p1 x0-in-UV by simp-all
  show subpathin 0 1 (some-loop V x0 b) ‘ {0..1} ⊆ V
    using p-imgV by (simp add: path-image-def)
qed

```

```

have seg-eq:
  loop-class  $V\ x0$  (segment-loop (some-loop  $V\ x0\ b$ )  $0\ 1$ ) =
    loop-class  $V\ x0$  (some-loop  $V\ x0\ b$ )
  by (rule loop-class-eqI[OF segV p-loop segment-loop-base-full-in-set[OF p-loop]])
show ?thesis
  using seg-eq b-eq by simp
qed

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
  using w-in
proof (induction  $w$ )
  case WordNil
  have nil-loopU: word-loop WordNil  $\in$  loop-space  $U\ x0$ 
    by (simp add: constant-loop-in-space[OF x0-in-U])
  have nil-segU: segment-loop (word-loop WordNil)  $0\ 1 \in$  loop-space  $U\ x0$ 
proof (rule segment-loop-in-U)
  show path (word-loop WordNil)
    using nil-loopU unfolding loop-space-def by simp
  show path-image (word-loop WordNil)  $\subseteq$   $W$ 
    using nil-loopU unfolding loop-space-def by simp
  show ( $0::real$ )  $\in$   $\{0..1\}$ 
    by simp
  show ( $1::real$ )  $\in$   $\{0..1\}$ 
    by simp
  show word-loop WordNil  $0 \in U \cap V$  word-loop WordNil  $1 \in U \cap V$ 
    using x0-in-UV by simp-all
  show subpathin  $0\ 1$  (word-loop WordNil) ‘  $\{0..1\} \subseteq U$ 
proof
  fix  $y$ 
  assume  $y \in$  subpathin  $0\ 1$  (word-loop WordNil) ‘  $\{0..1\}$ 
  then obtain  $t$  where  $t \in \{0..1\}$  and  $y$ -eq:  $y =$  subpathin  $0\ 1$  (word-loop
WordNil)  $t$ 
    by blast
  have  $y = x0$ 
    using  $y$ -eq by (simp add: word-loop.simps subpathin-def)
  then show  $y \in U$ 
    using x0-in-U by simp
  qed
qed
have nil-hom:
  homotopic-paths  $U$  (segment-loop (word-loop WordNil)  $0\ 1$ ) (word-loop WordNil)
  by (rule segment-loop-base-full-in-set[OF nil-loopU])
have nil-class:
  loop-class  $U\ x0$  (segment-loop (word-loop WordNil)  $0\ 1$ ) = one1
proof –

```

```

have loop-class  $U\ x0$  (segment-loop (word-loop WordNil) 0 1) =
  loop-class  $U\ x0$  (word-loop WordNil)
  by (rule loop-class-eqI[OF nil-segU nil-loopU nil-hom])
then show ?thesis
  unfolding word-loop.simps fundamental-group-one-def by simp
qed
have one1-in: one1  $\in G1$ 
  by (rule fundamental-group-one-in-space[OF x0-in-U])
have red:
  carrier-fpw-reduction  $G1\ G2\ mult1\ one1\ mult2\ one2$ 
  (WordLeft one1 WordNil) WordNil
  by (rule carrier-fpw-reduction.step,
    rule carrier-fpw-reduction-step.remove-left-one[OF one1-in], simp)
show ?case
  using nil-class red
  by (simp add: carrier-full-amalg-equiv.of-reduction)
next
case (WordLeft a rest)
then have a-in:  $a \in G1$  and rest-in: fpw-in-space  $G1\ G2\ rest$ 
  by auto
have a-loopU: some-loop  $U\ x0\ a \in loop-space\ U\ x0$ 
  using some-loop-spec[OF a-in] by auto
have a-loopW: some-loop  $U\ x0\ a \in loop-space\ W\ x0$ 
  using a-loopU unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
  case True
  then show ?thesis
    using segment-loop-some-loop-left-class[OF a-in]
    by simp
next
case False
have rest-loopW: word-loop rest  $\in loop-space\ W\ x0$ 
  by (rule word-loop-in-W[OF rest-in])
have rest-valid:
  valid-partition (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
  by (rule word-loop-valid-partition[OF rest-in])
have rest-svk:
  svk-partition (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
  using rest-valid unfolding valid-partition-def by auto
have rest-times-nonempty: word-partition-times rest  $\neq []$ 
  using rest-valid unfolding valid-partition-def by auto
have rest-times-hd: hd (word-partition-times rest) = 0
  using rest-valid unfolding valid-partition-def by auto
have rest-times: word-partition-times rest = 0 # tl (word-partition-times rest)
proof (cases word-partition-times rest)
  case Nil

```

```

with rest-times-nonempty show ?thesis
  by simp
next
  case (Cons s ss)
  have s = 0
    using rest-times-hd Cons by simp
  then show ?thesis
    using Cons by simp
qed
have scaled-times:
  map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest) =
   $1 / 2 \# \text{map } (\lambda t. (1 + t) / 2) (\text{tl } (\text{word-partition-times rest}))$ 
proof -
  have map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest) =
    map ( $\lambda t. (1 + t) / 2$ ) ( $0 \# \text{tl } (\text{word-partition-times rest})$ )
    using rest-times by simp
  also have  $\dots = 1 / 2 \# \text{map } (\lambda t. (1 + t) / 2) (\text{tl } (\text{word-partition-times}$ 
rest))
    by simp
  finally show ?thesis .
qed
have tail-eq:
  partition-word (some-loop U x0 a +++ word-loop rest)
    (map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest))
    (word-partition-bits rest) =
  partition-word (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
    by (rule partition-word-joinpaths-tail-scaled[OF a-loopW rest-loopW rest-svk])
have head-eq:
  loop-class U x0
    (segment-loop (some-loop U x0 a +++ word-loop rest)  $0 (1 / 2)$ ) = a
    using segment-loop-some-loop-left-class[OF a-in]
  by (simp add: segment-loop-joinpaths-head[OF a-loopU rest-loopW])
have tail-eq':
  partition-word (some-loop U x0 a +++ word-loop rest)
    ( $1 / 2 \# \text{map } (\lambda t. (1 + t) / 2) (\text{tl } (\text{word-partition-times rest}))$ )
    (word-partition-bits rest) =
  partition-word (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
    using tail-eq scaled-times by simp
have step-eq:
  partition-word (word-loop (WordLeft a rest))
    (word-partition-times (WordLeft a rest))
    (word-partition-bits (WordLeft a rest)) =
  WordLeft a
    (partition-word (word-loop rest) (word-partition-times rest)
    (word-partition-bits rest))
proof -
  have partition-word (word-loop (WordLeft a rest))

```

```

      (word-partition-times (WordLeft a rest))
      (word-partition-bits (WordLeft a rest)) =
WordLeft (loop-class U x0
  (segment-loop (some-loop U x0 a +++ word-loop rest) 0 (1 / 2)))
  (partition-word (some-loop U x0 a +++ word-loop rest)
    (1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
    (word-partition-bits rest))
using False scaled-times by simp
also have ... =
  WordLeft a
    (partition-word (word-loop rest) (word-partition-times rest)
      (word-partition-bits rest))
using head-eq tail-eq' by simp
finally show ?thesis .
qed
have tail-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (word-loop rest) (word-partition-times rest)
      (word-partition-bits rest))
  rest
by (rule WordLeft.IH[OF rest-in])
have
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordLeft a
      (partition-word (word-loop rest) (word-partition-times rest)
        (word-partition-bits rest)))
    (WordLeft a rest)
by (rule carrier-full-amalg-equiv-left-context[OF tail-rel a-in])
then show ?thesis
by (subst step-eq) simp
qed
next
case (WordRight b rest)
then have b-in:  $b \in G2$  and rest-in: fpw-in-space G1 G2 rest
by auto
have b-loopV: some-loop V x0  $b \in \text{loop-space } V x0$ 
using some-loop-spec[OF b-in] by auto
have b-loopW: some-loop V x0  $b \in \text{loop-space } W x0$ 
using b-loopV unfolding loop-space-def by auto
show ?case
proof (cases rest = WordNil)
case True
then show ?thesis
using segment-loop-some-loop-right-class[OF b-in]
by simp
next
case False
have rest-loopW: word-loop rest  $\in \text{loop-space } W x0$ 
by (rule word-loop-in-W[OF rest-in])

```

```

have rest-valid:
  valid-partition (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
  by (rule word-loop-valid-partition[OF rest-in])
have rest-svk:
  svk-partition (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
  using rest-valid unfolding valid-partition-def by auto
have rest-times-nonempty: word-partition-times rest  $\neq$  []
  using rest-valid unfolding valid-partition-def by auto
have rest-times-hd: hd (word-partition-times rest) = 0
  using rest-valid unfolding valid-partition-def by auto
have rest-times: word-partition-times rest = 0 # tl (word-partition-times rest)
proof (cases word-partition-times rest)
  case Nil
  with rest-times-nonempty show ?thesis
  by simp
next
  case (Cons s ss)
  have s = 0
  using rest-times-hd Cons by simp
  then show ?thesis
  using Cons by simp
qed
have scaled-times:
  map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest) =
   $1 / 2$  # map ( $\lambda t. (1 + t) / 2$ ) (tl (word-partition-times rest))
proof -
  have map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest) =
  map ( $\lambda t. (1 + t) / 2$ ) (0 # tl (word-partition-times rest))
  using rest-times by simp
  also have  $\dots = 1 / 2$  # map ( $\lambda t. (1 + t) / 2$ ) (tl (word-partition-times
rest))
  by simp
  finally show ?thesis .
qed
have tail-eq:
  partition-word (some-loop V x0 b +++ word-loop rest)
  (map ( $\lambda t. (1 + t) / 2$ ) (word-partition-times rest))
  (word-partition-bits rest) =
  partition-word (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
  by (rule partition-word-joinpaths-tail-scaled[OF b-loopW rest-loopW rest-svk])
have head-eq:
  loop-class V x0
  (segment-loop (some-loop V x0 b +++ word-loop rest) 0 (1 / 2)) = b
  using segment-loop-some-loop-right-class[OF b-in]
  by (simp add: segment-loop-joinpaths-head[OF b-loopV rest-loopW])
have tail-eq':

```

```

partition-word (some-loop V x0 b +++ word-loop rest)
  (1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
  (word-partition-bits rest) =
partition-word (word-loop rest) (word-partition-times rest) (word-partition-bits
rest)
using tail-eq scaled-times by simp
have step-eq:
partition-word (word-loop (WordRight b rest))
  (word-partition-times (WordRight b rest))
  (word-partition-bits (WordRight b rest)) =
  WordRight b
    (partition-word (word-loop rest) (word-partition-times rest)
(word-partition-bits rest))
proof -
have partition-word (word-loop (WordRight b rest))
  (word-partition-times (WordRight b rest))
  (word-partition-bits (WordRight b rest)) =
  WordRight (loop-class V x0
    (segment-loop (some-loop V x0 b +++ word-loop rest) 0 (1 / 2)))
    (partition-word (some-loop V x0 b +++ word-loop rest)
    (1 / 2 # map (λt. (1 + t) / 2) (tl (word-partition-times rest)))
    (word-partition-bits rest))
using False scaled-times by simp
also have ... =
  WordRight b
    (partition-word (word-loop rest) (word-partition-times rest)
(word-partition-bits rest))
using head-eq tail-eq' by simp
finally show ?thesis .
qed
have tail-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (word-loop rest) (word-partition-times rest)
(word-partition-bits rest))
  rest
by (rule WordRight.IH[OF rest-in])
have
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordRight b
    (partition-word (word-loop rest) (word-partition-times rest)
(word-partition-bits rest)))
    (WordRight b rest)
by (rule carrier-full-amalg-equiv-right-context[OF tail-rel b-in])
then show ?thesis
by (subst step-eq) simp
qed
qed

lemma valid-partition-hd:

```

```

assumes valid-partition  $p$   $ts$   $bs$ 
shows  $ts \neq []$   $hd\ ts = 0$ 
using assms unfolding valid-partition-def by auto

lemma valid-partition-cases:
assumes valid-partition  $p$  ( $t \# ts$ )  $bs$ 
shows  $t = 0$  and svk-partition  $p$  ( $t \# ts$ )  $bs$ 
using assms unfolding valid-partition-def by auto

lemma svk-partition-head-props:
assumes svk-partition  $p$  ( $t \# ts$ )  $bs$ 
shows  $t \in \{0..1\}$  and  $p\ t \in U \cap V$ 
proof –
  show  $t \in \{0..1\}$ 
  proof (cases  $ts$ )
    case Nil
      then show ?thesis
        using assms by (cases  $bs$ ) auto
    next
      case (Cons  $u$   $us$ )
        then show ?thesis
          using assms by (cases  $bs$ ) auto
  qed
next
  show  $p\ t \in U \cap V$ 
  proof (cases  $ts$ )
    case Nil
      then show ?thesis
        using assms by (cases  $bs$ ) auto
    next
      case (Cons  $u$   $us$ )
        then show ?thesis
          using assms by (cases  $bs$ ) auto
  qed
qed

lemma svk-partition-tail:
assumes svk-partition  $p$  ( $t \# u \# ts$ ) ( $b \# bs$ )
shows svk-partition  $p$  ( $u \# ts$ )  $bs$ 
using assms by simp

lemma svk-partition-step-props:
assumes svk-partition  $p$  ( $t \# u \# ts$ ) ( $b \# bs$ )
shows  $t \in \{0..1\}$ 
  and  $p\ t \in U \cap V$ 
  and  $u \in \{0..1\}$ 
  and  $t < u$ 
  and (if  $b$  then subpathin  $t\ u\ p\ '\{0..1\} \subseteq U$  else subpathin  $t\ u\ p\ '\{0..1\} \subseteq V$ )
using assms by simp-all

```

**lemma** *svk-partition-next-in-intersection*:  
**assumes** *svk-partition*  $p$  ( $t \# u \# ts$ ) ( $b \# bs$ )  
**shows**  $p \ u \in U \cap V$   
**proof** –  
**have** *svk-partition*  $p$  ( $u \# ts$ )  $bs$   
**using** *assms* **by** *simp*  
**then show** *?thesis*  
**by** (*rule svk-partition-head-props*(2))  
**qed**

**lemma** *svk-partition-nonempty*:  
**assumes** *svk-partition*  $p$   $ts$   $bs$   
**shows**  $ts \neq []$   
**using** *assms* **by** (*cases*  $ts$ ) *simp-all*

**lemma** *svk-partition-last-eq-one*:  
**assumes** *part*: *svk-partition*  $p$   $ts$   $bs$   
**shows**  $last\ ts = 1$   
**using** *part*  
**proof** (*induction*  $ts$  *arbitrary*:  $bs$ )  
**case** *Nil*  
**then show** *?case*  
**by** *simp*  
**next**  
**case** (*Cons*  $t$   $ts$ )  
**show** *?case*  
**proof** (*cases*  $ts$ )  
**case** *Nil*  
**then show** *?thesis*  
**using** *Cons.prem*s **by** (*cases*  $bs$ ) *auto*  
**next**  
**case** (*Cons*  $u$   $us$ )  
**then obtain**  $b$   $bs'$  **where**  $bs = b \# bs'$   
**using** *Cons.prem*s **by** (*cases*  $bs$ ) *auto*  
**have** *tail*: *svk-partition*  $p$  ( $u \# us$ )  $bs'$   
**using** *Cons.prem*s *Cons*  $bs$  **by** *simp*  
**from** *Cons.IH*[*of*  $bs'$ ] *Cons tail* **show** *?thesis*  
**by** *simp*  
**qed**  
**qed**

**lemma** *svk-partition-last-in-intersection*:  
**assumes** *part*: *svk-partition*  $p$   $ts$   $bs$   
**shows**  $p$  ( $last\ ts$ )  $\in U \cap V$   
**using** *part*  
**proof** (*induction*  $ts$  *arbitrary*:  $bs$ )  
**case** *Nil*  
**then show** *?case*

```

    by simp
next
case (Cons t ts)
show ?case
proof (cases ts)
  case Nil
  then show ?thesis
    using Cons.prem1 by (cases bs) auto
next
case (Cons u us)
then obtain b bs' where bs: bs = b # bs'
  using Cons.prem1 by (cases bs) auto
have tail: svk-partition p (u # us) bs'
  using Cons.prem1 Cons bs by simp
from Cons.IH[of bs] Cons tail show ?thesis
  by simp
qed
qed

```

```

lemma svk-partition-last-props:
  assumes part: svk-partition p ts bs
  shows ts ≠ [] and last ts = 1 and p (last ts) ∈ U ∩ V
  using svk-partition-nonempty[OF part]
    svk-partition-last-eq-one[OF part]
    svk-partition-last-in-intersection[OF part]
  by auto

```

```

lemma valid-partition-last-props:
  assumes valid-partition p ts bs
  shows ts ≠ [] and last ts = 1 and p (last ts) ∈ U ∩ V
proof -
  have ts-ne: ts ≠ []
    using assms unfolding valid-partition-def by auto
  have part: svk-partition p ts bs
    using assms unfolding valid-partition-def by auto
  show ts ≠ []
    by (rule ts-ne)
  from svk-partition-last-props[OF part] ts-ne
  show last ts = 1 and p (last ts) ∈ U ∩ V
    by auto
qed

```

```

lemma subpathin-endpoints-in-set:
  assumes seg-in: subpathin u v p ' {0..1} ⊆ S
  shows p u ∈ S and p v ∈ S
proof -
  have u-in: subpathin u v p 0 ∈ S
    using seg-in by auto
  then show p u ∈ S

```

by (*simp add: subpathin-def*)  
 have *v-in*:  $\text{subpathin } u \ v \ p \ 1 \in S$   
 using *seg-in* by *auto*  
 then show  $p \ v \in S$   
 by (*simp add: subpathin-def*)  
 qed

**lemma** *subpathin-image-subset-union*:

assumes *tu*:  $t \leq u$   
 and *uv*:  $u \leq v$   
 shows  $\text{subpathin } t \ v \ p \ ' \ {0..1} \subseteq \text{subpathin } t \ u \ p \ ' \ {0..1} \cup \text{subpathin } u \ v \ p \ ' \ {0..1}$   
 proof –  
 have *seg-subset*:  $\text{closed-segment } t \ v \subseteq \text{closed-segment } t \ u \cup \text{closed-segment } u \ v$   
 using *tu uv* by (*auto simp: closed-segment-eq-real-ivl*)  
 show ?thesis  
 using *seg-subset* by (*auto simp: subpathin-image*)  
 qed

**lemma** *subpathin-image-subset-trans*:

assumes *tu*:  $t \leq u$   
 and *uv*:  $u \leq v$   
 and *left*:  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq S$   
 and *right*:  $\text{subpathin } u \ v \ p \ ' \ {0..1} \subseteq S$   
 shows  $\text{subpathin } t \ v \ p \ ' \ {0..1} \subseteq S$   
 using *subpathin-image-subset-union*[*OF tu uv, of p*] *left right* by *blast*

**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*  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq U$  *else*  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq V$ )  
 and *cover-partition*  $p \ (u \ \# \ ts) \ bs$   
 using *assms* by *simp-all*

**lemma** *cover-partition-consI*:

assumes  $t \in \{0..1\}$   
 and  $u \in \{0..1\}$   
 and  $t < u$   
 and (*if b then*  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq U$  *else*  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq V$ )  
 and *cover-partition*  $p \ (u \ \# \ ts) \ bs$   
 shows *cover-partition*  $p \ (t \ \# \ u \ \# \ ts) \ (b \ \# \ bs)$   
 using *assms* by *simp*

**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 (cases b)
  case True
    then have leftU: subpathin t u p ‘ {0..1} ⊆ U
      using cp by simp
    from diff True have c-false: ¬ c
      by simp
    then have rightV: subpathin u v p ‘ {0..1} ⊆ V
      using cp by simp
    show ?thesis
      using subpathin-endpoints-in-set(2)[OF leftU] sub-
pathin-endpoints-in-set(1)[OF rightV]
      by blast
  next
    case False
    then have leftV: subpathin t u p ‘ {0..1} ⊆ V
      using cp by simp
    from diff False have c-true: c
      by simp
    then have rightU: subpathin u v p ‘ {0..1} ⊆ U
      using cp by simp
    show ?thesis
      using subpathin-endpoints-in-set(1)[OF rightU] sub-
pathin-endpoints-in-set(2)[OF leftV]
      by blast
qed

lemma cover-partition-pair-svk-partition:
  assumes cp: cover-partition p [t, u] [b]
    and ptUV: p t ∈ U ∩ V
    and puUV: p u ∈ U ∩ V
  shows svk-partition p [t, u] [b]
proof –
  have t01: t ∈ {0..1}
    using cp by simp
  have u01: u ∈ {0..1}
    using cp by simp
  have tu: t < u
    using cp by simp
  have u1: u = 1
    using cp by simp
  have seg: (if b then subpathin t u p ‘ {0..1} ⊆ U else subpathin t u p ‘ {0..1} ⊆
V)
proof (cases b)
  case True
    then show ?thesis
      using cp u1 by simp
  next
    case False
    then show ?thesis

```

```

    using cp u1 by simp
  qed
  have tail: svk-partition p [u] []
    using u1 puUV by simp
  show ?thesis
  proof (cases b)
    case True
    then show ?thesis
      using t01 ptUV u01 tu seg u1 puUV by simp
    next
    case False
    then show ?thesis
      using t01 ptUV u01 tu seg u1 puUV by simp
  qed
qed

lemma cover-partition-compress-svk-partition:
  assumes cp: cover-partition p (t # ts) bs
    and ptUV: p t ∈ U ∩ V
    and plastUV: p (last (t # ts)) ∈ U ∩ V
  shows ∃ ts' bs'. svk-partition p (t # ts') bs'
  using assms
proof (induction length bs arbitrary: bs t ts rule: less-induct)
  case less
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases ts)
      case Nil-ts: Nil
      then have t1: t = 1
        using less.prem(1) Nil by simp
      have svk-partition p [t] []
        using t1 less.prem(2) by simp
      show ?thesis
    proof
      show ∃ bs'. svk-partition p (t # []) bs'
    proof
      show svk-partition p (t # []) []
        using t1 less.prem(2) by simp
    qed
  qed
  next
  case (Cons u us)
  then have False
    using less.prem(1) Nil by simp
  then show ?thesis
    by simp
qed

```

```

next
  case (Cons b bs0)
  note bs-cons = Cons
  show ?thesis
  proof (cases bs0)
    case Nil
    then obtain u where ts: ts = [u]
    proof (cases ts)
      case Nil
      then have False
        using less.premis(1) bs-cons Nil by simp
      then show thesis
        by simp
    next
    case (Cons u ts0)
    note ts-cons = Cons
    then show thesis
    proof (cases ts0)
      case Nil
      have ts-single: ts = [u]
        using ts-cons Nil by simp
      from ts-single show thesis
        by (rule that[of u])
    next
    case (Cons v us)
    then have False
      using less.premis(1) bs-cons Nil ts-cons by simp
    then show thesis
      by simp
  qed
  qed
  have puUV: p u ∈ U ∩ V
    using less.premis(3) unfolding ts by simp
  have t01: t ∈ {0..1}
    using less.premis(1) bs-cons Nil ts by simp
  have u01: u ∈ {0..1}
    using less.premis(1) bs-cons Nil ts by simp
  have tu: t < u
    using less.premis(1) bs-cons Nil ts by simp
  have u1: u = 1
    using less.premis(1) bs-cons Nil ts by simp
  have seg-tu: (if b then subpathin t u p ‘ {0..1} ⊆ U else subpathin t u p ‘
{0..1} ⊆ V)
  proof (cases b)
    case True
    then show ?thesis
      using less.premis(1) bs-cons Nil ts u1 by simp
  next
  case False

```

```

    then show ?thesis
      using less.premis(1) bs-cons Nil ts u1 by simp
qed
have part-tu: svk-partition p [t, u] [b]
proof (cases b)
  case True
  then show ?thesis
    using t01 less.premis(2) u01 tu seg-tu u1 puUV by simp
next
  case False
  then show ?thesis
    using t01 less.premis(2) u01 tu seg-tu u1 puUV by simp
qed
show ?thesis
proof
  show  $\exists bs'. svk-partition p (t \# [u]) bs'$ 
  proof
    show svk-partition p (t \# [u]) [b]
      by (rule part-tu)
  qed
qed
next
case (Cons c bs^)
note bs0-cons = Cons
have bs-eq: bs = b \# c \# bs'
  using bs-cons bs0-cons by simp
then obtain u v us where ts: ts = u \# v \# us
proof (cases ts)
  case Nil
  then have False
    using less.premis(1) bs-cons bs0-cons by simp
  then show thesis
    by simp
next
case (Cons u ts0)
note ts-cons = Cons
then show thesis
proof (cases ts0)
  case Nil
  then have False
    using less.premis(1) bs-cons bs0-cons ts-cons by simp
  then show thesis
    by simp
next
case (Cons v us)
have ts-long: ts = u \# v \# us
  using ts-cons Cons by simp
from ts-long show thesis
  by (rule that[of u v us])

```

```

qed
qed
have cp-step: cover-partition p (t # u # v # us) (b # c # bs')
  using less.premis(1) bs-cons bs0-cons ts by simp
have t01: t ∈ {0..1} and u01: u ∈ {0..1} and tu: t < u
  and seg-tu: (if b then subpathin t u p ' {0..1} ⊆ U else subpathin t u p '
{0..1} ⊆ V)
  and cp-tail: cover-partition p (u # v # us) (c # bs')
  using cp-step by (rule cover-partition-step-props)+
show ?thesis
proof (cases b = c)
case True
note bits-eq = True
have uv: u < v
  using cp-tail by simp
have tu-le: t ≤ u
  using tu by simp
have uv-le: u ≤ v
  using uv by simp
have seg-tv: (if b then subpathin t v p ' {0..1} ⊆ U else subpathin t v p '
{0..1} ⊆ V)
proof (cases b)
case True
then have leftU: subpathin t u p ' {0..1} ⊆ U
  using seg-tu by simp
have rightU: subpathin u v p ' {0..1} ⊆ U
  using cp-tail bits-eq True by simp
have seg-U: subpathin t v p ' {0..1} ⊆ U
  by (rule subpathin-image-subset-trans[OF tu-le uv-le leftU rightU])
then show ?thesis
  using True by simp
next
case False
then have leftV: subpathin t u p ' {0..1} ⊆ V
  using seg-tu by simp
have rightV: subpathin u v p ' {0..1} ⊆ V
  using cp-tail bits-eq False by simp
have seg-V: subpathin t v p ' {0..1} ⊆ V
  by (rule subpathin-image-subset-trans[OF tu-le uv-le leftV rightV])
then show ?thesis
  using False by simp
qed
have cp-merged: cover-partition p (t # v # us) (b # bs')
  using cp-tail t01 seg-tv tu u01 by simp
have shorter: length (b # bs') < length bs
  using bs-eq by simp
have plast-merged: p (last (t # v # us)) ∈ U ∩ V
  using less.premis(3) unfolding ts by simp
have merged-part: ∃ ts' bs''. svk-partition p (t # ts') bs''

```

```

    by (rule less.hyps[of b # bs' t v # us]) (use shorter cp-merged less.prem(2)
plast-merged in simp-all)
    from merged-part obtain ts' bs'' where svk-partition p (t # ts') bs''
      by blast
    then show ?thesis
      by blast
  next
  case False
  have puUV: p u ∈ U ∩ V
    by (rule cover-partition-switch-point[OF cp-step False])
  have shorter: length (c # bs') < length bs
    using bs-eq by simp
  have plast-tail: p (last (u # v # us)) ∈ U ∩ V
    using less.prem(3) unfolding ts by simp
  have tail-exists: ∃ us' bs''. svk-partition p (u # us') bs''
    by (rule less.hyps[of c # bs' u v # us]) (use shorter cp-tail puUV plast-tail
in simp-all)
  from tail-exists obtain us' bs'' where tail-part: svk-partition p (u # us')
bs''
    by blast
  have svk-partition p (t # u # us') (b # bs'')
    using t01 u01 tu seg-tu less.prem(2) puUV tail-part by simp
  then show ?thesis
    by blast
qed
qed
qed
qed

```

```

lemma cover-partition-last-eq-one:
  assumes cp: cover-partition p ts bs
    and ts-ne: ts ≠ []
  shows last ts = 1
    using cp ts-ne
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by simp
next
  case (Cons t ts)
  show ?case
  proof (cases ts)
    case Nil
    then show ?thesis
      using Cons.prem by (cases bs) simp-all
  next
  case (Cons u us)
  then obtain b bs' where bs: bs = b # bs'
    using Cons.prem by (cases bs) auto

```

```

have tail: cover-partition p (u # us) bs'
  using Cons.prem Cons bs by simp
have ts-ne-tail: ts ≠ []
  using Cons by simp
have last-ts: last ts = 1
  by (rule Cons.IH[of bs']) (use tail ts-ne-tail Cons in simp-all)
have last-tail: last (u # us) = 1
  using last-ts Cons by simp
then show ?thesis
  using Cons by simp
qed

```

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 -
  have n-real-pos: 0 < real n
  proof -
    have real 0 < real n
      using n-pos by (rule less-imp-of-nat-less)
    then show ?thesis
      by simp
  qed
  have lower: 0 ≤ real i / real n
    using n-real-pos by (simp add: zero-le-divide-iff)
  have i-real-le: real i ≤ real n
    using i-le by simp
  have upper: real i / real n ≤ 1
  proof -
    have real i / real n ≤ real n / real n
      using i-real-le n-real-pos by (intro divide-right-mono) simp-all
    then show ?thesis
      using n-real-pos by simp
  qed
  show ?thesis
    using lower upper by auto
qed

```

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 -
  have real i < real (Suc i)
    by simp
  then show ?thesis
    using n-pos by (simp add: divide-strict-right-mono)

```

qed

```
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)
using k-le cover
proof (induction k)
  case 0
  then show ?case
  by simp
next
  case (Suc k)
  have k-le-n: k ≤ n
  using Suc.prem1 by simp
  have i0-lt-n: n - Suc k < n
  using Suc.prem1 n-pos by arith
  have i0-le-n: n - Suc k ≤ n
  by arith
  have i1-le-n: n - k ≤ n
  by arith
  have nk: n - k = Suc (n - Suc k)
  using Suc.prem1 by arith
  have t01: real (n - Suc k) / real n ∈ {0..1}
  by (rule nat-real-div-in-unit-interval[OF n-pos i0-le-n])
  have u01: real (n - k) / real n ∈ {0..1}
  by (rule nat-real-div-in-unit-interval[OF n-pos i1-le-n])
  have tu: real (n - Suc k) / real n < real (n - k) / real n
  unfolding nk by (rule nat-real-div-strict-mono[OF n-pos i0-lt-n])
  have seg-side-suc:
    (if side (n - Suc k)
     then subpathin (real (n - Suc k) / real n) (real (Suc (n - Suc k)) / real n)
     else subpathin (real (n - Suc k) / real n) (real (Suc (n - Suc k)) / real n) p
     ‘ {0..1} ⊆ V)
  by (rule Suc.prem1(2)[of n - Suc k]) (use i0-lt-n in simp-all)
  have seg-side:
```

```

    (if side (n - Suc k)
      then subpathin (real (n - Suc k) / real n) (real (n - k) / real n) p ‘ {0..1}
    ⊆ U
      else subpathin (real (n - Suc k) / real n) (real (n - k) / real n) p ‘ {0..1}
    ⊆ V)
  proof (cases side (n - Suc k))
    case True
      then show ?thesis
        using seg-side-suc nk by simp
    next
      case False
        then show ?thesis
          using seg-side-suc nk by simp
  qed
  have tail-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)
  proof -
    fix i
    assume i-lb: n - k ≤ i
      and i-lt: i < n
    have i-lb': n - Suc k ≤ i
      using i-lb by arith
    show (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)
      by (rule Suc.prem(2))[OF i-lb' i-lt]
  qed
  have tail-cp: cover-partition p (subdivision-times n k) (subdivision-bits side n k)
    by (rule Suc.IH[OF k-le-n tail-cover])
  show ?case
  proof (cases k)
    case 0
      have t0-nonneg: 0 ≤ real (n - Suc 0) / real n
        and t0-le1: real (n - Suc 0) / real n ≤ 1
        using t01 0 by auto
      have tu0: real (n - Suc 0) / real n < 1
        using tu 0 n-pos by simp
      have seg0-raw:
        (if side (n - Suc 0)
          then subpathin (real (n - Suc 0) / real n) (real (n - 0) / real n) p ‘ {0..1}
        ⊆ U
          else subpathin (real (n - Suc 0) / real n) (real (n - 0) / real n) p ‘ {0..1}
        ⊆ V)
        using seg-side
      unfolding 0
      by simp
  end

```

```

have end-eq: real (n - 0) / real n = 1
  using n-pos by simp
have seg0:
  (if side (n - Suc 0)
    then subpathin (real (n - Suc 0) / real n) 1 p ‘ {0..1} ⊆ U
    else subpathin (real (n - Suc 0) / real n) 1 p ‘ {0..1} ⊆ V)
  using seg0-raw end-eq by (cases side (n - Suc 0)) simp-all
have tail0: cover-partition p [1] []
  using tail-cp 0 by simp
then show ?thesis
  unfolding 0 subdivision-times.simps subdivision-bits.simps
  using t0-nonneg t0-le1 tu0 seg0 tail0
  by simp
next
case (Suc j)
have t01-suc: real (n - Suc (Suc j)) / real n ∈ {0..1}
  using t01 by (simp only: Suc)
have u01-suc: real (n - Suc j) / real n ∈ {0..1}
  using u01 by (simp only: Suc)
have tu-suc: real (n - Suc (Suc j)) / real n < real (n - Suc j) / real n
  using tu by (simp only: Suc)
have seg-suc:
  (if side (n - Suc (Suc j))
    then subpathin (real (n - Suc (Suc j)) / real n) (real (n - Suc j) / real n)
    else subpathin (real (n - Suc (Suc j)) / real n) (real (n - Suc j) / real n)
  p ‘ {0..1} ⊆ U
  p ‘ {0..1} ⊆ V)
  using seg-side by (simp only: Suc)
have tail-suc:
  cover-partition p
  (real (n - Suc j) / real n # subdivision-times n j)
  (side (n - Suc j) # subdivision-bits side n j)
  using tail-cp by (simp only: Suc subdivision-times.simps subdivi-
sion-bits.simps)
have times-suc:
  subdivision-times n (Suc (Suc j)) =
  real (n - Suc (Suc j)) / real n # real (n - Suc j) / real n # subdivision-times
n j
  by simp
have bits-suc:
  subdivision-bits side n (Suc (Suc j)) =
  side (n - Suc (Suc j)) # side (n - Suc j) # subdivision-bits side n j
  by simp
show ?thesis
  unfolding Suc times-suc bits-suc
  by (rule cover-partition-consI[OF t01-suc u01-suc tu-suc seg-suc tail-suc])
qed
qed

```

```

lemma subdivision-times-start:
  assumes n-pos:  $0 < n$ 
  shows subdivision-times  $n\ n = 0 \# \text{subdivision-times } n\ (n - 1)$ 
  using n-pos by (cases n) simp-all

lemma loop-has-valid-partition:
  assumes p-loop:  $p \in \text{loop-space } W\ x0$ 
  shows  $\exists ts\ bs.\ \text{valid-partition } p\ ts\ bs$ 
proof –
  obtain n :: nat where n-pos:  $0 < n$ 
  and n-cover:
     $\forall i < n.$ 
       $\text{subpathin } (\text{real } i / \text{real } n)\ (\text{real } (\text{Suc } i) / \text{real } n)\ p\ \text{' } \{0..1\} \subseteq U \vee$ 
       $\text{subpathin } (\text{real } i / \text{real } n)\ (\text{real } (\text{Suc } i) / \text{real } n)\ p\ \text{' } \{0..1\} \subseteq V$ 
  using loop-subdivision-by-cover[OF p-loop] by blast
  let ?side =  $\lambda i.\ \text{subpathin } (\text{real } i / \text{real } n)\ (\text{real } (\text{Suc } i) / \text{real } n)\ p\ \text{' } \{0..1\} \subseteq U$ 
  have raw: cover-partition p (subdivision-times  $n\ n$ ) (subdivision-bits ?side  $n\ n$ )
  proof (rule cover-partition-subdivision-from[OF n-pos le-refl])
    fix i
    assume  $n - n \leq i$  and i-lt:  $i < n$ 
    from n-cover[rule-format, OF i-lt]
    show (if ?side i
      then  $\text{subpathin } (\text{real } i / \text{real } n)\ (\text{real } (\text{Suc } i) / \text{real } n)\ p\ \text{' } \{0..1\} \subseteq U$ 
      else  $\text{subpathin } (\text{real } i / \text{real } n)\ (\text{real } (\text{Suc } i) / \text{real } n)\ p\ \text{' } \{0..1\} \subseteq V$ 
      by auto
    )
  qed
  have times0: subdivision-times  $n\ n = 0 \# \text{subdivision-times } n\ (n - 1)$ 
    by (rule subdivision-times-start[OF n-pos])
  have times-ne: subdivision-times  $n\ n \neq []$ 
    using times0 by simp
  have last1: last (subdivision-times  $n\ n$ ) = 1
    by (rule cover-partition-last-eq-one[OF raw times-ne])
  have p0UV:  $p\ 0 \in U \cap V$  and p1UV:  $p\ 1 \in U \cap V$ 
    using p-loop x0-in-UV unfolding loop-space-def pathstart-def pathfinish-def by
auto
  have raw0: cover-partition p ( $0 \# \text{subdivision-times } n\ (n - 1)$ ) (subdivision-bits
?side  $n\ n$ )
    using raw times0 by simp
  have plastUV:  $p\ (\text{last } (0 \# \text{subdivision-times } n\ (n - 1))) \in U \cap V$ 
    using p1UV last1 times0 by simp
  from cover-partition-compress-svk-partition[OF raw0 p0UV plastUV]
  obtain ts' bs' where part: svk-partition p ( $0 \# ts'$ ) bs'
    by blast
  have valid-partition p ( $0 \# ts'$ ) bs'
    unfolding valid-partition-def using part by simp
  then show ?thesis
    by blast
qed

```

```

lemma svk-partition-partition-word-in-space:
  assumes p-loop:  $p \in \text{loop-space } W \ x0$ 
    and part: svk-partition  $p \ ts \ bs$ 
  shows fpw-in-space  $G1 \ G2$  (partition-word  $p \ ts \ bs$ )
  using part
proof (induction  $ts$  arbitrary:  $bs$ )
  case Nil
  then show ?case
    by simp
next
  case (Cons  $t \ ts$ )
  from p-loop have p-path: path  $p$  and p-image: path-image  $p \subseteq W$ 
    unfolding loop-space-def by auto
  show ?case
  proof (cases  $ts$ )
    case Nil
    then show ?thesis
      using Cons.prems by (cases  $bs$ ) simp-all
    next
    case (Cons  $u \ us$ )
    then obtain  $b \ bs'$  where  $bs = b \ \# \ bs'$ 
      using Cons.prems by (cases  $bs$ ) auto
    have tail: svk-partition  $p \ (u \ \# \ us) \ bs'$ 
      using Cons.prems Cons  $bs$  by simp
    have tail-in: fpw-in-space  $G1 \ G2$  (partition-word  $p \ (u \ \# \ us) \ bs'$ )
      using p-loop Cons.IH[of  $bs'$ ] Cons tail by simp
    have  $t01$ :  $t \in \{0..1\}$  and  $ptUV$ :  $p \ t \in U \cap V$ 
      using Cons.prems Cons  $bs$  by simp-all
    have  $u01$ :  $u \in \{0..1\}$  and seg-side:
      (if  $b$  then subpathin  $t \ u \ p \ \{0..1\} \subseteq U$  else subpathin  $t \ u \ p \ \{0..1\} \subseteq V$ )
      using Cons.prems Cons  $bs$  by simp-all
    have  $puUV$ :  $p \ u \in U \cap V$ 
      using svk-partition-next-in-intersection[of  $p \ t \ u \ us \ b \ bs'$ ] Cons.prems Cons  $bs$ 
by simp
    show ?thesis
    proof (cases  $b$ )
      case True
      have segU: segment-loop  $p \ t \ u \in \text{loop-space } U \ x0$ 
        by (rule segment-loop-in-U[OF  $p$ -path  $p$ -image  $t01 \ u01 \ ptUV \ puUV$ ]) (use
seg-side True in simp)
      have loop-class  $U \ x0$  (segment-loop  $p \ t \ u$ )  $\in G1$ 
        by (rule loop-class-in-space[OF segU])
      then show ?thesis
        using tail-in True  $bs \ Cons$  by simp
      next
      case False
      have segV: segment-loop  $p \ t \ u \in \text{loop-space } V \ x0$ 
        by (rule segment-loop-in-V[OF  $p$ -path  $p$ -image  $t01 \ u01 \ ptUV \ puUV$ ]) (use
seg-side False in simp)

```

```

    have loop-class  $V x0$  (segment-loop  $p t u$ )  $\in G2$ 
      by (rule loop-class-in-space[OF segV])
    then show ?thesis
      using tail-in False bs Cons by simp
  qed
qed
qed

lemma valid-partition-partition-word-in-space:
  assumes p-loop:  $p \in \text{loop-space } W x0$ 
    and part: valid-partition  $p ts bs$ 
  shows fpw-in-space  $G1 G2$  (partition-word  $p ts bs$ )
  using assms unfolding valid-partition-def
  by (auto intro: svk-partition-partition-word-in-space)

lemma svk-partition-partition-loop-in-W:
  assumes p-loop:  $p \in \text{loop-space } W x0$ 
    and part: svk-partition  $p ts bs$ 
  shows partition-loop  $p ts \in \text{loop-space } W x0$ 
  using part
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by (simp add: constant-loop-in-space[OF x0-in-W])
next
  case (Cons t ts)
  from p-loop have p-path: path  $p$  and p-image: path-image  $p \subseteq W$ 
    unfolding loop-space-def by auto
  show ?case
  proof (cases ts)
    case Nil
    then show ?thesis
      by (simp add: constant-loop-in-space[OF x0-in-W])
  next
    case (Cons u us)
    then obtain  $b bs'$  where  $bs = b \# bs'$ 
      using Cons.prem1 by (cases bs) auto
    have tail: svk-partition  $p (u \# us) bs'$ 
      using Cons.prem2 Cons bs by simp
    have tail-loop: partition-loop  $p (u \# us) \in \text{loop-space } W x0$ 
      using p-loop Cons.IH[of bs'] Cons tail by simp
    have t01:  $t \in \{0..1\}$  and ptUV:  $p t \in U \cap V$ 
      using Cons.prem3 Cons bs by simp-all
    have u01:  $u \in \{0..1\}$  and puUV:  $p u \in U \cap V$ 
      using Cons.prem4 Cons bs svk-partition-next-in-intersection[of p t u us b bs']
  by simp-all
  have segW:
    segment-loop  $p t u \in \text{loop-space } W x0$ 
  proof (rule segment-loop-in-W[OF p-path p-image t01 u01 ptUV puUV])

```

**have** *sub-imgW*: *path-image* (*subpathin* *t u p*)  $\subseteq W$   
**using** *p-image path-image-subpathin-subset*[*OF t01 u01, of p*] **by** *blast*  
**show** *subpathin* *t u p* ‘  $\{0..1\} \subseteq W$   
**using** *sub-imgW* **by** (*simp add: path-image-def*)  
**qed**  
**have** *joined-loop*:  
*segment-loop* *p t u* +++ *partition-loop* *p (u # us)*  $\in$  *loop-space* *W x0*  
**by** (*rule loop-space-join*[*OF segW tail-loop*])  
**show** *?thesis*  
**using** *joined-loop Cons* **by** *simp*  
**qed**  
**qed**

**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*  
**using** *assms unfolding valid-partition-def*  
**by** (*auto intro: svk-partition-partition-loop-in-W*)

**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** –  
**have** *A-in*: *loop-class* ( $U \cap V$ ) *x0 p*  $\in$  *fundamental-group-space* ( $U \cap V$ ) *x0*  
**by** (*rule loop-class-in-space*[*OF p-loop*])  
**have** *i1* (*loop-class* ( $U \cap V$ ) *x0 p*) = *loop-class* *U x0 (loop-image id p)*  
**proof** (*rule fundamental-group-map-eqI*)  
**show** *loop-class* ( $U \cap V$ ) *x0 p*  $\in$  *fundamental-group-space* ( $U \cap V$ ) *x0*  
**by** (*rule A-in*)  
**show**  $p \in$  *loop-space* ( $U \cap V$ ) *x0*  
**by** (*rule p-loop*)  
**show** *loop-class* ( $U \cap V$ ) *x0 p* = *loop-class* ( $U \cap V$ ) *x0 p*  
**by** *simp*  
**show** *continuous-on* ( $U \cap V$ ) *id*  
**by** *simp*  
**show** *id*  $\in$  ( $U \cap V$ )  $\rightarrow$  *U*  
**by** *auto*  
**show** *id x0* = *x0*  
**by** *simp*  
**qed**  
**then show** *?thesis*  
**by** (*simp add: loop-image-def*)  
**qed**

**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** –

**have** *A-in*:  $\text{loop-class } (U \cap V) x0 p \in \text{fundamental-group-space } (U \cap V) x0$   
**by** (rule *loop-class-in-space*[*OF p-loop*])  
**have** *i2* ( $\text{loop-class } (U \cap V) x0 p = \text{loop-class } V x0 (\text{loop-image } id p)$ )  
**proof** (rule *fundamental-group-map-eqI*)  
**show**  $\text{loop-class } (U \cap V) x0 p \in \text{fundamental-group-space } (U \cap V) x0$   
**by** (rule *A-in*)  
**show**  $p \in \text{loop-space } (U \cap V) x0$   
**by** (rule *p-loop*)  
**show**  $\text{loop-class } (U \cap V) x0 p = \text{loop-class } (U \cap V) x0 p$   
**by** *simp*  
**show** *continuous-on*  $(U \cap V) id$   
**by** *simp*  
**show**  $id \in (U \cap V) \rightarrow V$   
**by** *auto*  
**show**  $id x0 = x0$   
**by** *simp*  
**qed**  
**then show** *?thesis*  
**by** (*simp add: loop-image-def*)  
**qed**

**lemma** *j1-segment-loop-eq*:  
**assumes** *segU*:  $\text{segment-loop } p t u \in \text{loop-space } U x0$   
**shows** *j1* ( $\text{loop-class } U x0 (\text{segment-loop } p t u) = \text{loop-class } W x0 (\text{segment-loop } p t u)$ )  
**proof** –  
**have** *A-in*:  $\text{loop-class } U x0 (\text{segment-loop } p t u) \in \text{fundamental-group-space } U x0$   
**by** (rule *loop-class-in-space*[*OF segU*])  
**have** *j1* ( $\text{loop-class } U x0 (\text{segment-loop } p t u) = \text{loop-class } W x0 (\text{loop-image } id (\text{segment-loop } p t u))$ )  
**proof** (rule *fundamental-group-map-eqI*)  
**show**  $\text{loop-class } U x0 (\text{segment-loop } p t u) \in \text{fundamental-group-space } U x0$   
**by** (rule *A-in*)  
**show**  $\text{segment-loop } p t u \in \text{loop-space } U x0$   
**by** (rule *segU*)  
**show**  $\text{loop-class } U x0 (\text{segment-loop } p t u) = \text{loop-class } U x0 (\text{segment-loop } p t u)$   
**by** *simp*  
**show** *continuous-on*  $U id$   
**by** *simp*  
**show**  $id \in U \rightarrow W$   
**by** *auto*  
**show**  $id x0 = x0$   
**by** *simp*  
**qed**  
**then show** *?thesis*  
**by** (*simp add: loop-image-def*)  
**qed**

```

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 –
  have A-in: loop-class V x0 (segment-loop p t u)  $\in$  fundamental-group-space V x0
    by (rule loop-class-in-space[OF segV])
  have j2 (loop-class V x0 (segment-loop p t u)) =
    loop-class W x0 (loop-image id (segment-loop p t u))
proof (rule fundamental-group-map-eqI)
  show loop-class V x0 (segment-loop p t u)  $\in$  fundamental-group-space V x0
    by (rule A-in)
  show segment-loop p t u  $\in$  loop-space V x0
    by (rule segV)
  show loop-class V x0 (segment-loop p t u) =
    loop-class V x0 (segment-loop p t u)
    by simp
  show continuous-on V id
    by simp
  show id  $\in$  V  $\rightarrow$  W
    by auto
  show id x0 = x0
    by simp
qed
then show ?thesis
  by (simp add: loop-image-def)
qed

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)
  using part
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by (simp add: fundamental-group-one-def)
next
  case (Cons t ts)
  from p-loop have p-path: path p and p-image: path-image p  $\subseteq$  W
    unfolding loop-space-def by auto
  show ?case
proof (cases ts)
  case ts-nil: Nil
  then show ?thesis
proof (cases bs)
  case bs-nil: Nil
  have pw: partition-word p (t # ts) bs = WordNil

```

```

    using ts-nil bs-nil by simp
  have pl: partition-loop p (t # ts) = (λ-. x0)
    using ts-nil by simp
  have eval-nil0: svk-word-eval WordNil = oneW
    by (rule decode.carrier-full-amalg-eval.simps(1))
  have eval-nil: svk-word-eval WordNil = loop-class W x0 (λ-. x0)
    using eval-nil0 by (simp add: fundamental-group-one-def)
  have svk-word-eval (partition-word p (t # ts) bs) = svk-word-eval WordNil
    using pw by simp
  also have ... = loop-class W x0 (λ-. x0)
    by (rule eval-nil)
  also have ... = loop-class W x0 (partition-loop p (t # ts))
    using pl by simp
  finally show ?thesis
.
next
case bs-cons: (Cons b bs')
then show ?thesis
  using Cons.premts ts-nil by simp
qed
next
case (Cons u us)
then obtain b bs' where bs: bs = b # bs'
  using Cons.premts by (cases bs) auto
have tail: svk-partition p (u # us) bs'
  using Cons.premts Cons bs by simp
have tail-ts: svk-partition p ts bs'
  using Cons tail by simp
have tail-eval-ts:
  svk-word-eval (partition-word p ts bs') =
  loop-class W x0 (partition-loop p ts)
  by (rule Cons.IH[OF tail-ts])
have tail-eval:
  svk-word-eval (partition-word p (u # us) bs') =
  loop-class W x0 (partition-loop p (u # us))
  using Cons tail-eval-ts by simp
have tail-loop: partition-loop p (u # us) ∈ loop-space W x0
  by (rule svk-partition-partition-loop-in-W[OF p-loop tail])
have t01: t ∈ {0..1} and ptUV: p t ∈ U ∩ V
  using Cons.premts Cons bs by simp-all
have u01: u ∈ {0..1} and puUV: p u ∈ U ∩ V
  using Cons.premts Cons bs svk-partition-next-in-intersection[of p t u us b bs']
by simp-all
have seg-side:
  (if b then subpathin t u p ' {0..1} ⊆ U else subpathin t u p ' {0..1} ⊆ V)
  using Cons.premts Cons bs by simp
have segW:
  segment-loop p t u ∈ loop-space W x0
proof (rule segment-loop-in-W[OF p-path p-image t01 u01 ptUV puUV])

```

```

have sub-imgW: path-image (subpathin t u p)  $\subseteq$  W
  using p-image path-image-subpathin-subset[OF t01 u01, of p] by blast
show subpathin t u p ‘ {0..1}  $\subseteq$  W
  using sub-imgW by (simp add: path-image-def)
qed
have segW-in: loop-class W x0 (segment-loop p t u)  $\in$  fundamental-group-space
W x0
  by (rule loop-class-in-space[OF segW])
  have tail-in: loop-class W x0 (partition-loop p (u # us))  $\in$  fundamen-
tal-group-space W x0
  by (rule loop-class-in-space[OF tail-loop])
have mult-eq:
  multW (loop-class W x0 (segment-loop p t u))
    (loop-class W x0 (partition-loop p (u # us))) =
  loop-class W x0 (segment-loop p t u +++ partition-loop p (u # us))
  by (rule fundamental-group-mult-eqI[OF segW-in tail-in segW tail-loop])
simp-all
show ?thesis
proof (cases b)
  case True
  have segU: segment-loop p t u  $\in$  loop-space U x0
  by (rule segment-loop-in-U[OF p-path p-image t01 u01 ptUV puUV]) (use
seg-side True in simp)
  have j1-eq:
  j1 (loop-class U x0 (segment-loop p t u)) =
  loop-class W x0 (segment-loop p t u)
  by (rule j1-segment-loop-eq[OF segU])
show ?thesis
proof -
  have svk-word-eval (partition-word p (t # ts) (True # bs')) =
  multW (j1 (loop-class U x0 (segment-loop p t u)))
  (svk-word-eval (partition-word p (u # us) bs'))
  using Cons by simp
  also have ... =
  multW (loop-class W x0 (segment-loop p t u))
  (loop-class W x0 (partition-loop p (u # us)))
  using j1-eq tail-eval by simp
  also have ... =
  loop-class W x0 (segment-loop p t u +++ partition-loop p (u # us))
  by (rule mult-eq)
  also have ... = loop-class W x0 (partition-loop p (t # ts))
  using Cons by simp
  finally have branch-true:
  svk-word-eval (partition-word p (t # ts) (True # bs')) =
  loop-class W x0 (partition-loop p (t # ts)) .
  have bs-true: bs = True # bs'
  using bs True by simp
show ?thesis
  unfolding bs-true using branch-true .

```

```

qed
next
case False
have segV: segment-loop p t u ∈ loop-space V x0
  by (rule segment-loop-in-V[OF p-path p-image t01 u01 ptUV puUV]) (use
seg-side False in simp)
have j2-eq:
  j2 (loop-class V x0 (segment-loop p t u)) =
  loop-class W x0 (segment-loop p t u)
  by (rule j2-segment-loop-eq[OF segV])
show ?thesis
proof -
  have svk-word-eval (partition-word p (t # ts) (False # bs')) =
    multW (j2 (loop-class V x0 (segment-loop p t u)))
    (svk-word-eval (partition-word p (u # us) bs'))
  using Cons by simp
  also have ... =
    multW (loop-class W x0 (segment-loop p t u))
    (loop-class W x0 (partition-loop p (u # us)))
  using j2-eq tail-eval by simp
  also have ... =
    loop-class W x0 (segment-loop p t u +++ partition-loop p (u # us))
  by (rule mult-eq)
  also have ... = loop-class W x0 (partition-loop p (t # ts))
  using Cons by simp
  finally have branch-false:
    svk-word-eval (partition-word p (t # ts) (False # bs')) =
    loop-class W x0 (partition-loop p (t # ts)) .
  have bs-false: bs = False # bs'
  using bs False by simp
  show ?thesis
  unfolding bs-false using branch-false .
qed
qed
qed
qed

lemma valid-partition-eval-partition-word:
  assumes p-loop: p ∈ 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 -
  have svk-part: svk-partition p ts bs
  using part unfolding valid-partition-def by auto
  show ?thesis
  by (rule svk-partition-eval-partition-word[OF p-loop svk-part])
qed

```

**lemma** *valid-partition-decode-partition-word*:  
**assumes** *p-loop*:  $p \in \text{loop-space } W \ x0$   
**and** *part*: *valid-partition*  $p \ ts \ bs$   
**shows** *svk-decode* (*partition-word*  $p \ ts \ bs$ ) =  
*loop-class*  $W \ x0 \ (\text{partition-loop } p \ ts)$   
**proof** –  
**have** *in-space*: *fpw-in-space*  $G1 \ G2 \ (\text{partition-word } p \ ts \ bs)$   
**by** (*rule valid-partition-partition-word-in-space*[*OF p-loop part*])  
**show** *?thesis*  
**using** *valid-partition-eval-partition-word*[*OF p-loop part*]  
**by** (*simp add: svk-decode-eq-eval*[*OF in-space*])  
**qed**

**lemma** *pair-interval-member*:  
**fixes**  $x \ y :: \text{real} \times \text{real}$  **and**  $x1 \ x2 \ y1 \ y2 \ u \ v :: \text{real}$   
**assumes**  $x: x = (x1, x2)$   
**and**  $y: y = (y1, y2)$   
**and**  $\text{mix1}: u *_R x1 + v *_R y1 \in \{0..1\}$   
**and**  $\text{mix2}: u *_R x2 + v *_R y2 \in \{0..1\}$   
**shows**  $u *_R x + v *_R y \in \{0..1\} \times \{0..1\}$   
**proof** –  
**have** *pair-form*:  
 $u *_R x + v *_R y =$   
 $(u *_R x1 + v *_R y1, u *_R x2 + v *_R y2)$   
**using**  $x \ y$  **by** *simp*  
**have** *pair-in-Q*:  
 $(u *_R x1 + v *_R y1, u *_R x2 + v *_R y2) \in \{0..1\} \times \{0..1\}$   
**using**  $\text{mix1} \ \text{mix2}$  **by** *auto*  
**from** *pair-in-Q* **show** *?thesis*  
**by** (*subst pair-form*) *simp*  
**qed**

**lemma** *affine-closed-segment-member*:  
**fixes**  $a \ b \ u :: \text{real}$   
**assumes**  $u01: u \in \{0..1\}$   
**shows**  $(b - a) * u + a \in \text{closed-segment } a \ b$   
**proof** –  
**have**  $(b - a) * u + a = \text{linepath } a \ b \ u$   
**by** (*simp add: linepath-def algebra-simps*)  
**moreover** **have**  $\text{linepath } a \ b \ u \in \text{closed-segment } a \ b$   
**using**  $u01$  **by** (*rule linepath-in-path*)  
**ultimately** **show** *?thesis*  
**by** *simp*  
**qed**

**lemma** *affine-subinterval-member*:  
**fixes**  $a \ b \ u :: \text{real}$   
**assumes**  $ab: a \leq b$   
**and**  $u01: u \in \{0..1\}$

**shows**  $(b - a) * u + a \in \{a..b\}$   
**proof** –  
**have**  $(b - a) * u + a \in \text{closed-segment } a \ b$   
**by** (rule *affine-closed-segment-member*[OF u01])  
**also have**  $\text{closed-segment } a \ b = \{a..b\}$   
**by** (rule *closed-segment-eq-real-ivl1*[OF ab])  
**finally show** ?thesis .  
**qed**

**lemma** *affine-unit-interval-member*:

**fixes**  $a \ b \ u :: \text{real}$   
**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** –  
**have**  $(b - a) * u + a \in \{a..b\}$   
**by** (rule *affine-subinterval-member*[OF ab u01])  
**moreover have**  $\{a..b\} \subseteq \{0..1\}$   
**using**  $a01 \ b01 \ ab$  **by** *auto*  
**ultimately show** ?thesis  
**by** *blast*  
**qed**

**lemma** *square-edge-homotopic*:

**fixes**  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$   
**assumes**  $h\text{-cont}: \text{continuous-map } (\text{top-of-set } (\{0..1\} \times \{0..1\})) (\text{top-of-set } S) \ h$   
**shows** *homotopic-paths*  $S$   
 $((\lambda t. h \ (t, 0)) \ +++ \ (\lambda t. h \ (1, t)))$   
 $((\lambda t. h \ (0, t)) \ +++ \ (\lambda t. h \ (t, 1)))$   
**proof** –  
**let**  $?g = \text{linepath } (0::\text{real}, 0::\text{real}) \ (1, 0) \ +++ \ \text{linepath } (1, 0) \ (1, 1)$   
**let**  $?k = \text{linepath } (0::\text{real}, 0::\text{real}) \ (0, 1) \ +++ \ \text{linepath } (0, 1) \ (1, 1)$   
**have**  $hk:$   
 $\text{homotopic-paths } (\{0..1\} \times \{0..1\}) \ ?g \ ?k$   
**proof** (rule *homotopic-paths-linear*)  
**show**  $\text{path } ?g \ \text{path } ?k$   
**by** *simp-all*  
**show**  $\text{pathstart } ?k = \text{pathstart } ?g \ \text{pathfinish } ?k = \text{pathfinish } ?g$   
**by** *simp-all*  
**show**  $\text{closed-segment } (?g \ t) \ (?k \ t) \subseteq \{0..1\} \times \{0..1\}$  **if**  $t \in \{0..1\}$  **for**  $t$   
**proof** (rule *closed-segment-subset*)  
**show**  $?g \ t \in \{0..1\} \times \{0..1\} \ ?k \ t \in \{0..1\} \times \{0..1\}$   
**using** *that* **by** (*auto simp: joinpaths-def linepath-def*)  
**show**  $\text{convex } ((\{0::\text{real}..1\}) \times (\{0::\text{real}..1\}))$   
**by** (*intro convex-Times*) *auto*  
**qed**  
**qed**

```

from h-cont have h-on: continuous-on ( $\{0..1\} \times \{0..1\}$ ) h
  and h-into:  $h \in (\{0..1\} \times \{0..1\}) \rightarrow S$ 
  by simp-all
have img:
  homotopic-paths  $S$  ( $h \circ ?g$ ) ( $h \circ ?k$ )
  by (rule homotopic-paths-continuous-image[OF hk h-on h-into])
have g-eq:  $h \circ ?g = ((\lambda t. h (t, 0)) +++ (\lambda t. h (1, t)))$ 
  by (rule ext) (simp add: joinpaths-def linepath-def)
have k-eq:  $h \circ ?k = ((\lambda t. h (0, t)) +++ (\lambda t. h (t, 1)))$ 
  by (rule ext) (simp add: joinpaths-def linepath-def)
show ?thesis
  using img unfolding g-eq k-eq .
qed

lemma rectangle-edge-homotopic:
  fixes  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$ 
  assumes h-cont: continuous-map (top-of-set ( $\{0..1\} \times \{0..1\}$ )) (top-of-set  $S$ ) 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:  $c \leq d$ 
  shows homotopic-paths  $S$ 
    (subpathin  $a$   $b$   $(\lambda t. h (t, c)) +++ (\lambda t. h (b, (d - c) * t + c))$ )
     $((\lambda t. h (a, (d - c) * t + c)) +++ \text{subpathin } a \ b (\lambda t. h (t, d)))$ 
proof -
  let  $?Q = \{0..1\} \times \{0..1\}$ 
  let  $?r = \lambda z :: \text{real} \times \text{real}. ((b - a) * \text{fst } z + a, (d - c) * \text{snd } z + c)$ 
  have r-on: continuous-on  $?Q$   $?r$ 
    by (intro continuous-intros)
  have r-into:  $?r \in ?Q \rightarrow ?Q$ 
proof
  fix  $z :: \text{real} \times \text{real}$ 
  assume z-in:  $z \in ?Q$ 
  then have z1:  $\text{fst } z \in \{0..1\}$  and z2:  $\text{snd } z \in \{0..1\}$ 
    by auto
  have x-in:  $(b - a) * \text{fst } z + a \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF a01 b01 ab z1])
  have y-in:  $(d - c) * \text{snd } z + c \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF c01 d01 cd z2])
  show  $?r z \in ?Q$ 
    using x-in y-in by simp
qed
from h-cont have h-on: continuous-on  $?Q$  h and h-into:  $h \in ?Q \rightarrow S$ 
  by simp-all
have hr-on: continuous-on  $?Q$  ( $h \circ ?r$ )
proof -
  have continuous-on  $?Q$   $(\lambda x. h (?r x))$ 
  proof (rule continuous-on-compose2[OF h-on])
    show continuous-on  $?Q$   $?r$ 
    by (rule r-on)

```

```

    show ?r ' ?Q ⊆ ?Q
      using r-into by auto
    qed
  then show ?thesis
    by (simp add: comp-def)
  qed
  have hr-into: (h ∘ ?r) ∈ ?Q → S
  proof
    fix z :: real × real
    assume z-in: z ∈ ?Q
    obtain x y where z: z = (x, y)
      by (cases z)
    have x01: x ∈ {0..1} and y01: y ∈ {0..1}
      using z-in z by auto
    have rx-in: (b - a) * x + a ∈ {0..1}
      by (rule affine-unit-interval-member[OF a01 b01 ab x01])
    have ry-in: (d - c) * y + c ∈ {0..1}
      by (rule affine-unit-interval-member[OF c01 d01 cd y01])
    have rz-in: ?r z ∈ ?Q
      using z rx-in ry-in by simp
    have hz-in: h (?r z) ∈ S
      using h-into rz-in by auto
    show (h ∘ ?r) z ∈ S
      using hz-in by (simp add: comp-def)
  qed
  have hr-cont: continuous-map (top-of-set ?Q) (top-of-set S) (h ∘ ?r)
    using hr-on hr-into by simp
  have base:
    homotopic-paths S
      ((λt. (h ∘ ?r) (t, 0)) +++ (λt. (h ∘ ?r) (1, t)))
      ((λt. (h ∘ ?r) (0, t)) +++ (λt. (h ∘ ?r) (t, 1)))
    by (rule square-edge-homotopic[OF hr-cont])
  have left-eq:
    ((λt. (h ∘ ?r) (t, 0)) +++ (λt. (h ∘ ?r) (1, t))) =
      (subpathin a b (λt. h (t, c)) +++ (λt. h (b, (d - c) * t + c)))
    by (rule ext) (simp add: subpathin-def joinpaths-def)
  have right-eq:
    ((λt. (h ∘ ?r) (0, t)) +++ (λt. (h ∘ ?r) (t, 1))) =
      ((λt. h (a, (d - c) * t + c)) +++ subpathin a b (λt. h (t, d)))
    by (rule ext) (simp add: subpathin-def joinpaths-def)
  show ?thesis
  proof (subst left-eq[symmetric], subst right-eq[symmetric])
    show homotopic-paths S
      (((λt. (h ∘ ?r) (t, 0)) +++ (λt. (h ∘ ?r) (1, t))))
      (((λt. (h ∘ ?r) (0, t)) +++ (λt. (h ∘ ?r) (t, 1))))
      by (rule base)
  qed
  qed
  qed

```

```

lemma rectangle-edge-homotopic-in-set:
  fixes  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$ 
  assumes  $h\text{-cont}$ : 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  $rectS$ :  $h \text{ ` } (\{a..b\} \times \{c..d\}) \subseteq S$ 
  shows homotopic-paths  $S$ 
    (subpathin  $a\ b$  ( $\lambda t. h\ (t, c)$ ) +++ ( $\lambda t. h\ (b, (d - c) * t + c)$ ))
    (( $\lambda t. h\ (a, (d - c) * t + c)$ ) +++ subpathin  $a\ b$  ( $\lambda t. h\ (t, d)$ ))
proof -
  let  $?Q = \{0..1\} \times \{0..1\}$ 
  let  $?R = \{a..b\} \times \{c..d\}$ 
  let  $?r = \lambda z :: \text{real} \times \text{real}. ((b - a) * \text{fst } z + a, (d - c) * \text{snd } z + c)$ 
  have  $r\text{-on}$ : continuous-on  $?Q\ ?r$ 
    by (intro continuous-intros)
  have  $r\text{-into}$ :  $?r \in ?Q \rightarrow ?R$ 
proof
  fix  $z :: \text{real} \times \text{real}$ 
  assume  $z\text{-in}$ :  $z \in ?Q$ 
  then have  $z1$ :  $\text{fst } z \in \{0..1\}$  and  $z2$ :  $\text{snd } z \in \{0..1\}$ 
    by auto
  have  $x\text{-in}$ :  $(b - a) * \text{fst } z + a \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF  $a01\ b01\ ab\ z1$ ])
  have  $y\text{-in}$ :  $(d - c) * \text{snd } z + c \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF  $c01\ d01\ cd\ z2$ ])
  show  $?r\ z \in ?R$ 
    using  $x\text{-in}\ y\text{-in}$  by simp
qed
have  $r\text{-rect}$ :  $?r \text{ ` } ?Q \subseteq ?R$ 
proof
  fix  $x$ 
  assume  $x \in ?r \text{ ` } ?Q$ 
  then obtain  $z$  where  $z\text{-in}$ :  $z \in ?Q$  and  $x\text{-eq}$ :  $x = ?r\ z$ 
    by blast
  have  $x1$ :  $(b - a) * \text{fst } z + a \in \{a..b\}$ 
    by (rule affine-subinterval-member[OF  $ab$ ]) (use  $z\text{-in}$  in auto)
  have  $x2$ :  $(d - c) * \text{snd } z + c \in \{c..d\}$ 
    by (rule affine-subinterval-member[OF  $cd$ ]) (use  $z\text{-in}$  in auto)
  show  $x \in ?R$ 
    using  $x1\ x2\ x\text{-eq}$  by simp
qed
from  $h\text{-cont}$  have  $h\text{-on}$ : continuous-on  $?Q\ h$ 
  and  $h\text{-into}W$ :  $h \in ?Q \rightarrow W$ 
  by simp-all
have  $hr\text{-on}$ : continuous-on  $?Q\ (h \circ ?r)$ 
proof -
  have continuous-on  $?Q\ (\lambda x. h\ (?r\ x))$ 

```

```

proof (rule continuous-on-compose2[OF h-on])
  show continuous-on ?Q ?r
    by (rule r-on)
  show ?r ' ?Q  $\subseteq$  ?Q
    using r-into by auto
qed
then show ?thesis
  by (simp add: comp-def)
qed
have hr-into: (h  $\circ$  ?r)  $\in$  ?Q  $\rightarrow$  S
proof
  fix z :: real  $\times$  real
  assume z-in: z  $\in$  ?Q
  then have rz-in: ?r z  $\in$  ?R
  proof -
    have ?r z  $\in$  ?r ' ?Q
      using z-in by blast
    then show ?thesis
      using r-rect by blast
  qed
  show (h  $\circ$  ?r) z  $\in$  S
    using rectS rz-in by auto
qed
have hr-cont: continuous-map (top-of-set ?Q) (top-of-set S) (h  $\circ$  ?r)
  using hr-on hr-into by simp
have base:
  homotopic-paths S
  (( $\lambda$ t. (h  $\circ$  ?r) (t, 0)) +++ ( $\lambda$ t. (h  $\circ$  ?r) (1, t)))
  (( $\lambda$ t. (h  $\circ$  ?r) (0, t) +++ ( $\lambda$ t. (h  $\circ$  ?r) (t, 1)))
  by (rule square-edge-homotopic[OF hr-cont])
have left-eq:
  (( $\lambda$ t. (h  $\circ$  ?r) (t, 0)) +++ ( $\lambda$ t. (h  $\circ$  ?r) (1, t))) =
  (subpathin a b ( $\lambda$ t. h (t, c) +++ ( $\lambda$ t. h (b, (d - c) * t + c)))
  by (rule ext) (simp add: subpathin-def joinpaths-def)
have right-eq:
  (( $\lambda$ t. (h  $\circ$  ?r) (0, t) +++ ( $\lambda$ t. (h  $\circ$  ?r) (t, 1))) =
  (( $\lambda$ t. h (a, (d - c) * t + c) +++ subpathin a b ( $\lambda$ t. h (t, d)))
  by (rule ext) (simp add: subpathin-def joinpaths-def)
show ?thesis
  using base unfolding left-eq right-eq .
qed

definition vertical-strip-path ::
  ((real  $\times$  real)  $\Rightarrow$  'a)  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  'a
where
  vertical-strip-path h t c d = ( $\lambda$ u. h (t, (d - c) * u + c))

definition bridge-loop ::
  ((real  $\times$  real)  $\Rightarrow$  'a)  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  '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
unfolding bridge-loop-def vertical-strip-path-def segment-loop-def subpathin-def
by (rule ext) simp
```

**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**

```
fix x
assume x-in: x ∈ vertical-strip-path h t c d ' {0..1}
then obtain u where u01: u ∈ {0..1} and x-eq: x = vertical-strip-path h t c d
u
by blast
have yc: (d - c) * u + c ∈ {c..d}
by (rule affine-subinterval-member[OF cd u01])
show x ∈ h ' ({t} × {c..d})
using yc x-eq unfolding vertical-strip-path-def by auto
qed
```

**lemma** *rectangle-segment-loop-bridge-homotopic*:

```
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
and leftUV: h ' ({a} × {c..d}) ⊆ U ∩ V
and rightUV: h ' ({b} × {c..d}) ⊆ U ∩ V
and UVS: U ∩ V ⊆ S
```

**shows** *homotopic-paths* S

```
(segment-loop (λt. h (t, c)) a b +++ bridge-loop h b c d)
(bridge-loop h a c d +++ segment-loop (λt. h (t, d)) a b)
```

**proof** –

```
let ?pc = λt. h (t, c)
let ?pd = λt. h (t, d)
let ?la = vertical-strip-path h a c d
let ?lb = vertical-strip-path h b c d
let ?ac = connector (h (a, c))
let ?ad = connector (h (a, d))
let ?bc = connector (h (b, c))
let ?bd = connector (h (b, d))
let ?bot = subpathin a b ?pc
```

```

let ?top = subpathin a b ?pd
have h-on: continuous-on ({0..1} × {0..1}) h
  and h-intoW: h ∈ ({0..1} × {0..1}) → W
  using h-cont by simp-all
have x0S: x0 ∈ S
  using x0-in-UV UVS by blast

have acUV: h (a, c) ∈ U ∩ V
  using leftUV cd by auto
have adUV: h (a, d) ∈ U ∩ V
  using leftUV cd by auto
have bcUV: h (b, c) ∈ U ∩ V
  using rightUV cd by auto
have bdUV: h (b, d) ∈ U ∩ V
  using rightUV cd by auto

have pc-path: path ?pc
proof -
  have continuous-on {0..1} ?pc
  proof (rule continuous-on-compose2[OF h-on])
    show continuous-on {0..1} (λt. (t, c))
      by (intro continuous-intros)
    show (λt. (t, c)) ' {0..1} ⊆ {0..1} × {0..1}
      using c01 by auto
  qed
  then show ?thesis
    by (simp add: path-def)
qed

have pd-path: path ?pd
proof -
  have continuous-on {0..1} ?pd
  proof (rule continuous-on-compose2[OF h-on])
    show continuous-on {0..1} (λt. (t, d))
      by (intro continuous-intros)
    show (λt. (t, d)) ' {0..1} ⊆ {0..1} × {0..1}
      using d01 by auto
  qed
  then show ?thesis
    by (simp add: path-def)
qed

have la-path: path ?la
proof -
  have continuous-on {0..1} ?la
  proof -
    have continuous-on {0..1} (λu. h (a, (d - c) * u + c))
    proof (rule continuous-on-compose2[OF h-on])
      show continuous-on {0..1} (λu. (a, (d - c) * u + c))
        by (intro continuous-intros)
      show (λu. (a, (d - c) * u + c)) ' {0..1} ⊆ {0..1} × {0..1}
    end
  end

```

```

proof
  fix  $x$ 
  assume  $x \in (\lambda u. (a, (d - c) * u + c)) \text{ ' } \{0..1\}$ 
  then obtain  $u$  where  $u01: u \in \{0..1\}$  and  $x\text{-eq}: x = (a, (d - c) * u +$ 
c)
    by blast
    have  $y\text{-in}: (d - c) * u + c \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF c01 d01 cd u01])
    show  $x \in \{0..1\} \times \{0..1\}$ 
    using  $a01\ y\text{-in}\ x\text{-eq}$  by auto
  qed
qed
then show ?thesis
  by (simp add: vertical-strip-path-def)
qed
then show ?thesis
  by (simp add: vertical-strip-path-def path-def)
qed
have  $lb\text{-path}: \text{path } ?lb$ 
proof -
  have continuous-on  $\{0..1\}$   $?lb$ 
  proof -
  have continuous-on  $\{0..1\}$   $(\lambda u. h (b, (d - c) * u + c))$ 
  proof (rule continuous-on-compose2[OF h-on])
  show continuous-on  $\{0..1\}$   $(\lambda u. (b, (d - c) * u + c))$ 
  by (intro continuous-intros)
  show  $(\lambda u. (b, (d - c) * u + c)) \text{ ' } \{0..1\} \subseteq \{0..1\} \times \{0..1\}$ 
  proof
  fix  $x$ 
  assume  $x \in (\lambda u. (b, (d - c) * u + c)) \text{ ' } \{0..1\}$ 
  then obtain  $u$  where  $u01: u \in \{0..1\}$  and  $x\text{-eq}: x = (b, (d - c) * u +$ 
c)
    by blast
    have  $y\text{-in}: (d - c) * u + c \in \{0..1\}$ 
    by (rule affine-unit-interval-member[OF c01 d01 cd u01])
    show  $x \in \{0..1\} \times \{0..1\}$ 
    using  $b01\ y\text{-in}\ x\text{-eq}$  by auto
  qed
qed
then show ?thesis
  by (simp add: vertical-strip-path-def)
qed
then show ?thesis
  by (simp add: vertical-strip-path-def path-def)
qed

have  $pc\text{-img}W: \text{path-image } ?pc \subseteq W$ 
  using h-intoW c01 by (auto simp: path-image-def)
have  $pd\text{-img}W: \text{path-image } ?pd \subseteq W$ 

```

```

    using h-intoW d01 by (auto simp: path-image-def)
  have bot-imgS: ?bot ' {0..1} ⊆ S
    using rectS ab cd by (auto simp: subpathin-image-eq)
  have top-imgS: ?top ' {0..1} ⊆ S
    using rectS ab cd by (auto simp: subpathin-image-eq)
  have la-imgS: path-image ?la ⊆ S
proof -
  have vs-subset: vertical-strip-path h a c d ' {0..1} ⊆ h ' ({a} × {c..d})
    by (rule vertical-strip-path-image-subset[OF cd])
  have path-image ?la ⊆ U ∩ V
    using vs-subset leftUV by (auto simp: path-image-def)
  then show ?thesis
    using UVS by blast
qed
  have lb-imgS: path-image ?lb ⊆ S
proof -
  have vs-subset: vertical-strip-path h b c d ' {0..1} ⊆ h ' ({b} × {c..d})
    by (rule vertical-strip-path-image-subset[OF cd])
  have path-image ?lb ⊆ U ∩ V
    using vs-subset rightUV by (auto simp: path-image-def)
  then show ?thesis
    using UVS by blast
qed
  have la-imgUV: path-image ?la ⊆ U ∩ V
proof -
  have vs-subset: vertical-strip-path h a c d ' {0..1} ⊆ h ' ({a} × {c..d})
    by (rule vertical-strip-path-image-subset[OF cd])
  show ?thesis
    using vs-subset leftUV by (auto simp: path-image-def)
qed
  have lb-imgUV: path-image ?lb ⊆ U ∩ V
proof -
  have vs-subset: vertical-strip-path h b c d ' {0..1} ⊆ h ' ({b} × {c..d})
    by (rule vertical-strip-path-image-subset[OF cd])
  show ?thesis
    using vs-subset rightUV by (auto simp: path-image-def)
qed

  have ac-path: path ?ac and ac-imgS: path-image ?ac ⊆ S
    using connector-path[OF acUV] connector-image-subset[OF acUV] UVS by
blast+
  have ad-path: path ?ad and ad-imgS: path-image ?ad ⊆ S
    using connector-path[OF adUV] connector-image-subset[OF adUV] UVS by
blast+
  have bc-path: path ?bc and bc-imgS: path-image ?bc ⊆ S
    using connector-path[OF bcUV] connector-image-subset[OF bcUV] UVS by
blast+
  have bd-path: path ?bd and bd-imgS: path-image ?bd ⊆ S
    using connector-path[OF bdUV] connector-image-subset[OF bdUV] UVS by

```

*blast+*

```

have segc-loop: segment-loop ?pc a b ∈ loop-space S x0
proof (rule segment-loop-in-set[where S = S])
  show path ?pc
    by (rule pc-path)
  show path-image ?pc ⊆ W
    by (rule pc-imgW)
  show a ∈ {0..1} b ∈ {0..1}
    by (rule a01, rule b01)
  show ?pc a ∈ U ∩ V ?pc b ∈ U ∩ V
    using acUV bcUV by simp-all
  show path-image (connector (?pc a)) ⊆ S
    using connector-image-subset[OF acUV] UVS by blast
  show path-image (connector (?pc b)) ⊆ S
    using connector-image-subset[OF bcUV] UVS by blast
  show ?bot ‘ {0..1} ⊆ S
    by (rule bot-imgS)
  show x0 ∈ S
    by (rule x0S)
qed
have segd-loop: segment-loop ?pd a b ∈ loop-space S x0
proof (rule segment-loop-in-set[where S = S])
  show path ?pd
    by (rule pd-path)
  show path-image ?pd ⊆ W
    by (rule pd-imgW)
  show a ∈ {0..1} b ∈ {0..1}
    by (rule a01, rule b01)
  show ?pd a ∈ U ∩ V ?pd b ∈ U ∩ V
    using adUV bdUV by simp-all
  show path-image (connector (?pd a)) ⊆ S
    using connector-image-subset[OF adUV] UVS by blast
  show path-image (connector (?pd b)) ⊆ S
    using connector-image-subset[OF bdUV] UVS by blast
  show ?top ‘ {0..1} ⊆ S
    by (rule top-imgS)
  show x0 ∈ S
    by (rule x0S)
qed
have bridge-a-loop: bridge-loop h a c d ∈ loop-space S x0
unfolding bridge-loop-eq-segment-loop
proof (rule segment-loop-in-set[where S = S])
  show path ?la
    by (rule la-path)
  show path-image ?la ⊆ W
    using la-imgUV by blast
  show (0::real) ∈ {0..1}
    by simp

```

```

show (1::real) ∈ {0..1}
  by simp
show ?la 0 ∈ U ∩ V ?la 1 ∈ U ∩ V
  using acUV adUV by (simp-all add: vertical-strip-path-def)
show path-image (connector (?la 0)) ⊆ S
  using connector-image-subset[OF acUV] UVS by (simp add: vertical-strip-path-def; blast)
show path-image (connector (?la 1)) ⊆ S
  using connector-image-subset[OF adUV] UVS by (simp add: vertical-strip-path-def; blast)
have edge-S: h ‘ ({a} × {c..d}) ⊆ S
  by (rule order-trans[OF leftUV UVS])
have la-imgS': vertical-strip-path h a c d ‘ {0..1} ⊆ S
  by (rule order-trans[OF vertical-strip-path-image-subset[OF cd] edge-S])
have la-eq: ?la = vertical-strip-path h a c d
  by simp
show subpathin 0 1 ?la ‘ {0..1} ⊆ S
  using la-imgS' by simp
show x0 ∈ S
  by (rule x0S)
qed
have bridge-b-loop: bridge-loop h b c d ∈ loop-space S x0
unfolding bridge-loop-eq-segment-loop
proof (rule segment-loop-in-set[where S = S])
  show path ?lb
    by (rule lb-path)
  show path-image ?lb ⊆ W
    using lb-imgUV by blast
  show (0::real) ∈ {0..1}
    by simp
  show (1::real) ∈ {0..1}
    by simp
  show ?lb 0 ∈ U ∩ V ?lb 1 ∈ U ∩ V
    using bcUV bdUV by (simp-all add: vertical-strip-path-def)
  show path-image (connector (?lb 0)) ⊆ S
    using connector-image-subset[OF bcUV] UVS by (simp add: vertical-strip-path-def; blast)
  show path-image (connector (?lb 1)) ⊆ S
    using connector-image-subset[OF bdUV] UVS by (simp add: vertical-strip-path-def; blast)
  have edge-S: h ‘ ({b} × {c..d}) ⊆ S
    by (rule order-trans[OF rightUV UVS])
  have lb-imgS': vertical-strip-path h b c d ‘ {0..1} ⊆ S
    by (rule order-trans[OF vertical-strip-path-image-subset[OF cd] edge-S])
  have lb-eq: ?lb = vertical-strip-path h b c d
    by simp
  show subpathin 0 1 ?lb ‘ {0..1} ⊆ S
    using lb-imgS' by simp
  show x0 ∈ S

```

by (rule  $x0S$ )  
**qed**

have  $segc\text{-}path$ : path (segment-loop ?pc a b) and  $segc\text{-}imgS$ : path-image  
(segment-loop ?pc a b)  $\subseteq S$   
**using**  $segc\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $segd\text{-}path$ : path (segment-loop ?pd a b) and  $segd\text{-}imgS$ : path-image  
(segment-loop ?pd a b)  $\subseteq S$   
**using**  $segd\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $bridge\text{-}a\text{-}path$ : path (bridge-loop h a c d) and  $bridge\text{-}a\text{-}imgS$ : path-image  
(bridge-loop h a c d)  $\subseteq S$   
**using**  $bridge\text{-}a\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $bridge\text{-}b\text{-}path$ : path (bridge-loop h b c d) and  $bridge\text{-}b\text{-}imgS$ : path-image  
(bridge-loop h b c d)  $\subseteq S$   
**using**  $bridge\text{-}b\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $segc\text{-}finish$ : pathfinish (segment-loop ?pc a b) =  $x0$   
**using**  $segc\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $bridge\text{-}b\text{-}start$ : pathstart (bridge-loop h b c d) =  $x0$   
**using**  $bridge\text{-}b\text{-}loop$  **unfolding**  $loop\text{-}space\text{-}def$  **by** *auto*  
have  $bc\text{-}start$ : pathstart ?bc =  $x0$   
**using**  $connector\text{-}start[OF\ bcUV]$  **by** *simp*  
have  $bc\text{-}finish$ : pathfinish ?bc =  $h(b, c)$   
**using**  $connector\text{-}finish[OF\ bcUV]$  **by** *simp*

have  $edge\text{-}hom$ :  
homotopic-paths  $S$  (?bot +++ ?lb) (?la +++ ?top)  
**unfolding**  $vertical\text{-}strip\text{-}path\text{-}def$   
**by** (rule  $rectangle\text{-}edge\text{-}homotopic\text{-}in\text{-}set[OF\ h\text{-}cont\ a01\ b01\ c01\ d01\ ab\ cd\ rectS]$ )

have  $lb\text{-}finish$ : pathfinish ?lb =  $h(b, d)$   
**using**  $cd$  **by** (*simp add: vertical-strip-path-def pathfinish-def*)  
have  $lb\text{-}start$ : pathstart ?lb =  $h(b, c)$   
**by** (*simp add: vertical-strip-path-def pathstart-def*)  
have  $rev\text{-}bd\text{-}path$ : path (reversepath ?bd)  
**using**  $bd\text{-}path$  **by** *simp*  
have  $rev\text{-}bd\text{-}imgS$ : path-image (reversepath ?bd)  $\subseteq S$   
**using**  $bd\text{-}imgS$  **by** *simp*  
have  $rev\text{-}bd\text{-}start$ : pathstart (reversepath ?bd) =  $h(b, d)$   
**using**  $connector\text{-}finish[OF\ bdUV]$  **by** *simp*  
have  $s\text{-}b\text{-}path$ : path (?lb +++ reversepath ?bd)  
**using**  $lb\text{-}path\ bd\text{-}path\ lb\text{-}finish\ rev\text{-}bd\text{-}start$  **by** *simp*  
have  $s\text{-}b\text{-}imgS$ : path-image (?lb +++ reversepath ?bd)  $\subseteq S$   
**by** (rule  $subset\text{-}path\text{-}image\text{-}join[OF\ lb\text{-}imgS]$ ) (*use*  $bd\text{-}imgS$  **in** *simp*)  
have  $ac\text{-}finish$ : pathfinish ?ac =  $h(a, c)$   
**using**  $connector\text{-}finish[OF\ acUV]$  **by** *simp*  
have  $bot\text{-}start$ : pathstart ?bot =  $h(a, c)$   
**by** (*simp add: pathstart-def subpathin-def*)  
have  $pc\text{-}pathin$ : pathin (top-of-set  $W$ ) ?pc  
**by** (rule  $path\text{-}top\text{-}of\text{-}setI[OF\ pc\text{-}path\ pc\text{-}imgW]$ )

```

have bot-path: path ?bot
proof -
  have pathin (top-of-set W) ?bot
    by (rule pathin-subpathin[OF pc-pathin]) (use a01 b01 in auto)
  then show ?thesis
    by (simp add: pathin-canon-iff)
qed
have r-b-path: path (?ac +++ ?bot)
  using ac-path bot-path ac-finish bot-start by simp
have bot-finish: pathfinish ?bot = h (b, c)
  by (simp add: pathfinish-def subpathin-def)
have r-b-finish: pathfinish (?ac +++ ?bot) = h (b, c)
  using r-b-path bot-finish by simp
have bot-path-imgS: path-image ?bot  $\subseteq$  S
  using bot-imgS by (simp add: path-image-def)
have pd-pathin: pathin (top-of-set W) ?pd
  by (rule path-top-of-setI[OF pd-path pd-imgW])
have r-b-imgS: path-image (?ac +++ ?bot)  $\subseteq$  S
proof (rule subset-path-image-join[OF ac-imgS])
  show path-image ?bot  $\subseteq$  S
    using bot-imgS by (simp add: path-image-def)
qed
have assoc-bridge-b:
  homotopic-paths S (bridge-loop h b c d) (?bc +++ (?lb +++ reversepath ?bd))
proof -
  have bc-lb: pathfinish ?bc = pathstart ?lb
    using bc-finish lb-start by simp
  have lb-bd: pathfinish ?lb = pathstart (reversepath ?bd)
    using lb-finish rev-bd-start by simp
  have rev-bd-path: path (reversepath ?bd)
    using bd-path by simp
  have rev-bd-imgS: path-image (reversepath ?bd)  $\subseteq$  S
    using bd-imgS by simp
  have homotopic-paths S (?bc +++ (?lb +++ reversepath ?bd)) (((?bc +++
?lb) +++ reversepath ?bd))
    by (rule homotopic-paths-assoc[OF bc-path bc-imgS lb-path lb-imgS rev-bd-path
rev-bd-imgS bc-lb lb-bd])
  then show ?thesis
    unfolding bridge-loop-def by (rule homotopic-paths-sym)
qed
have s-b-start: pathstart (?lb +++ reversepath ?bd) = h (b, c)
  using lb-start by simp
have lhs-step1:
  homotopic-paths S
    (segment-loop ?pc a b +++ bridge-loop h b c d)
    (segment-loop ?pc a b +++ (?bc +++ (?lb +++ reversepath ?bd)))
  by (rule homotopic-paths-join-left[OF assoc-bridge-b segc-path segc-imgS]) (use
segc-finish bridge-b-start in simp)
have lhs-step2:

```

```

    homotopic-paths S
      (segment-loop ?pc a b +++ (?bc +++ (?lb +++ reversepath ?bd)))
      (((segment-loop ?pc a b +++ ?bc) +++ (?lb +++ reversepath ?bd)))
    by (rule homotopic-paths-assoc[OF segc-path segc-imgS bc-path bc-imgS s-b-path
s-b-imgS])
      (use segc-finish bc-start bc-finish s-b-start in simp-all)
    have segc-bc-path: path (segment-loop ?pc a b +++ ?bc)
      using segc-path bc-path segc-finish bc-start by simp
    have segc-bc-imgS: path-image (segment-loop ?pc a b +++ ?bc)  $\subseteq$  S
      by (rule subset-path-image-join[OF segc-imgS]) (use bc-imgS in simp)
    have lhs-step3:
      homotopic-paths S
        (((segment-loop ?pc a b +++ ?bc) +++ (?lb +++ reversepath ?bd)))
        (((?ac +++ ?bot) +++ reversepath ?bc) +++ ?bc) +++ (?lb +++ re-
reversepath ?bd))
    proof (rule homotopic-paths-eq[OF - -])
      show path (((segment-loop ?pc a b +++ ?bc) +++ (?lb +++ reversepath ?bd)))
        using segc-bc-path s-b-path bc-finish s-b-start by simp
      show path-image (((segment-loop ?pc a b +++ ?bc) +++ (?lb +++ reversepath
?bd)))  $\subseteq$  S
        by (rule subset-path-image-join[OF segc-bc-imgS]) (use s-b-imgS in simp)
      show ((segment-loop ?pc a b +++ ?bc) +++ (?lb +++ reversepath ?bd)) t =
        (((?ac +++ ?bot) +++ reversepath ?bc) +++ ?bc) +++ (?lb +++
reversepath ?bd)) t
        if t  $\in$  {0..1} for t
        using that by (simp add: segment-loop-def)
    qed
    have lhs-step4:
      homotopic-paths S
        (((?ac +++ ?bot) +++ reversepath ?bc) +++ ?bc) +++ (?lb +++ re-
reversepath ?bd))
        ((?ac +++ ?bot) +++ (?lb +++ reversepath ?bd))
      by (rule homotopic-paths-cancel-middle-local[OF r-b-path r-b-imgS bc-path
bc-imgS s-b-path s-b-imgS])
        (use r-b-finish bc-finish s-b-start in simp-all)
    have assoc-ac-bot-lb:
      homotopic-paths S (((?ac +++ ?bot) +++ ?lb)) (?ac +++ (?bot +++ ?lb))
    proof -
      have homotopic-paths S (?ac +++ (?bot +++ ?lb)) (((?ac +++ ?bot) +++
?lb))
        by (rule homotopic-paths-assoc[OF ac-path ac-imgS bot-path bot-path-imgS
lb-path lb-imgS])
          (use ac-finish bot-start bot-finish lb-start in simp-all)
      then show ?thesis
        by (rule homotopic-paths-sym)
    qed
    have assoc-r-b-lb:
      homotopic-paths S
        ((?ac +++ ?bot) +++ (?lb +++ reversepath ?bd))

```

```

    ((((?ac +++ ?bot) +++ ?lb) +++ reversepath ?bd))
  by (rule homotopic-paths-assoc[OF r-b-path r-b-imgS lb-path lb-imgS rev-bd-path
rev-bd-imgS])
    (use r-b-finish lb-start lb-finish rev-bd-start in simp-all)
  have assoc-ac-bot-lb-join:
    homotopic-paths S
    ((((?ac +++ ?bot) +++ ?lb) +++ reversepath ?bd))
    ((?ac +++ (?bot +++ ?lb) +++ reversepath ?bd)
  by (rule homotopic-paths-join-right[OF assoc-ac-bot-lb rev-bd-path rev-bd-imgS])
    (use lb-finish rev-bd-start in simp-all)
  have lhs-step5:
    homotopic-paths S
    ((?ac +++ ?bot) +++ (?lb +++ reversepath ?bd))
    ((?ac +++ (?bot +++ ?lb) +++ reversepath ?bd)
  by (rule homotopic-paths-trans[OF assoc-r-b-lb assoc-ac-bot-lb-join])
  have lhs-to-boundary:
    homotopic-paths S
    (segment-loop ?pc a b +++ bridge-loop h b c d)
    ((?ac +++ (?bot +++ ?lb) +++ reversepath ?bd)
  by (rule homotopic-paths-trans[OF lhs-step1])
    (rule homotopic-paths-trans[OF lhs-step2],
    rule homotopic-paths-trans[OF lhs-step3],
    rule homotopic-paths-trans[OF lhs-step4 lhs-step5])

  have top-path: path ?top
  proof -
    have pathin (top-of-set W) ?top
      by (rule pathin-subpathin[OF pd-pathin]) (use a01 b01 in auto)
    then show ?thesis
      by (simp add: pathin-canon-iff)
  qed
  have top-start: pathstart ?top = h (a, d)
    by (simp add: pathstart-def subpathin-def)
  have top-finish: pathfinish ?top = h (b, d)
    by (simp add: pathfinish-def subpathin-def)
  have top-path-imgS: path-image ?top  $\subseteq$  S
    using top-imgS by (simp add: path-image-def)
  have s-a-path: path (?top +++ reversepath ?bd)
    using top-path bd-path top-finish rev-bd-start by simp
  have s-a-imgS: path-image (?top +++ reversepath ?bd)  $\subseteq$  S
    by (rule subset-path-image-join[OF top-path-imgS rev-bd-imgS])
  have la-start: pathstart ?la = h (a, c)
    by (simp add: vertical-strip-path-def pathstart-def)
  have la-finish: pathfinish ?la = h (a, d)
    by (simp add: vertical-strip-path-def pathfinish-def)
  have ad-start: pathstart ?ad = x0
    using connector-start[OF adUV] by simp
  have ad-finish: pathfinish ?ad = h (a, d)
    using connector-finish[OF adUV] by simp

```

```

have rev-ad-path: path (reversepath ?ad)
  using ad-path by simp
have rev-ad-imgS: path-image (reversepath ?ad)  $\subseteq S$ 
  using ad-imgS by simp
have rev-ad-start: pathstart (reversepath ?ad) = h (a, d)
  using connector-finish[OF adUV] by simp
have rev-ad-finish: pathfinish (reversepath ?ad) = x0
  using ad-start by simp
have r-a-path: path (?ac +++ ?la)
  using ac-path la-path ac-finish la-start by simp
have r-a-imgS: path-image (?ac +++ ?la)  $\subseteq S$ 
  by (rule subset-path-image-join[OF ac-imgS la-imgS])
have r-a-rev-path: path ((?ac +++ ?la) +++ reversepath ?ad)
  using r-a-path rev-ad-path la-finish rev-ad-start by simp
have r-a-rev-imgS: path-image ((?ac +++ ?la) +++ reversepath ?ad)  $\subseteq S$ 
  by (rule subset-path-image-join[OF r-a-imgS rev-ad-imgS])
have assoc-bridge-a:
  homotopic-paths S (bridge-loop h a c d) (?ac +++ (?la +++ reversepath ?ad))
proof -
  have homotopic-paths S (?ac +++ (?la +++ reversepath ?ad)) (((?ac +++
  ?la) +++ reversepath ?ad))
  by (rule homotopic-paths-assoc[OF ac-path ac-imgS la-path la-imgS rev-ad-path
  rev-ad-imgS])
  (use ac-finish la-start la-finish rev-ad-start in simp-all)
  then show ?thesis
  unfolding bridge-loop-def by (rule homotopic-paths-sym)
qed
have bridge-a-finish: pathfinish (bridge-loop h a c d) = x0
  using bridge-a-loop unfolding loop-space-def by auto
have segd-start: pathstart (segment-loop ?pd a b) = x0
  using segd-loop unfolding loop-space-def by auto
have rhs-step1:
  homotopic-paths S
  (bridge-loop h a c d +++ segment-loop ?pd a b)
  ((?ac +++ (?la +++ reversepath ?ad)) +++ segment-loop ?pd a b)
  by (rule homotopic-paths-join-right[OF assoc-bridge-a segd-path segd-imgS])
  (use bridge-a-finish segd-start in simp-all)
have rhs-step2:
  homotopic-paths S
  ((?ac +++ (?la +++ reversepath ?ad)) +++ segment-loop ?pd a b)
  (((?ac +++ ?la) +++ reversepath ?ad) +++ segment-loop ?pd a b)
  by (rule homotopic-paths-join-right[
  OF homotopic-paths-assoc[OF ac-path ac-imgS la-path la-imgS rev-ad-path
  rev-ad-imgS] segd-path segd-imgS])
  (use ac-finish la-start la-finish rev-ad-start ad-start bridge-a-finish segd-start
  in simp-all)
have rhs-step3:
  homotopic-paths S
  (((?ac +++ ?la) +++ reversepath ?ad) +++ segment-loop ?pd a b)

```

$((((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } ?ad) \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
**proof** –  
**have** *segd-assoc*:  
 $\text{homotopic-paths } S (\text{segment-loop } ?pd \text{ a b}) (?ad \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
**proof** –  
**have** *homotopic-paths*  $S (?ad \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd)) (\text{segment-loop } ?pd \text{ a b})$   
**unfolding** *segment-loop-def*  
**by** (*rule homotopic-paths-assoc*[*OF ad-path ad-imgS top-path top-path-imgS rev-bd-path rev-bd-imgS*])  
*(use ad-finish top-start top-finish rev-bd-start in simp-all)*  
**then show** *?thesis*  
**by** (*rule homotopic-paths-sym*)  
**qed**  
**have** *step1*:  
 $\text{homotopic-paths } S$   
 $(((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } \text{segment-loop } ?pd \text{ a b})$   
 $(((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } (?ad \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd)))$   
**by** (*rule homotopic-paths-join-left*[*OF segd-assoc r-a-rev-path r-a-rev-imgS*])  
*(use segd-start ad-start rev-ad-finish in simp-all)*  
**have** *step2*:  
 $\text{homotopic-paths } S$   
 $(((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } (?ad \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd)))$   
 $((((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } ?ad) \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
**by** (*rule homotopic-paths-assoc*[*OF r-a-rev-path r-a-rev-imgS ad-path ad-imgS s-a-path s-a-imgS*])  
*(use rev-ad-finish ad-start ad-finish top-start in simp-all)*  
**show** *?thesis*  
**by** (*rule homotopic-paths-trans*[*OF step1 step2*])  
**qed**  
**have** *rhs-step4*:  
 $\text{homotopic-paths } S$   
 $((((?ac \text{ +++ } ?la) \text{ +++ } \text{reversepath } ?ad) \text{ +++ } ?ad) \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
 $((?ac \text{ +++ } ?la) \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
**by** (*rule homotopic-paths-cancel-middle-local*[*OF r-a-path r-a-imgS ad-path ad-imgS s-a-path s-a-imgS*])  
*(use la-finish ad-finish top-start in simp-all)*  
**have** *rhs-step5*:  
 $\text{homotopic-paths } S$   
 $((?ac \text{ +++ } ?la) \text{ +++ } (?top \text{ +++ } \text{reversepath } ?bd))$   
 $((?ac \text{ +++ } (?la \text{ +++ } ?top)) \text{ +++ } \text{reversepath } ?bd)$   
**proof** –  
**have** *step5a*:

```

    homotopic-paths S
      ((?ac +++ ?la) +++ (?top +++ reversepath ?bd))
      (((?ac +++ ?la) +++ ?top) +++ reversepath ?bd)
    by (rule homotopic-paths-assoc[OF r-a-path r-a-imgS top-path top-path-imgS
rev-bd-path rev-bd-imgS])
      (use la-finish top-start top-finish rev-bd-start in simp-all)
  have inner:
    homotopic-paths S
      (((?ac +++ ?la) +++ ?top))
      (?ac +++ (?la +++ ?top))
  proof -
    have homotopic-paths S
      (?ac +++ (?la +++ ?top))
      (((?ac +++ ?la) +++ ?top))
    by (rule homotopic-paths-assoc[OF ac-path ac-imgS la-path la-imgS top-path
top-path-imgS])
      (use ac-finish la-start la-finish top-start in simp-all)
    then show ?thesis
      by (rule homotopic-paths-sym)
  qed
  have step5b:
    homotopic-paths S
      (((?ac +++ ?la) +++ ?top) +++ reversepath ?bd)
      ((?ac +++ (?la +++ ?top)) +++ reversepath ?bd)
    by (rule homotopic-paths-join-right[OF inner rev-bd-path rev-bd-imgS])
      (use top-finish rev-bd-start in simp-all)
  show ?thesis
    by (rule homotopic-paths-trans[OF step5a step5b])
  qed
  have boundary-to-rhs:
    homotopic-paths S
      ((?ac +++ (?la +++ ?top)) +++ reversepath ?bd)
      (bridge-loop h a c d +++ segment-loop ?pd a b)
  proof -
    have homotopic-paths S (bridge-loop h a c d +++ segment-loop ?pd a b)
      ((?ac +++ (?la +++ ?top)) +++ reversepath ?bd)
    by (rule homotopic-paths-trans[OF rhs-step1])
      (rule homotopic-paths-trans[OF rhs-step2],
      rule homotopic-paths-trans[OF rhs-step3],
      rule homotopic-paths-trans[OF rhs-step4 rhs-step5])
    then show ?thesis
      by (rule homotopic-paths-sym)
  qed
  have edge-boundary:
    homotopic-paths S
      ((?ac +++ (?bot +++ ?lb)) +++ reversepath ?bd)
      ((?ac +++ (?la +++ ?top)) +++ reversepath ?bd)
  proof -

```

```

have pre:
  homotopic-paths S (?ac +++ (?bot +++ ?lb)) (?ac +++ (?la +++ ?top))
  by (rule homotopic-paths-join-left[OF edge-hom ac-path ac-imgS]) (use
ac-finish bot-start la-start in simp-all)
  show ?thesis
proof (rule homotopic-paths-join-right[OF pre rev-bd-path rev-bd-imgS])
  show pathfinish (?ac +++ (?bot +++ ?lb)) = pathstart (reversepath ?bd)
  using lb-finish rev-bd-start by simp
qed
qed

show ?thesis
by (rule homotopic-paths-trans[OF lhs-to-boundary])
  (rule homotopic-paths-trans[OF edge-boundary boundary-to-rhs])
qed

lemma horizontal-rectangle-segment-loop-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 ab: a ≤ b
  and segS: h ‘ ({a..b} × {c}) ⊆ S
  and acUV: h (a, c) ∈ U ∩ V
  and bcUV: h (b, c) ∈ U ∩ V
  and UVS: U ∩ V ⊆ S
shows segment-loop (λt. h (t, c)) a b ∈ loop-space S x0
proof –
have h-on: continuous-on ({0..1} × {0..1}) h
  and h-intoW: h ∈ ({0..1} × {0..1}) → W
  using h-cont by simp-all
have pc-path: path (λt. h (t, c))
proof –
have continuous-on {0..1} (λt. h (t, c))
proof (rule continuous-on-compose2[OF h-on])
  show continuous-on {0..1} (λt. (t, c))
  by (intro continuous-intros)
  show (λt. (t, c)) ‘ {0..1} ⊆ {0..1} × {0..1}
  using c01 by auto
qed
then show ?thesis
  by (simp add: path-def)
qed
have pc-imgW: path-image (λt. h (t, c)) ⊆ W
  using h-intoW c01 by (auto simp: path-image-def)
have seg-imgS: subpathin a b (λt. h (t, c)) ‘ {0..1} ⊆ S
  using segS ab by (auto simp: subpathin-image-eq)
have x0S: x0 ∈ S
  using x0-in-UV UVS by blast

```

```

show ?thesis
proof (rule segment-loop-in-set[where  $S = S$ ])
  show path  $(\lambda t. h(t, c))$ 
    by (rule pc-path)
  show path-image  $(\lambda t. h(t, c)) \subseteq W$ 
    by (rule pc-imgW)
  show  $a \in \{0..1\} \ b \in \{0..1\}$ 
    by (rule a01, rule b01)
  show  $(\lambda t. h(t, c)) \ a \in U \cap V \ (\lambda t. h(t, c)) \ b \in U \cap V$ 
    using acUV bcUV by simp-all
  show path-image (connector  $((\lambda t. h(t, c)) \ a)) \subseteq S$ 
    using connector-image-subset[OF acUV] UVS by blast
  show path-image (connector  $((\lambda t. h(t, c)) \ b)) \subseteq S$ 
    using connector-image-subset[OF bcUV] UVS by blast
  show subpathin a b  $(\lambda t. h(t, c)) \ \{0..1\} \subseteq S$ 
    by (rule seg-imgS)
  show  $x0 \in S$ 
    by (rule x0S)
qed
qed

lemma vertical-bridge-loop-in-set:
  fixes  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$ 
  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:  $h \ \{t\} \times \{c..d\} \subseteq U \cap V$ 
    and UVS:  $U \cap V \subseteq S$ 
  shows bridge-loop  $h \ t \ c \ d \in \text{loop-space } S \ x0$ 
proof -
  have h-on: continuous-on  $(\{0..1\} \times \{0..1\}) \ h$ 
    and h-intoW:  $h \in (\{0..1\} \times \{0..1\}) \rightarrow W$ 
    using h-cont by simp-all
  have vt-path: path (vertical-strip-path  $h \ t \ c \ d$ )
proof -
  have continuous-on  $\{0..1\}$  (vertical-strip-path  $h \ t \ c \ d$ )
proof -
  have continuous-on  $\{0..1\}$   $(\lambda u. h(t, (d - c) * u + c))$ 
proof (rule continuous-on-compose2[OF h-on])
  show continuous-on  $\{0..1\}$   $(\lambda u. (t, (d - c) * u + c))$ 
    by (intro continuous-intros)
  show  $(\lambda u. (t, (d - c) * u + c)) \ \{0..1\} \subseteq \{0..1\} \times \{0..1\}$ 
proof
  fix  $x$ 
  assume  $x \in (\lambda u. (t, (d - c) * u + c)) \ \{0..1\}$ 
  then obtain  $u$  where u01:  $u \in \{0..1\}$  and x-eq:  $x = (t, (d - c) * u +$ 
c)
    by blast

```

```

    have y-in:  $(d - c) * u + c \in \{0..1\}$ 
      by (rule affine-unit-interval-member[OF c01 d01 cd u01])
    show  $x \in \{0..1\} \times \{0..1\}$ 
      using t01 y-in x-eq by auto
  qed
  qed
  then show ?thesis
    by (simp add: vertical-strip-path-def)
  qed
  then show ?thesis
    by (simp add: vertical-strip-path-def path-def)
  qed
  have vt-imgW:  $\text{path-image } (\text{vertical-strip-path } h \ t \ c \ d) \subseteq W$ 
  proof
    fix x
    assume x-in:  $x \in \text{path-image } (\text{vertical-strip-path } h \ t \ c \ d)$ 
    then obtain u where u01:  $u \in \{0..1\}$  and x-eq:  $x = \text{vertical-strip-path } h \ t \ c \ d \ u$ 
    unfolding path-image-def by blast
    have yu01:  $(d - c) * u + c \in \{0..1\}$ 
      by (rule affine-unit-interval-member[OF c01 d01 cd u01])
    have point-in:  $(t, (d - c) * u + c) \in \{0..1\} \times \{0..1\}$ 
      using t01 yu01 by auto
    show  $x \in W$ 
      using h-intoW point-in x-eq by (auto simp: vertical-strip-path-def)
  qed
  have vt-imgS:  $\text{vertical-strip-path } h \ t \ c \ d \ ' \{0..1\} \subseteq S$ 
  proof -
    have vs-subset:  $\text{vertical-strip-path } h \ t \ c \ d \ ' \{0..1\} \subseteq h \ ' (\{t\} \times \{c..d\})$ 
      by (rule vertical-strip-path-image-subset[OF cd])
    show ?thesis
      using vs-subset edgeUV UVS by blast
  qed
  have tcUV:  $h \ (t, c) \in U \cap V$  and tdUV:  $h \ (t, d) \in U \cap V$ 
    using edgeUV cd by auto
  have x0S:  $x0 \in S$ 
    using x0-in-UV UVS by blast
  have segment-loop  $(\text{vertical-strip-path } h \ t \ c \ d) \ 0 \ 1 \in \text{loop-space } S \ x0$ 
  proof (rule segment-loop-in-set[where S = S])
    show path  $(\text{vertical-strip-path } h \ t \ c \ d)$ 
      by (rule vt-path)
    show path-image  $(\text{vertical-strip-path } h \ t \ c \ d) \subseteq W$ 
      by (rule vt-imgW)
    show  $0 \in \{0::\text{real}..1\}$ 
      by auto
    show  $1 \in \{0::\text{real}..1\}$ 
      by auto
    show  $\text{vertical-strip-path } h \ t \ c \ d \ 0 \ 1 \in U \cap V$ 
      using tcUV by (simp add: vertical-strip-path-def)
  qed

```

```

show vertical-strip-path h t c d 1 ∈ U ∩ V
  using tdUV by (simp add: vertical-strip-path-def)
show path-image (connector (vertical-strip-path h t c d 0)) ⊆ S
  using connector-image-subset[OF tcUV] UVS by (simp add: vertical-strip-path-def; blast)
show path-image (connector (vertical-strip-path h t c d 1)) ⊆ S
  using connector-image-subset[OF tdUV] UVS by (simp add: vertical-strip-path-def; blast)
show subpathin 0 1 (vertical-strip-path h t c d) ‘ {0..1} ⊆ S
  using vt-imgS by (simp add: subpathin-def)
show x0 ∈ S
  by (rule x0S)
qed
then show ?thesis
  by simp
qed

```

**lemma** rectangle-segment-loop-bridge-class-eq:

```

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
  and leftUV: h ‘ ({a} × {c..d}) ⊆ U ∩ V
  and rightUV: h ‘ ({b} × {c..d}) ⊆ U ∩ V
  and UVS: U ∩ V ⊆ S

```

```

shows fundamental-group-mult S x0
  (loop-class S x0 (segment-loop (λt. 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 (λt. h (t, d)) a b))

```

**proof** –

```

have acUV: h (a, c) ∈ U ∩ V
  using leftUV cd by auto
have bcUV: h (b, c) ∈ U ∩ V
  using rightUV cd by auto
have adUV: h (a, d) ∈ U ∩ V
  using leftUV cd by auto
have bdUV: h (b, d) ∈ U ∩ V
  using rightUV cd by auto

```

```

have segc-in-S: h ‘ ({a..b} × {c}) ⊆ S

```

**proof**

```

  fix x
  assume x-in: x ∈ h ‘ ({a..b} × {c})
  then obtain aa where aa-in: aa ∈ {a..b} and x-eq: x = h (aa, c)

```

```

    by auto
  have (aa, c) ∈ {a..b} × {c..d}
    using aa-in cd by auto
  then show x ∈ S
    using rectS x-eq by blast
qed
have segd-in-S: h ‘ ({a..b} × {d}) ⊆ S
proof
  fix x
  assume x-in: x ∈ h ‘ ({a..b} × {d})
  then obtain aa where aa-in: aa ∈ {a..b} and x-eq: x = h (aa, d)
    by auto
  have (aa, d) ∈ {a..b} × {c..d}
    using aa-in cd by auto
  then show x ∈ S
    using rectS x-eq by blast
qed
have segc-loop: segment-loop (λt. h (t, c)) a b ∈ loop-space S x0
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont a01 b01 c01 ab])
(use segc-in-S acUV bcUV UVS in auto)
have segd-loop: segment-loop (λt. h (t, d)) a b ∈ loop-space S x0
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont a01 b01 d01 ab])
(use segd-in-S adUV bdUV UVS in auto)
have bridge-b-loop: bridge-loop h b c d ∈ loop-space S x0
  by (rule vertical-bridge-loop-in-set[OF h-cont b01 c01 d01 cd]) (use rightUV
UVS in auto)
have bridge-a-loop: bridge-loop h a c d ∈ loop-space S x0
  by (rule vertical-bridge-loop-in-set[OF h-cont a01 c01 d01 cd]) (use leftUV UVS
in auto)

have left-join-loop:
  segment-loop (λt. h (t, c)) a b +++ bridge-loop h b c d ∈ loop-space S x0
  by (rule loop-space-join[OF segc-loop bridge-b-loop])
have right-join-loop:
  bridge-loop h a c d +++ segment-loop (λt. h (t, d)) a b ∈ loop-space S x0
  by (rule loop-space-join[OF bridge-a-loop segd-loop])

have segc-in: loop-class S x0 (segment-loop (λt. h (t, c)) a b) ∈ fundamen-
tal-group-space S x0
  by (rule loop-class-in-space[OF segc-loop])
have segd-in: loop-class S x0 (segment-loop (λt. h (t, d)) a b) ∈ fundamen-
tal-group-space S x0
  by (rule loop-class-in-space[OF segd-loop])
have bridge-a-in: loop-class S x0 (bridge-loop h a c d) ∈ fundamental-group-space
S x0
  by (rule loop-class-in-space[OF bridge-a-loop])
have bridge-b-in: loop-class S x0 (bridge-loop h b c d) ∈ fundamental-group-space
S x0
  by (rule loop-class-in-space[OF bridge-b-loop])

```

**have** *left-mult*:  
*fundamental-group-mult*  $S\ x0$   
(loop-class  $S\ x0$  (segment-loop  $(\lambda t. h\ (t, c))\ a\ b$ )  
(loop-class  $S\ x0$  (bridge-loop  $h\ b\ c\ d$ )) =  
loop-class  $S\ x0$  (segment-loop  $(\lambda t. h\ (t, c))\ a\ b\ +++$  bridge-loop  $h\ b\ c\ d$ )  
**by** (rule *fundamental-group-mult-eqI*[*OF segc-in bridge-b-in segc-loop*  
*bridge-b-loop*]) *simp-all*  
**have** *right-mult*:  
*fundamental-group-mult*  $S\ x0$   
(loop-class  $S\ x0$  (bridge-loop  $h\ a\ c\ d$ )  
(loop-class  $S\ x0$  (segment-loop  $(\lambda t. h\ (t, d))\ a\ b$ )) =  
loop-class  $S\ x0$  (bridge-loop  $h\ a\ c\ d\ +++$  segment-loop  $(\lambda t. h\ (t, d))\ a\ b$ )  
**by** (rule *fundamental-group-mult-eqI*[*OF bridge-a-in segd-in bridge-a-loop*  
*segd-loop*]) *simp-all*  
  
**have** *joined-hom*:  
*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  $(\lambda t. h\ (t, d))\ a\ b$ )  
**by** (rule *rectangle-segment-loop-bridge-homotopic*[*OF h-cont a01 b01 c01 d01 ab*  
*cd rectS leftUV rightUV UVS*])  
**have** *joined-eq*:  
loop-class  $S\ x0$  (segment-loop  $(\lambda t. h\ (t, c))\ a\ b\ +++$  bridge-loop  $h\ b\ c\ d$ ) =  
loop-class  $S\ x0$  (bridge-loop  $h\ a\ c\ d\ +++$  segment-loop  $(\lambda t. h\ (t, d))\ a\ b$ )  
**by** (rule *loop-class-eqI*[*OF left-join-loop right-join-loop joined-hom*])  
  
**show** *?thesis*  
**using** *left-mult right-mult joined-eq by simp*  
**qed**  
  
**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** –  
**have** *step*:  
*carrier-amalgam-equiv*  $H\ i1\ i2$  (bridge-word *True*  $h\ rest$ ) (bridge-word *False*  $h$   
 $rest$ )  
**using** *h-in*  
**by** (auto *simp: bridge-word.simps intro: carrier-amalgam-equiv.step car-*  
*rier-amalgam-step.identify*)  
**show** *carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
(bridge-word *True*  $h\ rest$ ) (bridge-word *False*  $h\ rest$ )  
**by** (rule *carrier-full-amalg-equiv.of-amalg*[*OF step*])  
**then show** *carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
(bridge-word *False*  $h\ rest$ ) (bridge-word *True*  $h\ rest$ )

by (rule carrier-full-amalg-equiv.sym)  
qed

**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*  $h\ c\ d\ (u\ \#\ ts)\ bs$   
using *assms* by *simp-all*

**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** (*cases b*)  
case *True*  
have *step*: *rectangle-partition*  $h\ c\ d\ (u\ \#\ v\ \#\ ts)\ (e\ \#\ bs)$   
by (rule *rectangle-partition-step-props*(5)[*OF rp*])  
have *tu*:  $t < u$  and *uv*:  $u < v$   
using *rp step* by *simp-all*  
have *leftU*:  $h\ '\ (\{t..u\} \times \{c..d\}) \subseteq U$   
using *rectangle-partition-step-props*(4)[*OF rp*] *True* by *simp*  
from *diff True* have *e-false*:  $\neg e$   
by *simp*  
then have *rightV*:  $h\ '\ (\{u..v\} \times \{c..d\}) \subseteq V$   
using *rectangle-partition-step-props*(4)[*OF step*] by *simp*  
show ?*thesis*  
**proof**  
fix  $x$   
assume *x-in*:  $x \in h\ '\ (\{u\} \times \{c..d\})$   
then obtain  $y$  where *y-cd*:  $y \in \{c..d\}$  and *x-eq*:  $x = h\ (u, y)$   
by *auto*  
have *u-tu*:  $u \in \{t..u\}$  and *u-uv*:  $u \in \{u..v\}$   
using *tu uv* by *auto*  
have *xU*:  $x \in U$   
using *leftU x-eq u-tu y-cd* by *auto*  
have *xV*:  $x \in V$   
using *rightV x-eq u-uv y-cd* by *auto*  
show  $x \in U \cap V$   
using *xU xV* by *blast*  
qed  
next  
case *False*  
have *step*: *rectangle-partition*  $h\ c\ d\ (u\ \#\ v\ \#\ ts)\ (e\ \#\ bs)$   
by (rule *rectangle-partition-step-props*(5)[*OF rp*])  
have *tu*:  $t < u$  and *uv*:  $u < v$   
using *rp step* by *simp-all*

```

have leftV: h ' ( $\{t..u\} \times \{c..d\}$ )  $\subseteq V$ 
  using rectangle-partition-step-props(4)[OF rp] False by simp
from diff False have e-true: e
  by simp
then have rightU: h ' ( $\{u..v\} \times \{c..d\}$ )  $\subseteq U$ 
  using rectangle-partition-step-props(4)[OF step] by simp
show ?thesis
proof
  fix x
  assume x-in:  $x \in h ' (\{u\} \times \{c..d\})$ 
  then obtain y where y-cd:  $y \in \{c..d\}$  and x-eq:  $x = h (u, y)$ 
  by auto
  have u-tu:  $u \in \{t..u\}$  and u-uv:  $u \in \{u..v\}$ 
  using tu uv by auto
  have xU:  $x \in U$ 
  using rightU x-eq u-uv y-cd by auto
  have xV:  $x \in V$ 
  using leftV x-eq u-tu y-cd by auto
  show  $x \in U \cap V$ 
  using xU xV by blast
qed
qed

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)
  using rel a-in
  by (cases b) (auto intro: carrier-full-amalg-equiv-left-context carrier-full-amalg-equiv-right-context)

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)
  using h-in bc bridge-word-identify
  by (cases b; cases c) auto

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  $\cap$  V) x0 (bridge-loop h b c d)) rest))
  (bridge-word True (loop-class (U  $\cap$  V) x0 (bridge-loop h a c d))
    (WordLeft (loop-class U x0 (segment-loop ( $\lambda t. h (t, d)$ ) a b)) rest))
proof –
define bottom where
  bottom = loop-class U x0 (segment-loop ( $\lambda t. h (t, c)$ ) a b)
define top where
  top = loop-class U x0 (segment-loop ( $\lambda t. h (t, d)$ ) a b)
define ga where
  ga = loop-class (U  $\cap$  V) x0 (bridge-loop h a c d)
define gb where
  gb = loop-class (U  $\cap$  V) x0 (bridge-loop h b c d)

have acUV: h (a, c)  $\in U \cap V$ 
using leftUV cd by auto
have bcUV: h (b, c)  $\in U \cap V$ 
using rightUV cd by auto
have adUV: h (a, d)  $\in U \cap V$ 
using leftUV cd by auto
have bdUV: h (b, d)  $\in U \cap V$ 
using rightUV cd by auto

have segc-in-U: h ‘ ( $\{a..b\} \times \{c\}$ )  $\subseteq U$ 
proof
  fix x
  assume x-in: x  $\in h$  ‘ ( $\{a..b\} \times \{c\}$ )
  then obtain aa where aa-in: aa  $\in \{a..b\}$  and x-eq: x = h (aa, c)
    by auto
  have (aa, c)  $\in \{a..b\} \times \{c..d\}$ 
    using aa-in cd by auto
  then show x  $\in U$ 
    using rectU x-eq by blast
qed
have segd-in-U: h ‘ ( $\{a..b\} \times \{d\}$ )  $\subseteq U$ 
proof
  fix x
  assume x-in: x  $\in h$  ‘ ( $\{a..b\} \times \{d\}$ )
  then obtain aa where aa-in: aa  $\in \{a..b\}$  and x-eq: x = h (aa, d)
    by auto
  have (aa, d)  $\in \{a..b\} \times \{c..d\}$ 
    using aa-in cd by auto
  then show x  $\in U$ 
    using rectU x-eq by blast
qed

```

```

have segc-loop: segment-loop ( $\lambda t. h (t, c)$ )  $a b \in \text{loop-space } U x0$ 
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont a01 b01 c01 ab])
    (use segc-in-U acUV bcUV in auto)
have segd-loop: segment-loop ( $\lambda t. h (t, d)$ )  $a b \in \text{loop-space } U x0$ 
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont a01 b01 d01 ab])
    (use segd-in-U adUV bdUV in auto)
have bridge-a-loop: bridge-loop  $h a c d \in \text{loop-space } (U \cap V) x0$ 
  by (rule vertical-bridge-loop-in-set[OF h-cont a01 c01 d01 cd leftUV]) simp
have bridge-b-loop: bridge-loop  $h b c d \in \text{loop-space } (U \cap V) x0$ 
  by (rule vertical-bridge-loop-in-set[OF h-cont b01 c01 d01 cd rightUV]) simp

have bottom-in: bottom  $\in G1$ 
  unfolding bottom-def by (rule loop-class-in-space[OF segc-loop])
have top-in: top  $\in G1$ 
  unfolding top-def by (rule loop-class-in-space[OF segd-loop])
have ga-in-H:  $ga \in H$ 
  unfolding ga-def by (rule loop-class-in-space[OF bridge-a-loop])
have gb-in-H:  $gb \in H$ 
  unfolding gb-def by (rule loop-class-in-space[OF bridge-b-loop])
have ga-in:  $i1 ga \in G1$ 
  by (rule i1-in-G1[OF ga-in-H])
have gb-in:  $i1 gb \in G1$ 
  by (rule i1-in-G1[OF gb-in-H])
have ab-in:  $\text{mult1 bottom } (i1 gb) \in G1$ 
  by (rule fundamental-group-mult-in-space[OF bottom-in gb-in])
have cd-in:  $\text{mult1 } (i1 ga) \text{ top} \in G1$ 
  by (rule fundamental-group-mult-in-space[OF ga-in top-in])

have ga-eq:  $i1 ga = \text{loop-class } U x0 \text{ (bridge-loop } h a c d)$ 
  unfolding ga-def
  by (subst i1-loop-class-eq[OF bridge-a-loop]) simp
have gb-eq:  $i1 gb = \text{loop-class } U x0 \text{ (bridge-loop } h b c d)$ 
  unfolding gb-def
  by (subst i1-loop-class-eq[OF bridge-b-loop]) simp
have mult-eq:  $\text{mult1 bottom } (i1 gb) = \text{mult1 } (i1 ga) \text{ top}$ 
  unfolding bottom-def top-def ga-eq gb-eq
  by (rule rectangle-segment-loop-bridge-class-eq[OF h-cont a01 b01 c01 d01 ab
    cd rectU leftUV rightUV])
    auto
have pair-eq:
  carrier-full-amalg-equiv  $G1 G2 H i1 i2 \text{mult1 one1 mult2 one2}$ 
  (WordLeft bottom (WordLeft (i1 gb) rest))
  (WordLeft (i1 ga) (WordLeft top rest))
  by (rule carrier-full-amalg-equiv-left-pair-eq[OF bottom-in gb-in ab-in ga-in
    top-in cd-in rest-in mult-eq])

show ?thesis
using pair-eq
by (simp only: bottom-def top-def bridge-word.simps ga-def gb-def)

```

qed

**lemma** *rectangle-segment-bridge-right-equiv*:

**fixes**  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$

**assumes**  $h\text{-cont}$ : *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**  $rectV$ :  $h \text{ ' } (\{a..b\} \times \{c..d\}) \subseteq V$

**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**  $rest\text{-in}$ : *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* ( $U \cap V$ )  $x0$  (*bridge-loop*  $h\ b\ c\ d$ )  $rest$ ))

(*bridge-word*  $False$  (*loop-class* ( $U \cap V$ )  $x0$  (*bridge-loop*  $h\ a\ c\ d$ ))

(*WordRight* (*loop-class*  $V\ x0$  (*segment-loop* ( $\lambda t. h\ (t, d)$ )  $a\ b$ )  $rest$ ))

**proof** –

**define** *bottom* **where**

$bottom = \text{loop-class } V\ x0\ (\text{segment-loop } (\lambda t. h\ (t, c))\ a\ b)$

**define** *top* **where**

$top = \text{loop-class } V\ x0\ (\text{segment-loop } (\lambda t. h\ (t, d))\ a\ b)$

**define** *ga* **where**

$ga = \text{loop-class } (U \cap V)\ x0\ (\text{bridge-loop } h\ a\ c\ d)$

**define** *gb* **where**

$gb = \text{loop-class } (U \cap V)\ x0\ (\text{bridge-loop } h\ b\ c\ d)$

**have**  $acUV$ :  $h\ (a, c) \in U \cap V$

**using**  $leftUV\ cd$  **by** *auto*

**have**  $bcUV$ :  $h\ (b, c) \in U \cap V$

**using**  $rightUV\ cd$  **by** *auto*

**have**  $adUV$ :  $h\ (a, d) \in U \cap V$

**using**  $leftUV\ cd$  **by** *auto*

**have**  $bdUV$ :  $h\ (b, d) \in U \cap V$

**using**  $rightUV\ cd$  **by** *auto*

**have**  $segc\text{-in-}V$ :  $h \text{ ' } (\{a..b\} \times \{c\}) \subseteq V$

**proof**

**fix**  $x$

**assume**  $x\text{-in}$ :  $x \in h \text{ ' } (\{a..b\} \times \{c\})$

**then obtain**  $aa$  **where**  $aa\text{-in}$ :  $aa \in \{a..b\}$  **and**  $x\text{-eq}$ :  $x = h\ (aa, c)$

**by** *auto*

**have**  $(aa, c) \in \{a..b\} \times \{c..d\}$

**using**  $aa\text{-in}\ cd$  **by** *auto*

**then show**  $x \in V$

**using**  $rectV\ x\text{-eq}$  **by** *blast*

qed

**have**  $segd\text{-in-}V$ :  $h \text{ ' } (\{a..b\} \times \{d\}) \subseteq V$

```

proof
  fix  $x$ 
  assume  $x\text{-in}: x \in h \text{ ' } (\{a..b\} \times \{d\})$ 
  then obtain  $aa$  where  $aa\text{-in}: aa \in \{a..b\}$  and  $x\text{-eq}: x = h (aa, d)$ 
  by auto
  have  $(aa, d) \in \{a..b\} \times \{c..d\}$ 
  using  $aa\text{-in}$   $cd$  by auto
  then show  $x \in V$ 
  using  $rectV$   $x\text{-eq}$  by blast
qed
have  $segc\text{-loop}: segment\text{-loop} (\lambda t. h (t, c)) a b \in loop\text{-space } V x0$ 
  by (rule horizontal-rectangle-segment-loop-in-set[ $OF$   $h\text{-cont}$   $a01$   $b01$   $c01$   $ab$ ])
  (use  $segc\text{-in-}V$   $acUV$   $bcUV$  in auto)
have  $segd\text{-loop}: segment\text{-loop} (\lambda t. h (t, d)) a b \in loop\text{-space } V x0$ 
proof –
  have  $UVV: U \cap V \subseteq V$ 
  by blast
  show ?thesis
  by (rule horizontal-rectangle-segment-loop-in-set[ $OF$   $h\text{-cont}$   $a01$   $b01$   $d01$   $ab$ 
 $segd\text{-in-}V$   $adUV$   $bdUV$   $UVV$ ])
qed
have  $bridge\text{-a-loop}: bridge\text{-loop } h a c d \in loop\text{-space } (U \cap V) x0$ 
  by (rule vertical-bridge-loop-in-set[ $OF$   $h\text{-cont}$   $a01$   $c01$   $d01$   $cd$   $leftUV$ ]) simp
have  $bridge\text{-b-loop}: bridge\text{-loop } h b c d \in loop\text{-space } (U \cap V) x0$ 
  by (rule vertical-bridge-loop-in-set[ $OF$   $h\text{-cont}$   $b01$   $c01$   $d01$   $cd$   $rightUV$ ]) simp

have  $bottom\text{-in}: bottom \in G2$ 
  unfolding  $bottom\text{-def}$  by (rule loop-class-in-space[ $OF$   $segc\text{-loop}$ ])
have  $top\text{-in}: top \in G2$ 
  unfolding  $top\text{-def}$  by (rule loop-class-in-space[ $OF$   $segd\text{-loop}$ ])
have  $ga\text{-in-}H: ga \in H$ 
  unfolding  $ga\text{-def}$  by (rule loop-class-in-space[ $OF$   $bridge\text{-a-loop}$ ])
have  $gb\text{-in-}H: gb \in H$ 
  unfolding  $gb\text{-def}$  by (rule loop-class-in-space[ $OF$   $bridge\text{-b-loop}$ ])
have  $ga\text{-in}: i2 ga \in G2$ 
  by (rule i2-in-}G2[ $OF$   $ga\text{-in-}H$ ])
have  $gb\text{-in}: i2 gb \in G2$ 
  by (rule i2-in-}G2[ $OF$   $gb\text{-in-}H$ ])
have  $ab\text{-in}: mult2 bottom (i2 gb) \in G2$ 
  by (rule fundamental-group-mult-in-space[ $OF$   $bottom\text{-in}$   $gb\text{-in}$ ])
have  $cd\text{-in}: mult2 (i2 ga) top \in G2$ 
  by (rule fundamental-group-mult-in-space[ $OF$   $ga\text{-in}$   $top\text{-in}$ ])

have  $ga\text{-eq}: i2 ga = loop\text{-class } V x0 (bridge\text{-loop } h a c d)$ 
  unfolding  $ga\text{-def}$ 
  by (subst i2-loop-class-eq[ $OF$   $bridge\text{-a-loop}$ ]) simp
have  $gb\text{-eq}: i2 gb = loop\text{-class } V x0 (bridge\text{-loop } h b c d)$ 
  unfolding  $gb\text{-def}$ 
  by (subst i2-loop-class-eq[ $OF$   $bridge\text{-b-loop}$ ]) simp

```

```

have mult-eq: mult2 bottom (i2 gb) = mult2 (i2 ga) top
  unfolding bottom-def top-def ga-eq gb-eq
  by (rule rectangle-segment-loop-bridge-class-eq[OF h-cont a01 b01 c01 d01 ab
cd rectV leftUV rightUV])
    auto
have pair-eq:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordRight bottom (WordRight (i2 gb) rest))
  (WordRight (i2 ga) (WordRight top rest))
  by (rule carrier-full-amalg-equiv-right-pair-eq[OF bottom-in gb-in ab-in ga-in
top-in cd-in rest-in mult-eq])

show ?thesis
  using pair-eq
  by (simp only: bottom-def top-def bridge-word.simps ga-def gb-def)
qed

lemma rectangle-partition-partition-word-with-tail-in-space:
  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 ∈ {0..1} and d01: d ∈ {0..1} and cd: c ≤ d
  and y-in: y ∈ {c, d}
  and part: rectangle-partition h c d ts bs
  and edgeUV:  $\bigwedge t. t \in \text{set } ts \implies h \text{ ' } (\{t\} \times \{c..d\}) \subseteq U \cap V$ 
  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)
  using part edgeUV
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by (cases bs) (simp-all add: tail-in)
next
  case (Cons t ts)
  show ?case
  proof (cases ts)
  case Nil
  then show ?thesis
    using Cons.prem1 tail-in by (cases bs) simp-all
  next
  case (Cons u us)
  obtain b bs' where bs: bs = b # bs'
    using Cons.prem1 Cons by (cases bs) auto
  have tail-part: rectangle-partition h c d (u # us) bs'
    using Cons.prem1 Cons bs by simp
  have tail-edgeUV:
     $\bigwedge x. x \in \text{set } (u \# us) \implies h \text{ ' } (\{x\} \times \{c..d\}) \subseteq U \cap V$ 
    using Cons.prem1 Cons by auto
  have tail-in':

```

$fpw\text{-in-space } G1 \ G2$  (partition-word-with-tail  $(\lambda t. h (t, y))$  ( $u \# us$ )  $bs'$  rest)  
**using**  $h\text{-cont } c01 \ d01 \ cd \ y\text{-in } Cons.IH[of \ bs'] \ Cons \ tail\text{-part } tail\text{-edge}UV \ tail\text{-in}$   
 $bs$   
**by simp**

**have**  $t01: t \in \{0..1\}$  **and**  $u01: u \in \{0..1\}$  **and**  $tu: t < u$   
**and**  $rect\text{-side}$ :  
*(if b then  $h' (\{t..u\} \times \{c..d\}) \subseteq U$  else  $h' (\{t..u\} \times \{c..d\}) \subseteq V$ )*  
**using**  $Cons.prem1 \ Cons \ bs$  **by simp-all**

**have**  $y01: y \in \{0..1\}$   
**using**  $y\text{-in } c01 \ d01$  **by auto**

**have**  $y\text{-cd}: y \in \{c..d\}$   
**using**  $y\text{-in } cd$  **by auto**

**have**  $tu\text{-le}: t \leq u$   
**using**  $tu$  **by simp**

**have**  $tyUV: h (t, y) \in U \cap V$

**proof** –

**have**  $t\text{-edge}: h' (\{t\} \times \{c..d\}) \subseteq U \cap V$   
**using**  $Cons.prem2[of t] \ Cons$  **by simp**

**moreover** **have**  $(t, y) \in \{t\} \times \{c..d\}$   
**using**  $y\text{-cd}$  **by auto**

**ultimately show**  $?thesis$   
**using**  $t\text{-edge}$   
**by blast**

**qed**

**have**  $uyUV: h (u, y) \in U \cap V$

**proof** –

**have**  $u\text{-edge}: h' (\{u\} \times \{c..d\}) \subseteq U \cap V$   
**using**  $Cons.prem2[of u] \ Cons$  **by simp**

**moreover** **have**  $(u, y) \in \{u\} \times \{c..d\}$   
**using**  $y\text{-cd}$  **by auto**

**ultimately show**  $?thesis$   
**using**  $u\text{-edge}$   
**by blast**

**qed**

**show**  $?thesis$

**proof** ( $cases \ b$ )

**case**  $True$

**have**  $segU: h' (\{t..u\} \times \{y\}) \subseteq U$

**proof**

**fix**  $z$

**assume**  $z\text{-in}: z \in h' (\{t..u\} \times \{y\})$

**then obtain**  $a$  **where**  $a\text{-in}: a \in \{t..u\}$  **and**  $z\text{-eq}: z = h (a, y)$   
**by auto**

**have**  $(a, y) \in \{t..u\} \times \{c..d\}$   
**using**  $a\text{-in } y\text{-cd}$  **by auto**

**then have**  $z\text{-strip}: z \in h' (\{t..u\} \times \{c..d\})$   
**using**  $z\text{-eq}$  **by blast**

```

    have stripU: h ' ({t..u} × {c..d}) ⊆ U
      using rect-side True by simp
    then show z ∈ U
      using z-strip by blast
  qed
  have segU: segment-loop (λt. h (t, y)) t u ∈ loop-space U x0
    by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont t01 u01 y01
tu-le])
    (use segyU tyUV uyUV in auto)
  have seg-in: loop-class U x0 (segment-loop (λt. h (t, y)) t u) ∈ G1
    by (rule loop-class-in-space[OF segU])
  show ?thesis
    using True seg-in tail-in' bs Cons by simp
next
case False
have segyV: h ' ({t..u} × {y}) ⊆ V
proof
  fix z
  assume z-in: z ∈ h ' ({t..u} × {y})
  then obtain a where a-in: a ∈ {t..u} and z-eq: z = h (a, y)
    by auto
  have (a, y) ∈ {t..u} × {c..d}
    using a-in y-cd by auto
  then have z-strip: z ∈ h ' ({t..u} × {c..d})
    using z-eq by blast
  have stripV: h ' ({t..u} × {c..d}) ⊆ V
    using rect-side False by simp
  then show z ∈ V
    using z-strip by blast
qed
have segV: segment-loop (λt. h (t, y)) t u ∈ loop-space V x0
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont t01 u01 y01
tu-le])
  (use segyV tyUV uyUV in auto)
have seg-in: loop-class V x0 (segment-loop (λt. h (t, y)) t u) ∈ G2
  by (rule loop-class-in-space[OF segV])
show ?thesis
  using False seg-in tail-in' bs Cons by simp
qed
qed
qed

```

**lemma** *rectangle-partition-partition-word-with-tail-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 ∈ {0..1} and d01: d ∈ {0..1} and cd: c ≤ d
  and part: rectangle-partition h c d (t # u # ts) (b # bs)
  and edgeUV: ∧x. x ∈ set (t # u # ts) ⇒ h ' ({x} × {c..d}) ⊆ U ∩ 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 (λx. 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 (λx. h (x, d)) (t # u # ts) (b # bs) rest))
using part rest-in edgeUV
proof (induction bs arbitrary: t u ts b rest)
  case Nil
  have ts-nil: ts = []
  proof (cases ts)
    case Nil
    then show ?thesis .
  next
  case (Cons v vs)
  with Nil.prem(1) show ?thesis
  by simp
qed
have u1: u = 1
  using Nil.prem(1) ts-nil by simp
have t01: t ∈ {0..1} and u01: u ∈ {0..1} and tu: t < u
  using Nil.prem(1) ts-nil by simp-all
have tu-le: t ≤ u
  using tu by simp
have rect-side: (if b then h ‘ ({t..u} × {c..d}) ⊆ U else h ‘ ({t..u} × {c..d}) ⊆
V)
  using Nil.prem(1) ts-nil u1 by simp
have t-in: t ∈ set (t # u # ts)
  by simp
have u-in: u ∈ set (t # u # ts)
  by simp
have leftUV: h ‘ ({t} × {c..d}) ⊆ U ∩ V
  by (rule Nil.prem(3)[OF t-in])
have rightUV: h ‘ ({u} × {c..d}) ⊆ U ∩ V
  by (rule Nil.prem(3)[OF u-in])
show ?case
proof (cases b)
  case True
  have rectU: h ‘ ({t..u} × {c..d}) ⊆ U
  using rect-side True by simp
  have rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordLeft (loop-class U x0 (segment-loop (λx. h (x, c)) t u))
      (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h u c d)) rest))
    (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h t c d))
      (WordLeft (loop-class U x0 (segment-loop (λx. h (x, d)) t u)) rest))
  by (rule rectangle-segment-bridge-left-equiv[
    where h = h and a = t and b = u and c = c and d = d and rest =

```

```

rest,
  OF h-cont t01 u01 c01 d01 tu-le cd rectU leftUV rightUV Nil.prem(2)]
have lhs-eq:
  partition-word-with-tail ( $\lambda x. h(x, c)$ ) (t # u # ts) [b]
    (bridge-word (last [b])
      (loop-class (U  $\cap$  V) x0 (bridge-loop h (last (t # u # ts)) c d)) rest) =
  WordLeft (loop-class U x0 (segment-loop ( $\lambda x. h(x, c)$ ) t u))
    (bridge-word True (loop-class (U  $\cap$  V) x0 (bridge-loop h u c d)) rest)
using True ts-nil by simp
have rhs-eq:
  bridge-word b (loop-class (U  $\cap$  V) x0 (bridge-loop h t c d))
    (partition-word-with-tail ( $\lambda x. h(x, d)$ ) (t # u # ts) [b] rest) =
  bridge-word True (loop-class (U  $\cap$  V) x0 (bridge-loop h t c d))
    (WordLeft (loop-class U x0 (segment-loop ( $\lambda x. h(x, d)$ ) t u)) rest)
using True ts-nil by simp
show ?thesis
unfolding lhs-eq rhs-eq by (rule rel)
next
case False
have rectV: h '({t..u}  $\times$  {c..d})  $\subseteq$  V
using rect-side False by simp
have rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordRight (loop-class V x0 (segment-loop ( $\lambda x. h(x, c)$ ) t u))
      (bridge-word False (loop-class (U  $\cap$  V) x0 (bridge-loop h u c d)) rest))
    (bridge-word False (loop-class (U  $\cap$  V) x0 (bridge-loop h t c d))
      (WordRight (loop-class V x0 (segment-loop ( $\lambda x. h(x, d)$ ) t u)) rest))
by (rule rectangle-segment-bridge-right-equiv[
  where h = h and a = t and b = u and c = c and d = d and rest =
rest,
```

OF h-cont t01 u01 c01 d01 tu-le cd rectV leftUV rightUV Nil.prem(2)]

```

have lhs-eq:
  partition-word-with-tail ( $\lambda x. h(x, c)$ ) (t # u # ts) [b]
    (bridge-word (last [b])
      (loop-class (U  $\cap$  V) x0 (bridge-loop h (last (t # u # ts)) c d)) rest) =
  WordRight (loop-class V x0 (segment-loop ( $\lambda x. h(x, c)$ ) t u))
    (bridge-word False (loop-class (U  $\cap$  V) x0 (bridge-loop h u c d)) rest)
using False ts-nil by simp
have rhs-eq:
  bridge-word b (loop-class (U  $\cap$  V) x0 (bridge-loop h t c d))
    (partition-word-with-tail ( $\lambda x. h(x, d)$ ) (t # u # ts) [b] rest) =
  bridge-word False (loop-class (U  $\cap$  V) x0 (bridge-loop h t c d))
    (WordRight (loop-class V x0 (segment-loop ( $\lambda x. h(x, d)$ ) t u)) rest)
using False ts-nil by simp
show ?thesis
unfolding lhs-eq rhs-eq by (rule rel)
qed
next
case (Cons cflag bs')
```

```

obtain  $v$   $us$  where  $ts: ts = v \# us$ 
  using  $Cons.prem(1)$  by  $(cases\ ts)\ simp\ all$ 
have  $t01: t \in \{0..1\}$  and  $u01: u \in \{0..1\}$  and  $tu: t < u$ 
  and  $rect\ side: (if\ b\ then\ h\ '(\{t..u\} \times \{c..d\}) \subseteq U\ else\ h\ '(\{t..u\} \times \{c..d\}) \subseteq$ 
 $V)$ 
  and  $tail\ part: rectangle\ partition\ h\ c\ d\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')$ 
  using  $Cons.prem(1)$   $ts$  by  $simp\ all$ 
have  $tu\ le: t \leq u$ 
  using  $tu$  by  $simp$ 
have  $t\ in: t \in set\ (t\ \#\ u\ \#\ ts)$ 
  by  $simp$ 
have  $u\ in: u \in set\ (t\ \#\ u\ \#\ ts)$ 
  by  $simp$ 
have  $leftUV: h\ '(\{t\} \times \{c..d\}) \subseteq U \cap V$ 
  by  $(rule\ Cons.prem(3)[OF\ t\ in])$ 
have  $midUV: h\ '(\{u\} \times \{c..d\}) \subseteq U \cap V$ 
  by  $(rule\ Cons.prem(3)[OF\ u\ in])$ 
have  $top\ tail\ in:$ 
   $fpw\ in\ space\ G1\ G2$ 
   $(partition\ word\ with\ tail\ (\lambda x. h\ (x, d))\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')\ rest)$ 
  by  $(rule\ rectangle\ partition\ partition\ word\ with\ tail\ in\ space[OF\ h\ cont\ c01\ d01$ 
 $cd])$ 
   $(use\ tail\ part\ Cons.prem(3)\ Cons.prem(2)\ ts\ in\ auto)$ 
have  $gu\ loop: bridge\ loop\ h\ u\ c\ d \in loop\ space\ (U \cap V)\ x0$ 
  by  $(rule\ vertical\ bridge\ loop\ in\ set[OF\ h\ cont\ u01\ c01\ d01\ cd])\ (use\ midUV\ in$ 
 $auto)$ 
have  $gu\ in\ H: loop\ class\ (U \cap V)\ x0\ (bridge\ loop\ h\ u\ c\ d) \in H$ 
  by  $(rule\ loop\ class\ in\ space[OF\ gu\ loop])$ 
have  $tail\ rel:$ 
   $carrier\ full\ amalg\ equiv\ G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$ 
   $(partition\ word\ with\ tail\ (\lambda x. h\ (x, c))\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')$ 
   $(bridge\ word\ (last\ (cflag\ \#\ bs'))$ 
   $(loop\ class\ (U \cap V)\ x0\ (bridge\ loop\ h\ (last\ (u\ \#\ v\ \#\ us))\ c\ d))\ rest))$ 
   $(bridge\ word\ cflag\ (loop\ class\ (U \cap V)\ x0\ (bridge\ loop\ h\ u\ c\ d))$ 
   $(partition\ word\ with\ tail\ (\lambda x. h\ (x, d))\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')\ rest))$ 
  by  $(rule\ Cons.IH[OF\ tail\ part])$ 
   $(use\ h\ cont\ c01\ d01\ cd\ Cons.prem(3)\ Cons.prem(2)\ ts\ in\ auto)$ 
have  $tail\ switched:$ 
   $carrier\ full\ amalg\ equiv\ G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$ 
   $(partition\ word\ with\ tail\ (\lambda x. h\ (x, c))\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')$ 
   $(bridge\ word\ (last\ (cflag\ \#\ bs'))$ 
   $(loop\ class\ (U \cap V)\ x0\ (bridge\ loop\ h\ (last\ (u\ \#\ v\ \#\ us))\ c\ d))\ rest))$ 
   $(bridge\ word\ b\ (loop\ class\ (U \cap V)\ x0\ (bridge\ loop\ h\ u\ c\ d))$ 
   $(partition\ word\ with\ tail\ (\lambda x. h\ (x, d))\ (u\ \#\ v\ \#\ us)\ (cflag\ \#\ bs')\ rest))$ 
proof  $(cases\ b = cflag)$ 
  case  $True$ 
  then show  $?thesis$ 
  using  $tail\ rel$  by  $simp$ 
next

```

```

case False
have switch:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (bridge-word cflag (loop-class (U ∩ V) x0 (bridge-loop h u c d))
   (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs') rest))
  (bridge-word b (loop-class (U ∩ V) x0 (bridge-loop h u c d))
   (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs') rest))
  by (rule carrier-full-amalg-equiv.sym[OF bridge-word-switch[OF gu-in-H
False]])
show ?thesis
  by (rule carrier-full-amalg-equiv.trans[OF tail-rel switch])
qed
show ?case
proof (cases b)
  case True
  have seg-in: loop-class U x0 (segment-loop (λx. h (x, c)) t u) ∈ G1
  proof –
    have line-subset:  $\{t..u\} \times \{c\} \subseteq \{t..u\} \times \{c..d\}$ 
    using c01 cd by auto
    have rectU:  $h \text{ ‘ } (\{t..u\} \times \{c..d\}) \subseteq U$ 
    using rect-side True by simp
    have segU-line:  $h \text{ ‘ } (\{t..u\} \times \{c\}) \subseteq U$ 
    by (rule order-trans[OF image-mono[OF line-subset] rectU])
    have tcUV:  $h (t, c) \in U \cap V$ 
    by (rule subsetD[OF leftUV]) (use cd in auto)
    have ucUV:  $h (u, c) \in U \cap V$ 
    by (rule subsetD[OF midUV]) (use cd in auto)
    have seg-loop: segment-loop (λx. h (x, c)) t u ∈ loop-space U x0
    by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont t01 u01 c01 tu-le
segU-line tcUV ucUV]) auto
    show ?thesis
    by (rule loop-class-in-space[OF seg-loop])
  qed
have pref-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordLeft (loop-class U x0 (segment-loop (λx. h (x, c)) t u))
   (partition-word-with-tail (λx. h (x, c)) (u # v # us) (cflag # bs')
   (bridge-word (last (cflag # bs'))
   (loop-class (U ∩ V) x0 (bridge-loop h (last (u # v # us)) c d)) rest)))
  (WordLeft (loop-class U x0 (segment-loop (λx. h (x, c)) t u))
   (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h u c d))
   (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs') rest)))
proof –
  have tail-true:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word-with-tail (λx. h (x, c)) (u # v # us) (cflag # bs')
     (bridge-word (last (cflag # bs'))
     (loop-class (U ∩ V) x0 (bridge-loop h (last (u # v # us)) c d)) rest))
    (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h u c d)))

```

```

      (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs^) rest))
    using tail-switched True by simp
  show ?thesis
    by (rule carrier-full-amalg-equiv-left-context[OF tail-true seg-in])
qed
have cell-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordLeft (loop-class U x0 (segment-loop (λx. h (x, c)) t u))
    (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h u c d))
      (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs^) rest)))
  (bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h t c d))
    (WordLeft (loop-class U x0 (segment-loop (λx. h (x, d)) t u))
      (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs^) rest)))
  by (rule rectangle-segment-bridge-left-equiv[OF h-cont t01 u01 c01 d01 tu-le
cd])
  (use rect-side True leftUV midUV top-tail-in in auto)
have lhs-eq:
  partition-word-with-tail (λx. h (x, c)) (t # u # ts) (b # cflag # bs^)
  (bridge-word (last (b # cflag # bs^))
    (loop-class (U ∩ V) x0 (bridge-loop h (last (t # u # ts)) c d)) rest) =
  WordLeft (loop-class U x0 (segment-loop (λx. h (x, c)) t u))
  (partition-word-with-tail (λx. h (x, c)) (u # v # us) (cflag # bs^)
    (bridge-word (last (cflag # bs^))
      (loop-class (U ∩ V) x0 (bridge-loop h (last (u # v # us)) c d)) rest))
  using True ts by simp
have rhs-eq:
  bridge-word b (loop-class (U ∩ V) x0 (bridge-loop h t c d))
  (partition-word-with-tail (λx. h (x, d)) (t # u # ts) (b # cflag # bs^) rest)
=
  bridge-word True (loop-class (U ∩ V) x0 (bridge-loop h t c d))
  (WordLeft (loop-class U x0 (segment-loop (λx. h (x, d)) t u))
    (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs^) rest))
  using True ts by simp
show ?thesis
  unfolding lhs-eq rhs-eq by (rule carrier-full-amalg-equiv.trans[OF pref-rel
cell-rel])
next
case False
have seg-in: loop-class V x0 (segment-loop (λx. h (x, c)) t u) ∈ G2
proof -
  have line-subset: {t..u} × {c} ⊆ {t..u} × {c..d}
    using c01 cd by auto
  have rectV: h ‘ {t..u} × {c..d} ⊆ V
    using rect-side False by simp
  have segV-line: h ‘ {t..u} × {c} ⊆ V
    by (rule order-trans[OF image-mono[OF line-subset] rectV])
  have tcUV: h (t, c) ∈ U ∩ V
    by (rule subsetD[OF leftUV]) (use cd in auto)
  have ucUV: h (u, c) ∈ U ∩ V

```

```

    by (rule subsetD[OF midUV]) (use cd in auto)
  have seg-loop: segment-loop ( $\lambda x. h(x, c)$ )  $t u \in \text{loop-space } V x0$ 
  by (rule horizontal-rectangle-segment-loop-in-set[OF h-cont t01 u01 c01 tu-le
segV-line tcUV ucUV]) auto
  show ?thesis
  by (rule loop-class-in-space[OF seg-loop])
qed
have pref-rel:
  carrier-full-amalg-equiv  $G1 G2 H i1 i2 mult1 one1 mult2 one2$ 
  (WordRight (loop-class  $V x0$  (segment-loop ( $\lambda x. h(x, c)$ )  $t u$ ))
  (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ )
  (bridge-word (last (cflag #  $bs'$ ))
  (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h$  (last ( $u \# v \# us$ ))  $c d$ )) rest)))
  (WordRight (loop-class  $V x0$  (segment-loop ( $\lambda x. h(x, c)$ )  $t u$ ))
  (bridge-word False (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h u c d$ ))
  (partition-word-with-tail ( $\lambda x. h(x, d)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ ) rest)))
proof -
  have tail-false:
    carrier-ful-amalg-equiv  $G1 G2 H i1 i2 mult1 one1 mult2 one2$ 
    (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ )
    (bridge-word (last (cflag #  $bs'$ ))
    (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h$  (last ( $u \# v \# us$ ))  $c d$ )) rest))
    (bridge-word False (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h u c d$ ))
    (partition-word-with-tail ( $\lambda x. h(x, d)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ ) rest))
  using tail-switched False by simp
  show ?thesis
  by (rule carrier-full-amalg-equiv-right-context[OF tail-false seg-in])
qed
have cell-rel:
  carrier-full-amalg-equiv  $G1 G2 H i1 i2 mult1 one1 mult2 one2$ 
  (WordRight (loop-class  $V x0$  (segment-loop ( $\lambda x. h(x, c)$ )  $t u$ ))
  (bridge-word False (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h u c d$ ))
  (partition-word-with-tail ( $\lambda x. h(x, d)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ ) rest)))
  (bridge-word False (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h t c d$ ))
  (WordRight (loop-class  $V x0$  (segment-loop ( $\lambda x. h(x, d)$ )  $t u$ ))
  (partition-word-with-tail ( $\lambda x. h(x, d)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ ) rest)))
  by (rule rectangle-segment-bridge-right-equiv[OF h-cont t01 u01 c01 d01 tu-le
cd])
  (use rect-side False leftUV midUV top-tail-in in auto)
  have lhs-eq:
    partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $t \# u \# ts$ ) ( $b \# cflag \# bs'$ )
    (bridge-word (last ( $b \# cflag \# bs'$ ))
    (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h$  (last ( $t \# u \# ts$ ))  $c d$ )) rest) =
    WordRight (loop-class  $V x0$  (segment-loop ( $\lambda x. h(x, c)$ )  $t u$ ))
    (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $u \# v \# us$ ) (cflag #  $bs'$ )
    (bridge-word (last (cflag #  $bs'$ ))
    (loop-class ( $U \cap V$ )  $x0$  (bridge-loop  $h$  (last ( $u \# v \# us$ ))  $c d$ )) rest))
  using False  $ts$  by simp
  have rhs-eq:

```

```

    bridge-word b (loop-class (U ∩ V) x0 (bridge-loop h t c d))
      (partition-word-with-tail (λx. h (x, d)) (t # u # ts) (b # cflag # bs') rest)
  =
    bridge-word False (loop-class (U ∩ V) x0 (bridge-loop h t c d))
      (WordRight (loop-class V x0 (segment-loop (λx. h (x, d)) t u))
        (partition-word-with-tail (λx. h (x, d)) (u # v # us) (cflag # bs') rest))
    using False ts by simp
  show ?thesis
    unfolding lhs-eq rhs-eq by (rule carrier-full-amalg-equiv.trans[OF pref-rel
cell-rel])
  qed
qed

```

**lemma** *homotopic-paths-join-subpathin*:

```

  assumes p-path: path p
    and p-img: path-image p ⊆ S
    and u01: u ∈ {0..1}
    and v01: v ∈ {0..1}
    and w01: w ∈ {0..1}
    and uv: u ≤ v
    and vw: v ≤ w
  shows homotopic-paths S (subpathin u v p +++ subpathin v w p) (subpathin u
w p)
  proof (cases w = u)
  case True
  then have uvw: u = v v = w
    using uv vw by linarith+
  have pu-in: p u ∈ S
    using p-img u01 by (auto simp: path-image-def)
  have const-join: subpathin u v p +++ subpathin v w p = (λ-. p u)
    using uvw by (simp add: fun-eq-iff joinpaths-def subpathin-def)
  have const-subpath: subpathin u w p = (λ-. p u)
    using uvw by (simp add: fun-eq-iff subpathin-def)
  show ?thesis
    unfolding const-join const-subpath
    using pu-in by (simp add: path-def)
  next
  case False
  define a where a = (v - u) / (w - u)
  have wu-pos: 0 < w - u
    using False uv vw by linarith
  have a-nonneg: 0 ≤ a
    unfolding a-def using uv wu-pos by (simp add: field-simps)
  have a-le1: a ≤ 1
    unfolding a-def using uv vw wu-pos by (simp add: field-simps)
  have a01: a ∈ {0..1}
    using a-nonneg a-le1 by auto
  let ?f = λt::real. if t ≤ 1 / 2 then 2 * a * t else a + (1 - a) * (2 * t - 1)
  have f-eq: ?f = subpathin 0 a id +++ subpathin a 1 id

```

```

    by (rule ext) (simp add: joinpaths-def subpathin-def)
  have contf: continuous-on {0..1} ?f
proof -
  have id-path: path id
    by (simp add: path-def)
  have left-path: path (subpathin 0 a id)
    by (rule path-subpathin[OF id-path]) (use a01 in auto)
  have right-path: path (subpathin a 1 id)
    by (rule path-subpathin[OF id-path]) (use a01 in auto)
  have join-path: path (subpathin 0 a id +++ subpathin a 1 id)
    using left-path right-path a01
    by (simp add: pathstart-def pathfinish-def subpathin-def)
  show ?thesis
    unfolding f-eq
    using join-path by (simp add: path-def)
qed
have f01: ?f ∈ {0..1} → {0..1}
proof
  fix t :: real
  assume t01: t ∈ {0..1}
  show ?f t ∈ {0..1}
proof (cases t ≤ 1 / 2)
  case True
  have t-nonneg: 0 ≤ t
    using t01 by auto
  have two-t-le1: 2 * t ≤ 1
    using t01 True by linarith
  have lower: 0 ≤ 2 * a * t
    using a-nonneg t-nonneg by (intro mult-nonneg-nonneg) auto
  have upper1: a * (2 * t) ≤ a * 1
    using a-nonneg two-t-le1 by (intro mult-left-mono) auto
  have upper: 2 * a * t ≤ 1
    using upper1 a-le1 by (simp add: algebra-simps)
  show ?thesis
    using True lower upper by auto
next
  case False
  have two-t-minus1-nonneg: 0 ≤ 2 * t - 1
    using t01 False by linarith
  have two-t-minus1-le1: 2 * t - 1 ≤ 1
proof -
  have t-le1: t ≤ 1
    using t01 by auto
  then show ?thesis
    by linarith
qed
have one-minus-a-nonneg: 0 ≤ 1 - a
  using a-le1 by linarith
have lower: 0 ≤ a + (1 - a) * (2 * t - 1)

```

```

    using a-nonneg one-minus-a-nonneg two-t-minus1-nonneg
    by (intro add-nonneg-nonneg mult-nonneg-nonneg)
  have upper1:  $(1 - a) * (2 * t - 1) \leq (1 - a) * 1$ 
    using one-minus-a-nonneg two-t-minus1-le1 by (intro mult-left-mono) auto
  have upper:  $a + (1 - a) * (2 * t - 1) \leq 1$ 
  proof -
    have  $a + (1 - a) * (2 * t - 1) \leq a + (1 - a) * 1$ 
      using upper1 by linarith
    also have ... = 1
      by simp
    finally show ?thesis .
  qed
  show ?thesis
    using False lower upper by auto
  qed
  qed
  have sub-uw-path: path (subpathin u w p)
    by (rule path-subpathin[OF p-path u01 w01])
  have sub-uw-img: path-image (subpathin u w p)  $\subseteq S$ 
    using p-img path-image-subpathin-subset[OF u01 w01, of p] by blast
  show ?thesis
  proof (rule homotopic-paths-sym[OF homotopic-paths-reparametrize[where p =
subpathin u w p and f = ?f]])
    show path (subpathin u w p)
      by (rule sub-uw-path)
    show path-image (subpathin u w p)  $\subseteq S$ 
      by (rule sub-uw-img)
    show continuous-on  $\{0..1\}$  ?f
      by (rule contf)
    show ?f  $\in \{0..1\} \rightarrow \{0..1\}$ 
      by (rule f01)
    show ?f 0 = 0
      by simp
    show ?f 1 = 1
      using a-nonneg a-le1 by simp
    show (subpathin u v p +++ subpathin v w p) t = subpathin u w p (?f t)
      if t01:  $t \in \{0..1\}$  for t
    proof (cases t  $\leq 1 / 2$ )
      case True
      have (subpathin u v p +++ subpathin v w p) t = p (u + (v - u) * (2 * t))
        using True by (simp add: joinpaths-def subpathin-def algebra-simps)
      also have ... = p (u + (w - u) * (2 * a * t))
      proof -
        have wa-eq:  $(w - u) * a = v - u$ 
          unfolding a-def using wu-pos by (simp add: field-simps algebra-simps)
        have scale-eq:  $(w - u) * (2 * a * t) = (v - u) * (2 * t)$ 
        proof -
          have  $(w - u) * (2 * a * t) = ((w - u) * a) * (2 * t)$ 
            by (simp add: algebra-simps)

```

```

    also have ... = (v - u) * (2 * t)
      using wa-eq by simp
    finally show ?thesis .
  qed
  have arg-eq: u + (v - u) * (2 * t) = u + (w - u) * (2 * a * t)
    using scale-eq by linarith
  show ?thesis
    by (subst arg-eq) simp
  qed
  also have ... = subpathin u w p (?f t)
    using True by (simp add: subpathin-def algebra-simps)
  finally show ?thesis .
next
case False
have (subpathin u v p +++ subpathin v w p) t = p (v + (w - v) * (2 * t -
1))
  using t01 False by (simp add: joinpaths-def subpathin-def algebra-simps)
also have ... = p (u + (w - u) * (a + (1 - a) * (2 * t - 1)))
proof -
  have one-minus-a: 1 - a = (w - v) / (w - u)
    unfolding a-def using wu-pos by (simp add: field-simps)
  have ua-eq: u + (w - u) * a = v
    unfolding a-def using wu-pos by (simp add: field-simps)
  have tail-eq: (w - u) * (1 - a) = w - v
    using one-minus-a wu-pos by (simp add: field-simps)
  have arg-eq: u + (w - u) * (a + (1 - a) * (2 * t - 1)) = v + (w - v) *
(2 * t - 1)
  proof -
    have u + (w - u) * (a + (1 - a) * (2 * t - 1)) =
      (u + (w - u) * a) + ((w - u) * (1 - a)) * (2 * t - 1)
      by (simp add: algebra-simps)
    also have ... = v + (w - v) * (2 * t - 1)
      using ua-eq tail-eq by simp
    finally show ?thesis .
  qed
  qed
  show ?thesis
    using arg-eq by simp
  qed
  also have ... = subpathin u w p (?f t)
    using False by (simp add: subpathin-def algebra-simps)
  finally show ?thesis .
qed
qed
qed

```

```

lemma subpathin-full [simp]:
  subpathin 0 1 p = p
  unfolding subpathin-def by (rule ext) simp

```

**lemma** *segment-loop-refl*:  
**assumes**  $puUV: p\ u \in U \cap V$   
**shows** *homotopic-paths*  $W$  (*segment-loop*  $p\ u\ u$ )  $(\lambda\cdot. x0)$   
**proof** –  
**let**  $?c = \text{connector } (p\ u)$   
**have**  $c\text{-path}: \text{path } ?c$   
**by** (*rule connector-path*[*OF puUV*])  
**have**  $c\text{-img}: \text{path-image } ?c \subseteq W$   
**using** *connector-image-subset*[*OF puUV*] **by** *blast*  
**have**  $rev\text{-c-path}: \text{path } (\text{reversepath } ?c)$   
**using**  $c\text{-path}$  **by** *simp*  
**have**  $rev\text{-c-img}: \text{path-image } (\text{reversepath } ?c) \subseteq W$   
**using**  $c\text{-img}$  **by** *simp*  
**have**  $hom\text{-rid}: \text{homotopic-paths } W$   $(?c\ \text{+++ } (\lambda\cdot. p\ u))\ ?c$   
**using** *homotopic-paths-rid-const*[*OF c-path c-img*] *connector-finish*[*OF puUV*]  
**by** (*simp add: pathfinish-def*)  
**have** *hom-join*:  
*homotopic-paths*  $W$   $((?c\ \text{+++ } (\lambda\cdot. p\ u))\ \text{+++ } \text{reversepath } ?c)$   $(?c\ \text{+++ } \text{reversepath } ?c)$   
**proof** (*rule homotopic-paths-join-right*[*OF hom-rid rev-c-path rev-c-img*])  
**show**  $\text{pathfinish } (?c\ \text{+++ } (\lambda\cdot. p\ u)) = \text{pathstart } (\text{reversepath } ?c)$   
**using** *connector-finish*[*OF puUV*]  
**by** (*simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def*)  
**qed**  
**have**  $hom\text{-inv}: \text{homotopic-paths } W$   $(?c\ \text{+++ } \text{reversepath } ?c)$   $(\lambda\cdot. x0)$   
**using** *homotopic-paths-rinv-const*[*OF c-path c-img*] *connector-start*[*OF puUV*]  
**by** *simp*  
**have** *homotopic-paths*  $W$  (*segment-loop*  $p\ u\ u$ )  $(?c\ \text{+++ } \text{reversepath } ?c)$   
**unfolding** *segment-loop-def* **by** (*simp add: hom-join*)  
**then show** *?thesis*  
**by** (*rule homotopic-paths-trans*[*OF - hom-inv*])  
**qed**

**lemma** *segment-loop-full*:  
**assumes**  $p\text{-loop}: p \in \text{loop-space } W\ x0$   
**shows** *homotopic-paths*  $W$  (*segment-loop*  $p\ 0\ 1$ )  $p$   
**by** (*rule segment-loop-base-full-in-set*[*OF p-loop*])

**lemma** *homotopic-paths-cancel-middle*:  
**assumes**  $r\text{-path}: \text{path } r$   
**and**  $r\text{-img}: \text{path-image } r \subseteq S$   
**and**  $c\text{-path}: \text{path } c$   
**and**  $c\text{-img}: \text{path-image } c \subseteq S$   
**and**  $s\text{-path}: \text{path } s$   
**and**  $s\text{-img}: \text{path-image } s \subseteq S$   
**and**  $rc: \text{pathfinish } r = \text{pathfinish } c$   
**and**  $cs: \text{pathstart } s = \text{pathfinish } c$   
**shows** *homotopic-paths*  $S$   $((((r\ \text{+++ } \text{reversepath } c)\ \text{+++ } c)\ \text{+++ } s))$   $(r\ \text{+++ } s)$

**proof** –

- have** *revc-path*: *path* (*reversepath* *c*)
- using** *c-path* **by** *simp*
- have** *revc-img*: *path-image* (*reversepath* *c*)  $\subseteq S$
- using** *c-img* **by** *simp*
- have** *mid-path*: *path* (*reversepath* *c* +++ *c*)
- using** *revc-path* *c-path* **by** *simp*
- have** *mid-img*: *path-image* (*reversepath* *c* +++ *c*)  $\subseteq S$
- by** (*rule subset-path-image-join*[*OF revc-img c-img*])
- have** *hom-cancel0*: *homotopic-paths* *S* (*reversepath* *c* +++ *c*) ( $\lambda$ -. *pathfinish* *c*)
- by** (*rule homotopic-paths-linv-const*[*OF c-path c-img*])
- have** *hom-cancel1*:  
*homotopic-paths* *S* (((*reversepath* *c* +++ *c*) +++ *s*)) ((( $\lambda$ -. *pathfinish* *c*) +++ *s*))
- proof** (*rule homotopic-paths-join-right*[*OF hom-cancel0 s-path s-img*])
- show** *pathfinish* (*reversepath* *c* +++ *c*) = *pathstart* *s*
- using** *cs* **by** (*simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def*)
- qed**
- have** *hom-cancel2*: *homotopic-paths* *S* ((( $\lambda$ -. *pathfinish* *c*) +++ *s*)) *s*
- using** *homotopic-paths-lid-const*[*OF s-path s-img*] *cs* **by** (*simp add: pathstart-def*)
- have** *hom-cancel*: *homotopic-paths* *S* (((*reversepath* *c* +++ *c*) +++ *s*)) *s*
- by** (*rule homotopic-paths-trans*[*OF hom-cancel1 hom-cancel2*])
- have** *hom-left*:  
*homotopic-paths* *S* (*r* +++ ((*reversepath* *c* +++ *c*) +++ *s*)) (*r* +++ *s*)
- proof** (*rule homotopic-paths-join-left*[*OF hom-cancel r-path r-img*])
- show** *pathfinish* *r* = *pathstart* ((*reversepath* *c* +++ *c*) +++ *s*)
- using** *rc* **by** (*simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def*)
- qed**
- have** *assoc1*:  
*homotopic-paths* *S* (((*r* +++ *reversepath* *c*) +++ *c*)) (*r* +++ (*reversepath* *c* +++ *c*))
- proof** –
- have** *homotopic-paths* *S* (*r* +++ (*reversepath* *c* +++ *c*)) (((*r* +++ *reversepath* *c*) +++ *c*))
- by** (*rule homotopic-paths-assoc*[*OF r-path r-img revc-path revc-img c-path c-img*]) (*use rc in simp-all*)
- then show** *?thesis*
- by** (*rule homotopic-paths-sym*)
- qed**
- have** *assoc1-join*:  
*homotopic-paths* *S* ((((*r* +++ *reversepath* *c*) +++ *c*) +++ *s*)) (((*r* +++ (*reversepath* *c* +++ *c*)) +++ *s*))
- proof** (*rule homotopic-paths-join-right*[*OF assoc1 s-path s-img*])
- show** *pathfinish* ((*r* +++ *reversepath* *c*) +++ *c*) = *pathstart* *s*
- using** *rc cs* **by** (*simp add: pathstart-def pathfinish-def joinpaths-def reversepath-def*)

```

qed
have assoc2:
  homotopic-paths S (((r +++ (reversepath c +++ c)) +++ s)) (r +++
((reversepath c +++ c)) +++ s))
proof -
  have homotopic-paths S (r +++ ((reversepath c +++ c)) +++ s)) (((r +++
(reversepath c +++ c)) +++ s))
    by (rule homotopic-paths-assoc[OF r-path r-img mid-path mid-img s-path
s-img]) (use rc cs in simp-all)
    then show ?thesis
      by (rule homotopic-paths-sym)
qed
have homotopic-paths S ((((r +++ reversepath c) +++ c) +++ s)) (r +++
((reversepath c +++ c)) +++ s))
  by (rule homotopic-paths-trans[OF assoc1-join assoc2])
then show ?thesis
  by (rule homotopic-paths-trans[OF - hom-left])
qed

```

**lemma** *segment-loop-join*:

```

assumes p-path: path p
  and p-img: path-image p  $\subseteq$  W
  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
  and puUV: p u  $\in$  U  $\cap$  V
  and pvUV: p v  $\in$  U  $\cap$  V
  and pwUV: p w  $\in$  U  $\cap$  V
shows homotopic-paths W (segment-loop p u v +++ segment-loop p v w)
(segment-loop p u w)
proof -
  let ?cu = connector (p u)
  let ?cv = connector (p v)
  let ?cw = connector (p w)
  let ?suw = subpathin u v p
  let ?svw = subpathin v w p
  let ?suw = subpathin u w p
  let ?head = ?cu +++ ?suw
  let ?tail = ?svw +++ reversepath ?cw

  have cu-path: path ?cu
    by (rule connector-path[OF puUV])
  have cv-path: path ?cv
    by (rule connector-path[OF pvUV])
  have cw-path: path ?cw
    by (rule connector-path[OF pwUV])
  have cu-img: path-image ?cu  $\subseteq$  W

```

**using** *connector-image-subset*[*OF puUV*] **by** *blast*  
**have** *cv-img: path-image ?cv*  $\subseteq W$   
**using** *connector-image-subset*[*OF pvUV*] **by** *blast*  
**have** *cw-img: path-image ?cw*  $\subseteq W$   
**using** *connector-image-subset*[*OF pwUV*] **by** *blast*

**have** *svv-path: path ?svv*  
**by** (*rule path-subpathin*[*OF p-path u01 v01*])  
**have** *svw-path: path ?svw*  
**by** (*rule path-subpathin*[*OF p-path v01 w01*])  
**have** *suw-path: path ?suw*  
**by** (*rule path-subpathin*[*OF p-path u01 w01*])  
**have** *svv-img: path-image ?svv*  $\subseteq W$   
**using** *p-img path-image-subpathin-subset*[*OF u01 v01, of p*] **by** *blast*  
**have** *svw-img: path-image ?svw*  $\subseteq W$   
**using** *p-img path-image-subpathin-subset*[*OF v01 w01, of p*] **by** *blast*  
**have** *suw-img: path-image ?suw*  $\subseteq W$   
**using** *p-img path-image-subpathin-subset*[*OF u01 w01, of p*] **by** *blast*  
**have** *svv-start: pathstart ?svv = p u*  
**by** (*simp add: pathstart-def subpathin-def*)  
**have** *svw-start: pathstart ?svw = p v*  
**by** (*simp add: pathstart-def subpathin-def*)  
**have** *svv-finish: pathfinish ?svv = p w*  
**by** (*simp add: pathfinish-def subpathin-def*)  
**have** *rev-cw-path: path (reversepath ?cw)*  
**using** *cu-path* **by** *simp*  
**have** *rev-cw-img: path-image (reversepath ?cw)*  $\subseteq W$   
**using** *cu-img* **by** *simp*  
**have** *rev-cw-start: pathstart (reversepath ?cw) = p w*  
**using** *connector-finish*[*OF pwUV*] **by** *simp*

**have** *head-path: path ?head*  
**using** *cu-path svv-path connector-finish*[*OF puUV*] *svv-start* **by** *simp*  
**have** *head-img: path-image ?head*  $\subseteq W$   
**by** (*rule subset-path-image-join*[*OF cu-img svv-img*])  
**have** *tail-path: path ?tail*  
**using** *svw-path rev-cw-path svv-finish rev-cw-start* **by** *simp*  
**have** *tail-img: path-image ?tail*  $\subseteq W$   
**by** (*rule subset-path-image-join*[*OF svw-img rev-cw-img*])

**have** *seg-uv-loop: segment-loop p u v*  $\in$  *loop-space W x0*  
**proof** (*rule segment-loop-in-W*[*OF p-path p-img u01 v01 puUV pvUV*])  
**show** *subpathin u v p*  $\in$   $\{0..1\}$   $\subseteq W$   
**using** *svv-img* **by** (*simp add: path-image-def*)

**qed**  
**then have** *seg-uv-path: path (segment-loop p u v)* **and** *seg-uv-img: path-image*  
(*segment-loop p u v*)  $\subseteq W$   
**unfolding** *loop-space-def* **by** *auto*

```

have seg-vw-assoc:
  homotopic-paths W (segment-loop p v w) (?cv +++ ?tail)
proof –
  have homotopic-paths W (?cv +++ (?svw +++ reversepath ?cv)) ((?cv +++
?svw) +++ reversepath ?cv)
    by (rule homotopic-paths-assoc[OF cv-path cv-img svw-path svw-img
rev-cw-path rev-cw-img])
    (use connector-finish[OF pvUV] svw-start svw-finish rev-cw-start in simp-all)
  then show ?thesis
    unfolding segment-loop-def by (rule homotopic-paths-sym)
qed

have seg-join-start: pathfinish (segment-loop p u v) = pathstart (segment-loop p
v w)
  using pvUV by (simp add: segment-loop-def connector-start connector-finish)
have step1:
  homotopic-paths W (segment-loop p u v +++ segment-loop p v w) (segment-loop
p u v +++ (?cv +++ ?tail))
  by (rule homotopic-paths-join-left[OF seg-vw-assoc seg-uv-path seg-uv-img
seg-join-start])

have seg-cv-start: pathfinish (segment-loop p u v) = pathstart ?cv
  using pvUV by (simp add: segment-loop-def connector-start connector-finish)
have cv-tail-start: pathfinish ?cv = pathstart ?tail
  using connector-finish[OF pvUV] svw-start by simp
have step2:
  homotopic-paths W (segment-loop p u v +++ (?cv +++ ?tail)) ((segment-loop
p u v +++ ?cv) +++ ?tail)
  by (rule homotopic-paths-assoc[OF seg-uv-path seg-uv-img cv-path cv-img
tail-path tail-img seg-cv-start cv-tail-start])

have subpath-uv-finish: pathfinish (subpathin u v p) = p v
  by (simp add: subpathin-def pathfinish-def)
have head-finish-pv: pathfinish ?head = p v
  using connector-finish[OF puUV] subpath-uv-finish
  by (simp add: segment-loop-def connector-start connector-finish)
have head-cv-finish: pathfinish ?head = pathfinish ?cv
  using head-finish-pv connector-finish[OF pvUV] by simp
have tail-cv-start: pathstart ?tail = pathfinish ?cv
  using connector-finish[OF pvUV] svw-start by simp
have step3-raw:
  homotopic-paths W (((?head +++ reversepath ?cv) +++ ?cv) +++ ?tail)
  (?head +++ ?tail)
  by (rule homotopic-paths-cancel-middle[OF head-path head-img cv-path cv-img
tail-path tail-img head-cv-finish tail-cv-start])
have step3:
  homotopic-paths W ((segment-loop p u v +++ ?cv) +++ ?tail) (?head +++
?tail)
  unfolding segment-loop-def using step3-raw by simp

```

**have** *svw-finish*: *pathfinish*  $?svw = p v$   
**by** (*simp add*: *subpathin-def pathfinish-def*)  
**have** *svw-start-eq*: *pathstart*  $?svw = p v$   
**by** (*simp add*: *subpathin-def pathstart-def*)  
**have** *svw-finish*: *pathfinish*  $?svw = p w$   
**by** (*simp add*: *subpathin-def pathfinish-def*)  
**have** *head-svw-start*: *pathfinish*  $?head = pathstart ?svw$   
**using** *connector-finish*[*OF puUV*] *svw-finish svw-start-eq*  
**by** (*simp add*: *connector-start connector-finish*)  
**have** *svw-revcw-start*: *pathfinish*  $?svw = pathstart (reversepath ?cw)$   
**using** *connector-finish*[*OF pwUV*] *svw-finish* **by** *simp*  
**have** *step4a*:  
*homotopic-paths*  $W (?head +++ ?tail) (((?cu +++ ?svw) +++ ?svw) +++$   
*reversepath ?cw)*  
**by** (*rule homotopic-paths-assoc*[*OF head-path head-img svw-path svw-img*  
*rev-cw-path rev-cw-img head-svw-start svw-revcw-start*])  
**have** *step4b-inner*:  
*homotopic-paths*  $W (((?cu +++ ?svw) +++ ?svw)) (?cu +++ (?svw +++$   
 $?svw))$   
**proof** –  
**have** *cu-svw-start*: *pathfinish*  $?cu = pathstart ?svw$   
**using** *connector-finish*[*OF puUV*] **by** (*simp add*: *subpathin-def pathstart-def*)  
**have** *svw-svw-start*: *pathfinish*  $?svw = pathstart ?svw$   
**by** (*simp add*: *subpathin-def pathstart-def pathfinish-def*)  
**have** *homotopic-paths*  $W (?cu +++ (?svw +++ ?svw)) (((?cu +++ ?svw)$   
 $+++ ?svw))$   
**by** (*rule homotopic-paths-assoc*[*OF cu-path cu-img svw-path svw-img svw-path*  
*svw-img cu-svw-start svw-svw-start*])  
**then show** *?thesis*  
**by** (*rule homotopic-paths-sym*)  
**qed**  
**have** *step4b*:  
*homotopic-paths*  $W (((?cu +++ ?svw) +++ ?svw) +++ reversepath ?cw))$   
 $((?cu +++ (?svw +++ ?svw)) +++ reversepath ?cw))$   
**proof** –  
**have** *cusuv-svw-finish*: *pathfinish*  $((?cu +++ ?svw) +++ ?svw) = p w$   
**using** *connector-finish*[*OF puUV*] *svw-finish svw-finish*  
**by** (*simp add*: *connector-start connector-finish*)  
**have** *cusuv-svw-revcw-start*: *pathfinish*  $((?cu +++ ?svw) +++ ?svw) = path-$   
 $start (reversepath ?cw)$   
**using** *cusuv-svw-finish connector-finish*[*OF pwUV*] **by** *simp*  
**show** *?thesis*  
**by** (*rule homotopic-paths-join-right*[*OF step4b-inner rev-cw-path rev-cw-img*  
*cusuv-svw-revcw-start*])  
**qed**  
**have** *step4*:  
*homotopic-paths*  $W (?head +++ ?tail) (((?cu +++ (?svw +++ ?svw)) +++$   
 $reversepath ?cw))$

```

by (rule homotopic-paths-trans[OF step4a step4b])

have step5-inner0: homotopic-paths W (?suv +++ ?svw) ?suv
  by (rule homotopic-paths-join-subpathin[OF p-path p-img u01 v01 w01 uv vw])
have step5-inner1:
  homotopic-paths W (?cu +++ (?suv +++ ?svw)) (?cu +++ ?suv)
proof -
  have cu-svsvw-start: pathfinish ?cu = pathstart (?suv +++ ?svw)
  proof -
    have cu-finish: pathfinish ?cu = p u
      using connector-finish[OF puUV] by simp
    have svsvw-start: pathstart (?suv +++ ?svw) = p u
      by (simp add: joinpaths-def subpathin-def pathstart-def)
    show ?thesis
      using cu-finish svsvw-start by simp
  qed
  show ?thesis
    by (rule homotopic-paths-join-left[OF step5-inner0 cu-path cu-img
cu-svsvw-start])
  qed
  have step5:
    homotopic-paths W (((?cu +++ (?suv +++ ?svw)) +++ reversepath ?cw))
(segment-loop p u w)
  proof -
    have cu-svsvw-finish: pathfinish (?cu +++ (?suv +++ ?svw)) = p w
      using connector-finish[OF puUV] svv-finish svw-finish
      by (simp add: connector-start connector-finish)
    have cu-svsvw-revcw-start: pathfinish (?cu +++ (?suv +++ ?svw)) = pathstart
(reversepath ?cw)
      using cu-svsvw-finish connector-finish[OF pwUV] by simp
    have step5-raw:
      homotopic-paths W (((?cu +++ (?suv +++ ?svw)) +++ reversepath ?cw))
      (((?cu +++ ?suv) +++ reversepath ?cw))
    proof (rule homotopic-paths-join-right[OF step5-inner1 rev-cw-path
rev-cw-img])
      show pathfinish (?cu +++ (?suv +++ ?svw)) = pathstart (reversepath ?cw)
        by (rule cu-svsvw-revcw-start)
    qed
    have step5-assoc:
      homotopic-paths W (((?cu +++ ?suv) +++ reversepath ?cw))
      (?cu +++ (?suv +++ reversepath ?cw))
    proof -
      have cu-suv-start: pathfinish ?cu = pathstart ?suv
        using connector-finish[OF puUV] by (simp add: subpathin-def pathstart-def)
      have suw-revcw-start: pathfinish ?suv = pathstart (reversepath ?cw)
        using connector-finish[OF pwUV] by (simp add: subpathin-def pathfinish-def)
      have homotopic-paths W (?cu +++ (?suv +++ reversepath ?cw))
        (((?cu +++ ?suv) +++ reversepath ?cw))
        by (rule homotopic-paths-assoc[OF cu-path cu-img suw-path suw-img

```

```

rev-cw-path rev-cw-img cu-suw-start suw-revcw-start])
  then show ?thesis
    by (rule homotopic-paths-sym)
  qed
  have step5-assoc':
    homotopic-paths W (((?cu +++ ?suw) +++ reversepath ?cw))
      (connector (p u) +++ (subpathin u w p +++ reversepath (connector (p
w))))))
    using step5-assoc by simp
  have step5-final:
    homotopic-paths W (((?cu +++ (?suw +++ ?svw)) +++ reversepath ?cw))
      (connector (p u) +++ (subpathin u w p +++ reversepath (connector (p
w))))))
    by (rule homotopic-paths-trans[OF step5-raw step5-assoc'])
  show ?thesis
    unfolding segment-loop-def
    by (rule step5-raw)
  qed

  have homotopic-paths W (segment-loop p u v +++ segment-loop p v w)
    ((segment-loop p u v +++ ?cv) +++ ?tail)
    by (rule homotopic-paths-trans[OF step1 step2])
  then have homotopic-paths W (segment-loop p u v +++ segment-loop p v w)
    (?head +++ ?tail)
    by (rule homotopic-paths-trans[OF - step3])
  then have homotopic-paths W (segment-loop p u v +++ segment-loop p v w)
    (((?cu +++ (?suw +++ ?svw)) +++ reversepath ?cw))
    by (rule homotopic-paths-trans[OF - step4])
  then show ?thesis
    by (rule homotopic-paths-trans[OF - step5])
  qed

lemma segment-loop-join-in-set:
  assumes p-path: path p
    and p-imgS: path-image p  $\subseteq$  S
    and SW: S  $\subseteq$  W
    and UVS: U  $\cap$  V  $\subseteq$  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
    and puUV: p u  $\in$  U  $\cap$  V
    and pvUV: p v  $\in$  U  $\cap$  V
    and pwUV: p w  $\in$  U  $\cap$  V
  shows homotopic-paths S (segment-loop p u v +++ segment-loop p v w)
(segment-loop p u w)
proof -
  let ?cu = connector (p u)

```

```

let ?cv = connector (p v)
let ?cw = connector (p w)
let ?sw = subpathin u v p
let ?svw = subpathin v w p
let ?suw = subpathin u w p
let ?head = ?cu +++ ?suw
let ?tail = ?svw +++ reversepath ?cw

have p-imgW: path-image p  $\subseteq$  W
  using p-imgS SW by blast

have cu-path: path ?cu
  by (rule connector-path[OF puUV])
have cv-path: path ?cv
  by (rule connector-path[OF pvUV])
have cw-path: path ?cw
  by (rule connector-path[OF pwUV])
have cu-img: path-image ?cu  $\subseteq$  S
  using connector-image-subset[OF puUV] UVS by blast
have cv-img: path-image ?cv  $\subseteq$  S
  using connector-image-subset[OF pvUV] UVS by blast
have cw-img: path-image ?cw  $\subseteq$  S
  using connector-image-subset[OF pwUV] UVS by blast

have suv-path: path ?suv
  by (rule path-subpathin[OF p-path u01 v01])
have svw-path: path ?svw
  by (rule path-subpathin[OF p-path v01 w01])
have suw-path: path ?suw
  by (rule path-subpathin[OF p-path u01 w01])
have suv-img: path-image ?suv  $\subseteq$  S
  using p-imgS path-image-subpathin-subset[OF u01 v01, of p] by blast
have svw-img: path-image ?svw  $\subseteq$  S
  using p-imgS path-image-subpathin-subset[OF v01 w01, of p] by blast
have suw-img: path-image ?suw  $\subseteq$  S
  using p-imgS path-image-subpathin-subset[OF u01 w01, of p] by blast
have suv-start: pathstart ?suv = p u
  by (simp add: pathstart-def subpathin-def)
have svw-start: pathstart ?svw = p v
  by (simp add: pathstart-def subpathin-def)
have suw-finish: pathfinish ?suw = p w
  by (simp add: pathfinish-def subpathin-def)
have rev-cw-path: path (reversepath ?cw)
  using cw-path by simp
have rev-cw-img: path-image (reversepath ?cw)  $\subseteq$  S
  using cw-img by simp
have rev-cw-start: pathstart (reversepath ?cw) = p w
  using connector-finish[OF pwUV] by simp

```

```

have head-path: path ?head
  using cu-path svw-path connector-finish[OF puUV] svw-start by simp
have head-img: path-image ?head  $\subseteq S$ 
  by (rule subset-path-image-join[OF cu-img svw-img])
have tail-path: path ?tail
  using svw-path rev-cw-path svw-finish rev-cw-start by simp
have tail-img: path-image ?tail  $\subseteq S$ 
  by (rule subset-path-image-join[OF svw-img rev-cw-img])

have seg-uv-loop: segment-loop p u v  $\in$  loop-space S x0
proof (rule segment-loop-in-set[where S = S])
  show path p
    by (rule p-path)
  show path-image p  $\subseteq W$ 
    by (rule p-imgW)
  show u  $\in$  {0..1} v  $\in$  {0..1}
    by (rule u01, rule v01)+
  show p u  $\in U \cap V$  p v  $\in U \cap V$ 
    by (rule puUV, rule pvUV)+
  show path-image (connector (p u))  $\subseteq S$ 
    by (rule cu-img)
  show path-image (connector (p v))  $\subseteq S$ 
    by (rule cv-img)
  show subpathin u v p ‘ {0..1}  $\subseteq S$ 
    using svw-img by (simp add: path-image-def)
  show x0  $\in S$ 
    using x0-in-UV UVS by blast
qed
then have seg-uv-path: path (segment-loop p u v)
  and seg-uv-img: path-image (segment-loop p u v)  $\subseteq S$ 
  unfolding loop-space-def by auto

have seg-vw-assoc:
  homotopic-paths S (segment-loop p v w) (?cv +++ ?tail)
proof –
  have homotopic-paths S (?cv +++ (?svw +++ reversepath ?cw)) ((?cv +++
  ?svw) +++ reversepath ?cw)
    by (rule homotopic-paths-assoc[OF cv-path cv-img svw-path svw-img
  rev-cw-path rev-cw-img])
    (use connector-finish[OF pvUV] svw-start svw-finish rev-cw-start in simp-all)
  then show ?thesis
    unfolding segment-loop-def by (rule homotopic-paths-sym)
qed

have seg-join-start: pathfinish (segment-loop p u v) = pathstart (segment-loop p
  v w)
  using pvUV by (simp add: segment-loop-def connector-start connector-finish)
have step1:
  homotopic-paths S (segment-loop p u v +++ segment-loop p v w) (segment-loop

```

$p u v +++ (?cv +++ ?tail)$   
**by** (rule *homotopic-paths-join-left*[*OF seg-vw-assoc seg-uv-path seg-uv-img seg-join-start*])

**have** *seg-cv-start*: *pathfinish* (segment-loop  $p u v$ ) = *pathstart*  $?cv$   
**using** *pvUV* **by** (*simp add*: *segment-loop-def connector-start connector-finish*)  
**have** *cv-tail-start*: *pathfinish*  $?cv$  = *pathstart*  $?tail$   
**using** *connector-finish*[*OF pvUV*] *svw-start* **by** *simp*  
**have** *step2*:  
*homotopic-paths*  $S$  (segment-loop  $p u v +++ (?cv +++ ?tail)$ ) ((segment-loop  $p u v +++ ?cv$ )  $+++ ?tail$ )  
**by** (rule *homotopic-paths-assoc*[*OF seg-uv-path seg-uv-img cv-path cv-img tail-path tail-img seg-cv-start cv-tail-start*])

**have** *subpath-uv-finish*: *pathfinish* (subpathin  $u v p$ ) =  $p v$   
**by** (*simp add*: *subpathin-def pathfinish-def*)  
**have** *head-finish-pv*: *pathfinish*  $?head$  =  $p v$   
**using** *connector-finish*[*OF puUV*] *subpath-uv-finish*  
**by** (*simp add*: *segment-loop-def connector-start connector-finish*)  
**have** *head-cv-finish*: *pathfinish*  $?head$  = *pathfinish*  $?cv$   
**using** *head-finish-pv* *connector-finish*[*OF pvUV*] **by** *simp*  
**have** *tail-cv-start*: *pathstart*  $?tail$  = *pathfinish*  $?cv$   
**using** *connector-finish*[*OF pvUV*] *svw-start* **by** *simp*  
**have** *step3-raw*:  
*homotopic-paths*  $S$  ((( $?head +++ reversepath ?cv$ )  $+++ ?cv$ )  $+++ ?tail$ )  
( $?head +++ ?tail$ )  
**by** (rule *homotopic-paths-cancel-middle*[*OF head-path head-img cv-path cv-img tail-path tail-img head-cv-finish tail-cv-start*])  
**have** *step3*:  
*homotopic-paths*  $S$  ((segment-loop  $p u v +++ ?cv$ )  $+++ ?tail$ ) ( $?head +++ ?tail$ )  
**unfolding** *segment-loop-def* **using** *step3-raw* **by** *simp*

**have** *svw-finish*: *pathfinish*  $?svw$  =  $p v$   
**by** (*simp add*: *subpathin-def pathfinish-def*)  
**have** *svw-start-eq*: *pathstart*  $?svw$  =  $p v$   
**by** (*simp add*: *subpathin-def pathstart-def*)  
**have** *svw-finish*: *pathfinish*  $?svw$  =  $p w$   
**by** (*simp add*: *subpathin-def pathfinish-def*)  
**have** *head-svw-start*: *pathfinish*  $?head$  = *pathstart*  $?svw$   
**using** *connector-finish*[*OF puUV*] *svw-finish* *svw-start-eq*  
**by** (*simp add*: *connector-start connector-finish*)  
**have** *svw-revcw-start*: *pathfinish*  $?svw$  = *pathstart* (reversepath  $?cw$ )  
**using** *connector-finish*[*OF pwUV*] *svw-finish* **by** *simp*  
**have** *step4a*:  
*homotopic-paths*  $S$  ( $?head +++ ?tail$ ) ((( $?cu +++ ?svw$ )  $+++ ?svw$ )  $+++ reversepath ?cw$ )  
**by** (rule *homotopic-paths-assoc*[*OF head-path head-img svw-path svw-img rev-cw-path rev-cw-img head-svw-start svw-revcw-start*])

```

have step4b-inner:
  homotopic-paths  $S$  ((( $?cu$  +++  $?suv$ ) +++  $?svw$ ) ( $?cu$  +++ ( $?suv$  +++
 $?svw$ ))
proof –
  have cu-suv-start: pathfinish  $?cu = \text{pathstart } ?suv$ 
  using connector-finish[ $OF$   $puUV$ ] by (simp add: subpathin-def pathstart-def)
  have suv-svw-start: pathfinish  $?suv = \text{pathstart } ?svw$ 
  by (simp add: subpathin-def pathstart-def pathfinish-def)
  have homotopic-paths  $S$  ( $?cu$  +++ ( $?suv$  +++  $?svw$ )) ((( $?cu$  +++  $?suv$ ) +++
 $?svw$ ))
  by (rule homotopic-paths-assoc[ $OF$  cu-path cu-img suv-path suv-img svw-path
svw-img cu-suv-start suv-svw-start])
  then show ?thesis
  by (rule homotopic-paths-sym)
qed
have step4b:
  homotopic-paths  $S$  (((( $?cu$  +++  $?suv$ ) +++  $?svw$ ) +++ reversepath  $?cw$ )
  ((( $?cu$  +++ ( $?suv$  +++  $?svw$ )) +++ reversepath  $?cw$ ))
proof –
  have cusuv-svw-finish: pathfinish ((( $?cu$  +++  $?suv$ ) +++  $?svw$ ) =  $p$   $w$ 
  using connector-finish[ $OF$   $puUV$ ] svw-finish svw-finish
  by (simp add: connector-start connector-finish)
  have cusuv-svw-revcw-start: pathfinish ((( $?cu$  +++  $?suv$ ) +++  $?svw$ ) = path-
start (reversepath  $?cw$ )
  using cusuv-svw-finish connector-finish[ $OF$   $pwUV$ ] by simp
  show ?thesis
  by (rule homotopic-paths-join-right[ $OF$  step4b-inner rev-cw-path rev-cw-img
cusuv-svw-revcw-start])
qed
have step4:
  homotopic-paths  $S$  ( $?head$  +++  $?tail$ ) ((( $?cu$  +++ ( $?suv$  +++  $?svw$ )) +++
reversepath  $?cw$ ))
  by (rule homotopic-paths-trans[ $OF$  step4a step4b])

have step5-inner0: homotopic-paths  $S$  ( $?suv$  +++  $?svw$ )  $?suv$ 
by (rule homotopic-paths-join-subpathin[ $OF$  p-path p-imgS u01 v01 w01 uv vw])
have step5-inner1:
  homotopic-paths  $S$  ( $?cu$  +++ ( $?suv$  +++  $?svw$ )) ( $?cu$  +++  $?suv$ )
proof –
  have cu-svsvw-start: pathfinish  $?cu = \text{pathstart } (?suv$  +++  $?svw)$ 
proof –
  have cu-finish: pathfinish  $?cu = p$   $u$ 
  using connector-finish[ $OF$   $puUV$ ] by simp
  have svsvw-start: pathstart ( $?suv$  +++  $?svw$ ) =  $p$   $u$ 
  by (simp add: joinpaths-def subpathin-def pathstart-def)
  show ?thesis
  using cu-finish svsvw-start by simp
qed
show ?thesis

```

```

      by (rule homotopic-paths-join-left[OF step5-inner0 cu-path cu-img
cu-suvsw-start])
    qed
    have step5:
      homotopic-paths S (((?cu +++ (?suv +++ ?svw)) +++ reversepath ?cw))
(segment-loop p u w)
    proof -
      have cu-suvw-finish: pathfinish (?cu +++ (?suv +++ ?svw)) = p w
      using connector-finish[OF puUV] suv-finish svw-finish
      by (simp add: connector-start connector-finish)
      have cu-suvw-revcw-start: pathfinish (?cu +++ (?suv +++ ?svw)) = pathstart
(reversepath ?cw)
      using cu-suvw-finish connector-finish[OF pwUV] by simp
      have step5-raw:
        homotopic-paths S (((?cu +++ (?suv +++ ?svw)) +++ reversepath ?cw))
        (((?cu +++ ?suv) +++ reversepath ?cw))
      proof (rule homotopic-paths-join-right[OF step5-inner1 rev-cw-path
rev-cw-img])
        show pathfinish (?cu +++ (?suv +++ ?svw)) = pathstart (reversepath ?cw)
        by (rule cu-suvw-revcw-start)
      qed
      have step5-assoc:
        homotopic-paths S (((?cu +++ ?suv) +++ reversepath ?cw))
        (?cu +++ (?suv +++ reversepath ?cw))
      proof -
        have cu-suv-start: pathfinish ?cu = pathstart ?suv
        using connector-finish[OF puUV] by (simp add: subpathin-def pathstart-def)
        have suw-revcw-start: pathfinish ?suv = pathstart (reversepath ?cw)
        using connector-finish[OF pwUV] by (simp add: subpathin-def pathfinish-def)
        have homotopic-paths S (?cu +++ (?suv +++ reversepath ?cw))
        (((?cu +++ ?suv) +++ reversepath ?cw))
        by (rule homotopic-paths-assoc[OF cu-path cu-img suw-path suw-img
rev-cw-path rev-cw-img cu-suv-start suw-revcw-start])
        then show ?thesis
        by (rule homotopic-paths-sym)
      qed
      have step5-assoc':
        homotopic-paths S (((?cu +++ ?suv) +++ reversepath ?cw))
        (connector (p u) +++ (subpathin u w p +++ reversepath (connector (p
w))))
      using step5-assoc by simp
      have step5-final:
        homotopic-paths S (((?cu +++ (?suv +++ ?svw)) +++ reversepath ?cw))
        (connector (p u) +++ (subpathin u w p +++ reversepath (connector (p
w))))
      by (rule homotopic-paths-trans[OF step5-raw step5-assoc'])
      show ?thesis
      unfolding segment-loop-def
      by (rule step5-raw)

```

**qed**

```
have homotopic-paths  $S$  (segment-loop  $p$   $u$   $v$  +++ segment-loop  $p$   $v$   $w$ )
((segment-loop  $p$   $u$   $v$  +++ ? $cv$ ) +++ ? $tail$ )
by (rule homotopic-paths-trans[OF step1 step2])
then have homotopic-paths  $S$  (segment-loop  $p$   $u$   $v$  +++ segment-loop  $p$   $v$   $w$ )
(? $head$  +++ ? $tail$ )
by (rule homotopic-paths-trans[OF - step3])
then have homotopic-paths  $S$  (segment-loop  $p$   $u$   $v$  +++ segment-loop  $p$   $v$   $w$ )
(((? $cu$  +++ (? $sv$  +++ ? $sw$ )) +++ reversepath ? $cw$ )
by (rule homotopic-paths-trans[OF - step4])
then show ? $thesis$ 
by (rule homotopic-paths-trans[OF - step5])
```

**qed**

**lemma** partition-loop-nil [simp]:

```
partition-loop  $p$  [] = ( $\lambda$ -.  $x0$ )
by simp
```

**lemma** partition-loop-singleton [simp]:

```
partition-loop  $p$  [ $t$ ] = ( $\lambda$ -.  $x0$ )
by simp
```

**lemma** svk-partition-partition-loop-homotopic-segment-loop:

```
assumes  $p$ -loop:  $p \in$  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)
using  $part$ 
proof (induction  $ts$  arbitrary:  $t$   $bs$ )
case Nil
have  $t1$ :  $t = 1$ 
using svk-partition-last-eq-one[OF Nil.prem] by simp
have  $ptUV$ :  $p$   $t \in U \cap V$ 
by (rule svk-partition-head-props(2)[OF Nil.prem])
have homotopic-paths  $W$  (segment-loop  $p$   $t$  1) ( $\lambda$ -.  $x0$ )
using  $t1$   $ptUV$  by (simp add: segment-loop-refl)
then show ? $case$ 
by (simp add: homotopic-paths-sym)
```

**next**

```
case (Cons  $u$   $us$ )
from  $p$ -loop have  $p$ -path: path  $p$  and  $p$ -img: path-image  $p \subseteq W$ 
unfolding loop-space-def by auto
obtain  $b$   $bs'$  where  $bs$ :  $bs = b \#$   $bs'$ 
using Cons.prem by (cases  $bs$ ) auto
have  $tail$ : svk-partition  $p$  ( $u \#$   $us$ )  $bs'$ 
using Cons.prem  $bs$  by simp
have  $tail$ -hom:
homotopic-paths  $W$  (partition-loop  $p$  ( $u \#$   $us$ )) (segment-loop  $p$   $u$  1)
by (rule Cons.IH[OF  $tail$ ])
```

```

have t01:  $t \in \{0..1\}$  and ptUV:  $p t \in U \cap V$ 
  using Cons.premis bs by simp-all
have u01:  $u \in \{0..1\}$  and tu:  $t < u$ 
  using Cons.premis bs by simp-all
have puUV:  $p u \in U \cap V$ 
  using Cons.premis bs svk-partition-next-in-intersection[of p t u us b bs'] by simp
have seg-in:
  subpathin t u p '  $\{0..1\} \subseteq W$ 
  by (cases b) (use Cons.premis bs in auto)
have oneUV:  $p 1 \in U \cap V$ 
  using svk-partition-last-in-intersection[OF tail] svk-partition-last-eq-one[OF
tail] by simp
have seg-tu-loop: segment-loop p t u  $\in$  loop-space W x0
  by (rule segment-loop-in-W[OF p-path p-img t01 u01 ptUV puUV seg-in])
then have seg-tu-path: path (segment-loop p t u)
  and seg-tu-img: path-image (segment-loop p t u)  $\subseteq$  W
  unfolding loop-space-def by auto
have tail-loop: partition-loop p (u # us)  $\in$  loop-space W x0
  by (rule svk-partition-partition-loop-in-W[OF p-loop tail])
have join-hom:
  homotopic-paths W (segment-loop p t u +++ partition-loop p (u # us))
  (segment-loop p t u +++ segment-loop p u 1)
proof (rule homotopic-paths-join-left[OF tail-hom seg-tu-path seg-tu-img])
  show pathfinish (segment-loop p t u) = pathstart (partition-loop p (u # us))
    using seg-tu-loop tail-loop unfolding loop-space-def by simp
qed
have step1:
  homotopic-paths W (partition-loop p (t # u # us)) (segment-loop p t u +++
segment-loop p u 1)
  using join-hom by simp
have step2:
  homotopic-paths W (segment-loop p t u +++ segment-loop p u 1) (segment-loop
p t 1)
  by (rule segment-loop-join[OF p-path p-img t01 u01]) (use u01 tu oneUV ptUV
puUV in auto)
  show ?case
  by (rule homotopic-paths-trans[OF step1 step2])
qed

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 -
  obtain t ts' where ts:  $ts = t \# ts'$ 
  using valid-partition-hd(1)[OF part] by (cases ts) auto
  have valid-ts: valid-partition p (t # ts') bs
  using part unfolding ts by simp
  have t0:  $t = 0$ 

```

by (rule valid-partition-cases(1)[OF valid-ts])  
 have svk: svk-partition p (t # ts<sup>^</sup>) bs  
 by (rule valid-partition-cases(2)[OF valid-ts])  
 have homotopic-paths W (partition-loop p (t # ts<sup>^</sup>)) (segment-loop p t 1)  
 by (rule svk-partition-partition-loop-homotopic-segment-loop[OF p-loop svk])  
 then show ?thesis  
 using ts t0 by simp  
 qed

**lemma** valid-partition-partition-loop-homotopic:  
 assumes p-loop: p ∈ loop-space W x0  
 and part: valid-partition p ts bs  
 shows homotopic-paths W (partition-loop p ts) p  
**proof** –  
 have step1: homotopic-paths W (partition-loop p ts) (segment-loop p 0 1)  
 by (rule valid-partition-partition-loop-homotopic-segment-loop[OF p-loop part])  
 have step2: homotopic-paths W (segment-loop p 0 1) p  
 by (rule segment-loop-full[OF p-loop])  
 show ?thesis  
 by (rule homotopic-paths-trans[OF step1 step2])  
 qed

**lemma** valid-partition-partition-loop-eq:  
 assumes p-loop: p ∈ 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** –  
 have part-loop: partition-loop p ts ∈ loop-space W x0  
 by (rule valid-partition-partition-loop-in-W[OF p-loop part])  
 have hom: homotopic-paths W (partition-loop p ts) p  
 by (rule valid-partition-partition-loop-homotopic[OF p-loop part])  
 show ?thesis  
 by (rule loop-class-eqI[OF part-loop p-loop hom])  
 qed

**lemma** valid-partition-decode-partition-word-eq-loop-class:  
 assumes p-loop: p ∈ loop-space W x0  
 and part: valid-partition p ts bs  
 shows svk-decode (partition-word p ts bs) = loop-class W x0 p  
 using valid-partition-decode-partition-word[OF p-loop part]  
 valid-partition-partition-loop-eq[OF p-loop part]  
 by simp

**lemma** subpathin-image-subset-left:  
 assumes t01: t ∈ {0..1}  
 and u01: u ∈ {0..1}  
 and v01: v ∈ {0..1}  
 and tu: t ≤ u  
 and uv: u ≤ v

**shows**  $\text{subpathin } t \ u \ p \ ' \ {0..1} \subseteq \text{subpathin } t \ v \ p \ ' \ {0..1}$   
**proof** –  
**have**  $\text{closed-segment } t \ u \subseteq \text{closed-segment } t \ v$   
**using**  $t01 \ u01 \ v01 \ tu \ uv$  **by** (auto simp: closed-segment-eq-real-ivl)  
**then show** ?thesis  
**by** (auto simp: subpathin-image)  
**qed**

**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**  $\text{subpathin } u \ v \ p \ ' \ {0..1} \subseteq \text{subpathin } t \ v \ p \ ' \ {0..1}$   
**proof** –  
**have**  $\text{closed-segment } u \ v \subseteq \text{closed-segment } t \ v$   
**using**  $t01 \ u01 \ v01 \ tu \ uv$  **by** (auto simp: closed-segment-eq-real-ivl)  
**then show** ?thesis  
**by** (auto simp: subpathin-image)  
**qed**

**lemma** *subpathin-subpathin*:  
 $\text{subpathin } a \ b \ (\text{subpathin } u \ v \ p) =$   
 $\text{subpathin } (((v - u) * a) + u) \ (((v - u) * b) + u) \ p$   
**unfolding** *subpathin-def* **by** (rule ext) (simp add: algebra-simps)

**lemma** *segment-loop-subpathin*:  
 $\text{segment-loop } (\text{subpathin } u \ v \ p) \ a \ b =$   
 $\text{segment-loop } p \ (((v - u) * a) + u) \ (((v - u) * b) + u)$   
**unfolding** *segment-loop-def subpathin-subpathin subpathin-def*  
**by** (rule ext) (simp add: algebra-simps)

**lemma** *segment-loop-mult-eq-left*:  
**assumes**  $p\text{-path}: \text{path } p$   
**and**  $p\text{-img}W: \text{path-image } p \subseteq W$   
**and**  $t01: t \in \{0..1\}$   
**and**  $u01: u \in \{0..1\}$   
**and**  $v01: v \in \{0..1\}$   
**and**  $tu: t < u$   
**and**  $uv: u < v$   
**and**  $ptUV: p \ t \in U \cap V$   
**and**  $puUV: p \ u \in U \cap V$   
**and**  $pvUV: p \ v \in U \cap V$   
**and**  $\text{seg-tv}U: \text{subpathin } t \ v \ p \ ' \ {0..1} \subseteq U$   
**shows**  $\text{mult1 } (\text{loop-class } U \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } U \ x0 \ (\text{segment-loop } p \ u \ v)) =$   
 $\text{loop-class } U \ x0 \ (\text{segment-loop } p \ t \ v)$   
**proof** –

```

define q where  $q = \text{subpathin } t \ v \ p$ 
define a where  $a = (u - t) / (v - t)$ 

have tv:  $t < v$ 
  using tu uv by linarith
have a01:  $a \in \{0..1\}$ 
  unfolding a-def using tu uv tv by (auto simp: field-simps)
have q-path: path q
  unfolding q-def by (rule path-subpathin[OF p-path t01 v01])
have q-imgU: path-image q  $\subseteq U$ 
  unfolding q-def using seg-tvU by (simp add: path-image-def)
have q0UV:  $q \ 0 \in U \cap V$ 
  unfolding q-def using ptUV by (simp add: subpathin-def)
have qaUV:  $q \ a \in U \cap V$ 
proof -
  have qa-eq:  $q \ a = p \ u$ 
  proof -
    have ta-eq:  $t + (v - t) * a = u$ 
      unfolding a-def using tv by (simp add: field-simps)
    show ?thesis
      unfolding q-def subpathin-def using ta-eq by (simp add: algebra-simps)
  qed
  then show ?thesis
    using puUV by simp
qed
have q1UV:  $q \ 1 \in U \cap V$ 
  unfolding q-def using pvUV by (simp add: subpathin-def)
have join-hom-q:
  homotopic-paths U (segment-loop q 0 a +++ segment-loop q a 1) (segment-loop
q 0 1)
  by (rule segment-loop-join-in-set[OF q-path q-imgU]) (use a01 q0UV qaUV
q1UV in auto)

have seg-tu-eq: segment-loop q 0 a = segment-loop p t u
proof -
  have ta-eq:  $((v - t) * a) + t = u$ 
    unfolding a-def using tv by (simp add: field-simps algebra-simps)
  show ?thesis
    unfolding q-def using ta-eq by (simp add: segment-loop-subpathin)
qed
have seg-uv-eq: segment-loop q a 1 = segment-loop p u v
proof -
  have ta-eq:  $((v - t) * a) + t = u$ 
    unfolding a-def using tv by (simp add: field-simps algebra-simps)
  show ?thesis
    unfolding q-def using ta-eq by (simp add: segment-loop-subpathin)
qed
have seg-tv-eq: segment-loop q 0 1 = segment-loop p t v
  unfolding q-def by (simp add: segment-loop-subpathin)

```

**have**  $seg\text{-}tuU$ :  $subpathin\ t\ u\ p\ \{0..1\} \subseteq U$   
**by** (rule order-trans[OF subpathin-image-subset-left[OF t01 u01 v01]]) (use tu uv seg-tvU **in** auto)  
**have**  $seg\text{-}uvU$ :  $subpathin\ u\ v\ p\ \{0..1\} \subseteq U$   
**by** (rule order-trans[OF subpathin-image-subset-right[OF t01 u01 v01]]) (use tu uv seg-tvU **in** auto)

**have**  $loop\text{-}tu$ :  $segment\text{-}loop\ p\ t\ u \in loop\text{-}space\ U\ x0$   
**by** (rule segment-loop-in-U[OF p-path p-imgW t01 u01 ptUV puUV seg-tuU])  
**have**  $loop\text{-}uv$ :  $segment\text{-}loop\ p\ u\ v \in loop\text{-}space\ U\ x0$   
**by** (rule segment-loop-in-U[OF p-path p-imgW u01 v01 puUV pvUV seg-uvU])  
**have**  $loop\text{-}tv$ :  $segment\text{-}loop\ p\ t\ v \in loop\text{-}space\ U\ x0$   
**by** (rule segment-loop-in-U[OF p-path p-imgW t01 v01 ptUV pvUV seg-tvU])

**have**  $class\text{-}tu\text{-}in$ :  $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ t\ u) \in G1$   
**by** (rule loop-class-in-space[OF loop-tu])  
**have**  $class\text{-}uv\text{-}in$ :  $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ u\ v) \in G1$   
**by** (rule loop-class-in-space[OF loop-uv])  
**have**  $join\text{-}loop$ :  $segment\text{-}loop\ p\ t\ u\ +++\ segment\text{-}loop\ p\ u\ v \in loop\text{-}space\ U\ x0$   
**by** (rule loop-space-join[OF loop-tu loop-uv])

**have**  $mult\text{-}eq\text{-}join$ :  
 $mult1\ (loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ t\ u))$   
 $(loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ u\ v)) =$   
 $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ t\ u\ +++\ segment\text{-}loop\ p\ u\ v)$   
**by** (rule fundamental-group-mult-eqI[OF class-tu-in class-uv-in loop-tu loop-uv])  
*simp-all*

**have**  $join\text{-}eq$ :  
 $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ t\ u\ +++\ segment\text{-}loop\ p\ u\ v) =$   
 $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ p\ t\ v)$

**proof** –  
**have**  $join\text{-}eq\text{-}q$ :  
 $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ q\ 0\ a\ +++\ segment\text{-}loop\ q\ a\ 1) =$   
 $loop\text{-}class\ U\ x0\ (segment\text{-}loop\ q\ 0\ 1)$

**proof** (rule loop-class-eqI)  
**show**  $segment\text{-}loop\ q\ 0\ a\ +++\ segment\text{-}loop\ q\ a\ 1 \in loop\text{-}space\ U\ x0$   
**unfolding**  $seg\text{-}tu\text{-}eq\ seg\text{-}uv\text{-}eq$  **by** (rule join-loop)  
**show**  $segment\text{-}loop\ q\ 0\ 1 \in loop\text{-}space\ U\ x0$   
**using**  $seg\text{-}tv\text{-}eq\ loop\text{-}tv$  **by** *simp*  
**show**  $homotopic\text{-}paths\ U\ (segment\text{-}loop\ q\ 0\ a\ +++\ segment\text{-}loop\ q\ a\ 1)$   
 $(segment\text{-}loop\ q\ 0\ 1)$   
**by** (rule join-hom-q)

**qed**  
**show** *?thesis*  
**using**  $join\text{-}eq\text{-}q$  **by** (*simp only: seg-tu-eq seg-uv-eq seg-tv-eq*)

**qed**  
**show** *?thesis*  
**using**  $mult\text{-}eq\text{-}join\ join\text{-}eq$  **by** *simp*

qed

**lemma** *segment-loop-mult-eq-right*:

**assumes** *p-path*: *path* *p*  
**and** *p-imgW*: *path-image* *p*  $\subseteq$  *W*  
**and** *t01*:  $t \in \{0..1\}$   
**and** *u01*:  $u \in \{0..1\}$   
**and** *v01*:  $v \in \{0..1\}$   
**and** *tu*:  $t < u$   
**and** *uv*:  $u < v$   
**and** *ptUV*:  $p\ t \in U \cap V$   
**and** *puUV*:  $p\ u \in U \cap V$   
**and** *pvUV*:  $p\ v \in U \cap V$   
**and** *seg-tvV*: *subpathin* *t v p*  $\{0..1\} \subseteq V$   
**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** –

**define** *q* **where**  $q = \text{subpathin } t\ v\ p$   
**define** *a* **where**  $a = (u - t) / (v - t)$

**have** *tv*:  $t < v$   
**using** *tu uv* **by** *linarith*  
**have** *a01*:  $a \in \{0..1\}$   
**unfolding** *a-def* **using** *tu uv tv* **by** (*auto simp: field-simps*)  
**have** *q-path*: *path* *q*  
**unfolding** *q-def* **by** (*rule path-subpathin[OF p-path t01 v01]*)  
**have** *q-imgV*: *path-image* *q*  $\subseteq V$   
**unfolding** *q-def* **using** *seg-tvV* **by** (*simp add: path-image-def*)  
**have** *q0UV*:  $q\ 0 \in U \cap V$   
**unfolding** *q-def* **using** *ptUV* **by** (*simp add: subpathin-def*)  
**have** *qaUV*:  $q\ a \in U \cap V$

**proof** –

**have** *qa-eq*:  $q\ a = p\ u$

**proof** –

**have** *ta-eq*:  $t + (v - t) * a = u$

**unfolding** *a-def* **using** *tv* **by** (*simp add: field-simps*)

**show** *?thesis*

**unfolding** *q-def subpathin-def* **using** *ta-eq* **by** (*simp add: algebra-simps*)

qed

**then show** *?thesis*

**using** *puUV* **by** *simp*

qed

**have** *q1UV*:  $q\ 1 \in U \cap V$

**unfolding** *q-def* **using** *pvUV* **by** (*simp add: subpathin-def*)

**have** *join-hom-q*:

*homotopic-paths* *V* (*segment-loop* *q 0 a* +++ *segment-loop* *q a 1*) (*segment-loop*  
*q 0 1*)

**by** (*rule segment-loop-join-in-set[OF q-path q-imgV]*) (*use a01 q0UV qaUV*)

$q1UV$  in auto)

```

have seg-tu-eq: segment-loop q 0 a = segment-loop p t u
proof -
  have ta-eq: ((v - t) * a) + t = u
    unfolding a-def using tv by (simp add: field-simps algebra-simps)
  show ?thesis
    unfolding q-def using ta-eq by (simp add: segment-loop-subpathin)
qed
have seg-uv-eq: segment-loop q a 1 = segment-loop p u v
proof -
  have ta-eq: ((v - t) * a) + t = u
    unfolding a-def using tv by (simp add: field-simps algebra-simps)
  show ?thesis
    unfolding q-def using ta-eq by (simp add: segment-loop-subpathin)
qed
have seg-tv-eq: segment-loop q 0 1 = segment-loop p t v
  unfolding q-def by (simp add: segment-loop-subpathin)

have seg-tuV: subpathin t u p ‘ {0..1}  $\subseteq$  V
  by (rule order-trans[OF subpathin-image-subset-left[OF t01 u01 v01]]) (use tu
uv seg-tuV in auto)
have seg-uvV: subpathin u v p ‘ {0..1}  $\subseteq$  V
  by (rule order-trans[OF subpathin-image-subset-right[OF t01 u01 v01]]) (use tu
uv seg-tvV in auto)

have loop-tu: segment-loop p t u  $\in$  loop-space V x0
  by (rule segment-loop-in-V[OF p-path p-imgW t01 u01 ptUV puUV seg-tuV])
have loop-uv: segment-loop p u v  $\in$  loop-space V x0
  by (rule segment-loop-in-V[OF p-path p-imgW u01 v01 puUV pvUV seg-uvV])
have loop-tv: segment-loop p t v  $\in$  loop-space V x0
  by (rule segment-loop-in-V[OF p-path p-imgW t01 v01 ptUV pvUV seg-tvV])

have class-tu-in: loop-class V x0 (segment-loop p t u)  $\in$  G2
  by (rule loop-class-in-space[OF loop-tu])
have class-uv-in: loop-class V x0 (segment-loop p u v)  $\in$  G2
  by (rule loop-class-in-space[OF loop-uv])
have join-loop: segment-loop p t u +++ segment-loop p u v  $\in$  loop-space V x0
  by (rule loop-space-join[OF loop-tu loop-uv])

have mult-eq-join:
  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 u +++ segment-loop p u v)
  by (rule fundamental-group-mult-eqI[OF class-tu-in class-uv-in loop-tu loop-uv])
simp-all
have join-eq:
  loop-class V x0 (segment-loop p t u +++ segment-loop p u v) =
  loop-class V x0 (segment-loop p t v)

```

```

proof –
  have join-eq-q:
    loop-class  $V$   $x0$  (segment-loop  $q$   $0$   $a$   $+++$  segment-loop  $q$   $a$   $1$ ) =
      loop-class  $V$   $x0$  (segment-loop  $q$   $0$   $1$ )
  proof (rule loop-class-eqI)
    show segment-loop  $q$   $0$   $a$   $+++$  segment-loop  $q$   $a$   $1$   $\in$  loop-space  $V$   $x0$ 
      unfolding seg-tu-eq seg-uv-eq by (rule join-loop)
    show segment-loop  $q$   $0$   $1$   $\in$  loop-space  $V$   $x0$ 
      using seg-tv-eq loop-tv by simp
      show homotopic-paths  $V$  (segment-loop  $q$   $0$   $a$   $+++$  segment-loop  $q$   $a$   $1$ )
        (segment-loop  $q$   $0$   $1$ )
      by (rule join-hom-q)
    qed
  show ?thesis
    using join-eq-q by (simp only: seg-tu-eq seg-uv-eq seg-tv-eq)
  qed
show ?thesis
  using mult-eq-join join-eq by simp
qed

```

**lemma** *segment-word-split-left-equiv*:

```

assumes p-loop:  $p \in$  loop-space  $W$   $x0$ 
  and t01:  $t \in \{0..1\}$ 
  and u01:  $u \in \{0..1\}$ 
  and v01:  $v \in \{0..1\}$ 
  and tu:  $t < u$ 
  and uv:  $u < v$ 
  and ptUV:  $p\ t \in U \cap V$ 
  and puUV:  $p\ u \in U \cap V$ 
  and pvUV:  $p\ v \in U \cap V$ 
  and seg-tvU: subpathin  $t\ v\ p\ \{0..1\} \subseteq U$ 
  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 –
  from p-loop have p-path: path  $p$  and p-imgW: path-image  $p \subseteq W$ 
  unfolding loop-space-def by auto
  have seg-tuU: subpathin  $t\ u\ p\ \{0..1\} \subseteq U$ 
    by (rule order-trans[OF subpathin-image-subset-left[OF t01 u01 v01]]) (use tu uv seg-tvU in auto)
  have seg-uvU: subpathin  $u\ v\ p\ \{0..1\} \subseteq U$ 
    by (rule order-trans[OF subpathin-image-subset-right[OF t01 u01 v01]]) (use tu uv seg-tvU in auto)
  have loop-tu: segment-loop  $p\ t\ u \in$  loop-space  $U$   $x0$ 
    by (rule segment-loop-in-U[OF p-path p-imgW t01 u01 ptUV puUV seg-tuU])
  have loop-uv: segment-loop  $p\ u\ v \in$  loop-space  $U$   $x0$ 
    by (rule segment-loop-in-U[OF p-path p-imgW u01 v01 puUV pvUV seg-uvU])

```

```

have class-tu-in: loop-class U x0 (segment-loop p t u) ∈ G1
  by (rule loop-class-in-space[OF loop-tu])
have class-uv-in: loop-class U x0 (segment-loop p u v) ∈ G1
  by (rule loop-class-in-space[OF loop-uv])
have mult-in:
  mult1 (loop-class U x0 (segment-loop p t u))
    (loop-class U x0 (segment-loop p u v)) ∈ G1
  by (rule fundamental-group-mult-in-space[OF class-tu-in class-uv-in])
have mult-eq:
  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)
  by (rule segment-loop-mult-eq-left[OF p-path p-imgW t01 u01 v01 tu uv ptUV
    puUV pvUV seg-tvU])
have step:
  carrier-fpw-reduction-step G1 G2 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 (mult1 (loop-class U x0 (segment-loop p t u))
      (loop-class U x0 (segment-loop p u v))) rest)
proof (rule carrier-fpw-reduction-step.combine-left)
  show loop-class U x0 (segment-loop p t u) ∈ G1
    by (rule class-tu-in)
  show loop-class U x0 (segment-loop p u v) ∈ G1
    by (rule class-uv-in)
  show mult1 (loop-class U x0 (segment-loop p t u)) (loop-class U x0 (segment-loop
    p u v)) ∈ G1
    by (rule mult-in)
  show fpw-in-space G1 G2 rest
    by (rule rest-in)
qed
have red:
  carrier-fpw-reduction G1 G2 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 (mult1 (loop-class U x0 (segment-loop p t u))
      (loop-class U x0 (segment-loop p u v))) rest)
  by (rule carrier-fpw-reduction.step[OF step])
have rel:
  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 (mult1 (loop-class U x0 (segment-loop p t u))
      (loop-class U x0 (segment-loop p u v))) rest)
  by (rule carrier-full-amalg-equiv.of-reduction[OF red])
show ?thesis
  using rel mult-eq by simp
qed

```

**lemma** *segment-word-split-right-equiv*:  
**assumes** *p-loop*:  $p \in \text{loop-space } W \ x0$   
**and** *t01*:  $t \in \{0..1\}$   
**and** *u01*:  $u \in \{0..1\}$   
**and** *v01*:  $v \in \{0..1\}$   
**and** *tu*:  $t < u$   
**and** *uv*:  $u < v$   
**and** *ptUV*:  $p \ t \in U \cap V$   
**and** *puUV*:  $p \ u \in U \cap V$   
**and** *pvUV*:  $p \ v \in U \cap V$   
**and** *seg-tvV*:  $\text{subpathin } t \ v \ p \ \{0..1\} \subseteq V$   
**and** *rest-in*:  $\text{fpw-in-space } G1 \ G2 \ \text{rest}$   
**shows** *carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{WordRight } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \ \text{rest}))$   
 $(\text{WordRight } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ v)) \ \text{rest}))$   
**proof** –  
**from** *p-loop* **have** *p-path*:  $\text{path } p$  **and** *p-imgW*:  $\text{path-image } p \subseteq W$   
**unfolding** *loop-space-def* **by** *auto*  
**have** *seg-tuV*:  $\text{subpathin } t \ u \ p \ \{0..1\} \subseteq V$   
**by** (*rule order-trans*[*OF subpathin-image-subset-left*[*OF t01 u01 v01*]]) (*use tu uv seg-tvV in auto*)  
**have** *seg-uvV*:  $\text{subpathin } u \ v \ p \ \{0..1\} \subseteq V$   
**by** (*rule order-trans*[*OF subpathin-image-subset-right*[*OF t01 u01 v01*]]) (*use tu uv seg-tvV in auto*)  
**have** *loop-tu*:  $\text{segment-loop } p \ t \ u \in \text{loop-space } V \ x0$   
**by** (*rule segment-loop-in-V*[*OF p-path p-imgW t01 u01 ptUV puUV seg-tuV*])  
**have** *loop-uv*:  $\text{segment-loop } p \ u \ v \in \text{loop-space } V \ x0$   
**by** (*rule segment-loop-in-V*[*OF p-path p-imgW u01 v01 puUV pvUV seg-uvV*])  
**have** *class-tu-in*:  $\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u) \in G2$   
**by** (*rule loop-class-in-space*[*OF loop-tu*])  
**have** *class-uv-in*:  $\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v) \in G2$   
**by** (*rule loop-class-in-space*[*OF loop-uv*])  
**have** *mult-in*:  
 $\text{mult2 } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \in G2$   
**by** (*rule fundamental-group-mult-in-space*[*OF class-tu-in class-uv-in*])  
**have** *mult-eq*:  
 $\text{mult2 } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) =$   
 $\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ v)$   
**by** (*rule segment-loop-mult-eq-right*[*OF p-path p-imgW t01 u01 v01 tu uv ptUV puUV pvUV seg-tvV*])  
**have** *step*:  
*carrier-fpw-reduction-step*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{WordRight } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \ \text{rest}))$   
 $(\text{WordRight } (\text{mult2 } (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v))) \ \text{rest}))$

**proof** (*rule carrier-fpw-reduction-step.combine-right*)  
**show**  $\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u) \in G2$   
**by** (*rule class-tu-in*)  
**show**  $\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v) \in G2$   
**by** (*rule class-uv-in*)  
**show**  $\text{mult2} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u)) \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \in G2$   
**by** (*rule mult-in*)  
**show** *fpw-in-space*  $G1 \ G2 \ \text{rest}$   
**by** (*rule rest-in*)  
**qed**  
**have** *red*:  
*carrier-fpw-reduction*  $G1 \ G2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \ \text{rest}))$   
 $(\text{WordRight} \ (\text{mult2} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v))) \ \text{rest})$   
**by** (*rule carrier-fpw-reduction.step[OF step]*)  
**have** *rel*:  
*carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v)) \ \text{rest}))$   
 $(\text{WordRight} \ (\text{mult2} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u))$   
 $(\text{loop-class } V \ x0 \ (\text{segment-loop } p \ u \ v))) \ \text{rest})$   
**by** (*rule carrier-full-amalg-equiv.of-reduction[OF red]*)  
**show** *?thesis*  
**using** *rel mult-eq* **by** *simp*  
**qed**

**lemma** *segment-word-switch*:

**assumes** *p-path*:  $\text{path } p$   
**and** *p-imgW*:  $\text{path-image } p \subseteq W$   
**and** *t01*:  $t \in \{0..1\}$   
**and** *u01*:  $u \in \{0..1\}$   
**and** *tu*:  $t < u$   
**and** *ptUV*:  $p \ t \in U \cap V$   
**and** *puUV*:  $p \ u \in U \cap V$   
**and** *segUV*:  $\text{subpathin } t \ u \ p \ \{0..1\} \subseteq U \cap V$   
**and** *rest-in*: *fpw-in-space*  $G1 \ G2 \ \text{rest}$   
**and** *bc*:  $b \neq c$

**shows** *carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $(\text{if } b \ \text{then } \text{WordLeft} \ (\text{loop-class } U \ x0 \ (\text{segment-loop } p \ t \ u)) \ \text{rest}$   
 $\ \text{else } \text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u)) \ \text{rest})$   
 $(\text{if } c \ \text{then } \text{WordLeft} \ (\text{loop-class } U \ x0 \ (\text{segment-loop } p \ t \ u)) \ \text{rest}$   
 $\ \text{else } \text{WordRight} \ (\text{loop-class } V \ x0 \ (\text{segment-loop } p \ t \ u)) \ \text{rest})$

**proof** –

**have** *conn-t-img*:  $\text{path-image} \ (\text{connector } (p \ t)) \subseteq U \cap V$   
**using** *connector-image-subset[OF ptUV]* **by** *blast*  
**have** *conn-u-img*:  $\text{path-image} \ (\text{connector } (p \ u)) \subseteq U \cap V$

**using** *connector-image-subset*[*OF puUV*] **by** *blast*  
**have** *segH*: *segment-loop p t u*  $\in$  *loop-space* ( $U \cap V$ ) *x0*  
**by** (*rule segment-loop-in-set*[**where**  $S = U \cap V$ ])  
*(use p-path p-imgW t01 u01 ptUV puUV segUV x0-in-UV conn-t-img*  
*conn-u-img in auto)*  
**have** *h-in*: *loop-class* ( $U \cap V$ ) *x0* (*segment-loop p t u*)  $\in$  *H*  
**by** (*rule loop-class-in-space*[*OF segH*])  
**have** *base*:  
*carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2*  
*(bridge-word b (loop-class (U  $\cap$  V) x0 (segment-loop p t u)) rest)*  
*(bridge-word c (loop-class (U  $\cap$  V) x0 (segment-loop p t u)) rest)*  
**by** (*rule bridge-word-switch*[*OF h-in bc*])  
**show** *?thesis*  
**using** *base*  
**by** (*cases b; cases c*) (*simp-all add: i1-loop-class-eq*[*OF segH*] *i2-loop-class-eq*[*OF*  
*segH*])  
**qed**

**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*:  $p u \in U \cap V$   
**shows** *svk-partition p* ( $t \# u \# v \# us$ ) ( $b \# b \# bs$ )  
**proof** –  
**have** *t01*:  $t \in \{0..1\}$  **and** *v01*:  $v \in \{0..1\}$   
**and** *seg-tv*:  
*(if b then subpathin t v p ‘ {0..1}  $\subseteq$  U else subpathin t v p ‘ {0..1}  $\subseteq$  V)*  
**and** *tail*: *svk-partition p* ( $v \# us$ ) *bs*  
**using** *part* **by** *simp-all*  
**have** *ptUV*:  $p t \in U \cap V$   
**using** *part* **by** *simp*  
**have** *pvUV*:  $p v \in U \cap V$   
**using** *tail* **by** (*cases us; cases bs*) *simp-all*  
**have** *u01*:  $u \in \{0..1\}$   
**using** *t01 v01 tu uv* **by** *auto*  
**have** *seg-tu*:  
*(if b then subpathin t u p ‘ {0..1}  $\subseteq$  U else subpathin t u p ‘ {0..1}  $\subseteq$  V)*  
**proof** (*cases b*)  
**case** *True*  
**have** *subpathin t u p ‘ {0..1}  $\subseteq$  subpathin t v p ‘ {0..1}*  
**by** (*rule subpathin-image-subset-left*[*OF t01 u01 v01*]) (*use tu uv in auto*)  
**then show** *?thesis*  
**using** *seg-tv True* **by** *auto*  
**next**  
**case** *False*  
**have** *subpathin t u p ‘ {0..1}  $\subseteq$  subpathin t v p ‘ {0..1}*  
**by** (*rule subpathin-image-subset-left*[*OF t01 u01 v01*]) (*use tu uv in auto*)  
**then show** *?thesis*

```

    using seg-tv False by auto
qed
have seg-uv:
  (if b then subpathin u v p ‘ {0..1} ⊆ U else subpathin u v p ‘ {0..1} ⊆ V)
proof (cases b)
  case True
  have subpathin u v p ‘ {0..1} ⊆ subpathin t v p ‘ {0..1}
    by (rule subpathin-image-subset-right[OF t01 u01 v01]) (use tu uv in auto)
  then show ?thesis
    using seg-tv True by auto
next
  case False
  have subpathin u v p ‘ {0..1} ⊆ subpathin t v p ‘ {0..1}
    by (rule subpathin-image-subset-right[OF t01 u01 v01]) (use tu uv in auto)
  then show ?thesis
    using seg-tv False by auto
qed
show ?thesis
  using t01 ptUV u01 tu seg-tu puUV v01 pvUV uv seg-uv tail by simp
qed

```

```

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)
  using assms
proof (induction length bs + length cs arbitrary: t ts bs us cs rule: less-induct)
  case less
  note p-loop = less.prem1(1)
  note part1 = less.prem1(2)
  note part2 = less.prem1(3)

```

```

from p-loop have p-path: path p and p-imgW: path-image p ⊆ W
  unfolding loop-space-def by auto

```

```

show ?case
proof (cases bs)
  case Nil
  have ts-nil: ts = []
    using part1 Nil by (cases ts) simp-all
  have t1: t = 1
    using svk-partition-last-eq-one[OF part1] ts-nil by simp
  show ?thesis
proof (cases cs)
  case Nil
  then show ?thesis

```

```

    using ts-nil by simp
next
case (Cons c cs')
obtain v us' where us: us = v # us'
  using part2 Cons by (cases us) auto
have v01: v ∈ {0..1}
  using part2 Cons us by simp
have t < v
  using part2 Cons us by simp
then have False
  using t1 v01 by auto
then show ?thesis
  by simp
qed
next
case (Cons b bs')
note bs-cons = Cons
obtain u ts' where ts: ts = u # ts'
  using part1 Cons by (cases ts) auto
have t01: t ∈ {0..1} and ptUV: p t ∈ U ∩ V
  and u01: u ∈ {0..1} and tu: t < u
  and seg-tu:
    (if b then subpathin t u p ' {0..1} ⊆ U else subpathin t u p ' {0..1} ⊆ V)
  and tail1: svk-partition p (u # ts') bs'
  using part1 Cons ts by simp-all
have puUV: p u ∈ U ∩ V
  by (rule svk-partition-next-in-intersection[OF part1[unfolded ts Cons]])
have first-in1:
  (if b then loop-class U x0 (segment-loop p t u) ∈ G1
   else loop-class V x0 (segment-loop p t u) ∈ G2)
proof (cases b)
case True
  have seg-loop: segment-loop p t u ∈ loop-space U x0
    by (rule segment-loop-in-U[OF p-path p-imgW t01 u01 ptUV puUV]) (use
seg-tu True in auto)
  show ?thesis
    using True by (auto intro: loop-class-in-space[OF seg-loop])
next
case False
  have seg-loop: segment-loop p t u ∈ loop-space V x0
    by (rule segment-loop-in-V[OF p-path p-imgW t01 u01 ptUV puUV]) (use
seg-tu False in auto)
  show ?thesis
    using False by (auto intro: loop-class-in-space[OF seg-loop])
qed

show ?thesis
proof (cases cs)
case Nil

```

```

have us-nil: us = []
  using part2 Nil by (cases us) simp-all
have t1: t = 1
  using svk-partition-last-eq-one[OF part2] us-nil by simp
have contradiction: False
  using tu t1 u01 by auto
from contradiction show ?thesis
  by simp
next
case (Cons c cs')
obtain v us' where us: us = v # us'
  using part2 Cons by (cases us) auto
have v01: v ∈ {0..1} and tv: t < v
  and seg-tv:
    (if c then subpathin t v p ' {0..1} ⊆ U else subpathin t v p ' {0..1} ⊆ V)
  and tail2: svk-partition p (v # us') cs'
  using part2 Cons us by simp-all
have pvUV: p v ∈ U ∩ V
  using tail2 by (cases us'; cases cs') simp-all
have tail2-in:
  fpw-in-space G1 G2 (partition-word p (v # us') cs')
  by (rule svk-partition-partition-word-in-space[OF p-loop tail2])

show ?thesis
proof (cases u = v)
case True
  note uv-eq = True
  have smaller: length bs' + length cs' < length (b # bs') + length (c # cs')
    by simp
  have tail-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (u # ts') bs')
    (partition-word p (u # us') cs')
  proof -
  have tail2': svk-partition p (u # us') cs'
    using tail2 True by simp
  have smaller': length bs' + length cs' < length (b # bs') + length cs
    using smaller Cons by simp
  have bs-eq: bs = b # bs'
    by (rule bs-cons)
  have smaller'': length bs' + length cs' < length bs + length cs
    using smaller' bs-eq by simp
  have ih:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (u # ts') bs')
    (partition-word p (u # us') cs')
    by (rule less.hyps[OF smaller'' p-loop tail1 tail2'])
  show ?thesis
    by (rule ih)

```

```

qed
have pref-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (t # u # ts') (b # bs'))
  (if b then WordLeft (loop-class U x0 (segment-loop p t u))
    (partition-word p (u # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
proof (cases b)
case True
have class-in: loop-class U x0 (segment-loop p t u) ∈ G1
using first-in1 True by simp
have rel':
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordLeft (loop-class U x0 (segment-loop p t u))
    (partition-word p (u # ts') bs'))
  (WordLeft (loop-class U x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
  by (rule carrier-full-amalg-equiv-left-context[OF tail-rel class-in])
then show ?thesis
using rel' ts True by simp
next
case False
have class-in: loop-class V x0 (segment-loop p t u) ∈ G2
using first-in1 False by simp
have rel':
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # ts') bs'))
  (WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
  by (rule carrier-full-amalg-equiv-right-context[OF tail-rel class-in])
then show ?thesis
using rel' ts False by simp
qed
show ?thesis
proof (cases b = c)
case True
note bc-eq = True
show ?thesis
proof (cases b)
case True
then show ?thesis
using pref-rel bc-eq True ts us uv-eq bs-cons Cons by simp
next
case False
then show ?thesis
using pref-rel bc-eq False ts us uv-eq bs-cons Cons by simp
qed

```

```

next
case False
have segUV: subpathin t u p ‘ {0..1} ⊆ U ∩ V
  using seg-tu seg-tv True False by (cases b; cases c) auto
have tail2': svk-partition p (u # us') cs'
  using tail2 uv-eq by simp
have rest-in:
  fpw-in-space G1 G2 (partition-word p (u # us') cs')
  by (rule svk-partition-partition-word-in-space[OF p-loop tail2'])
have bc-ne: b ≠ c
  by (rule False)
have switch-raw:
  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))
    (partition-word p (u # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
  (if c then WordLeft (loop-class U x0 (segment-loop p t u))
    (partition-word p (u # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
  by (rule segment-word-switch[where rest = partition-word p (u # us')
cs',
  OF p-path p-imgW t01 u01 tu ptUV puUV segUV rest-in bc-ne])
have switch:
  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))
    (partition-word p (u # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t u))
    (partition-word p (u # us') cs'))
  (partition-word p (t # u # us') (c # cs'))
  using switch-raw by (cases c) simp-all
have step:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (t # u # ts') (b # bs'))
  (partition-word p (t # u # us') (c # cs'))
  by (rule carrier-full-amalg-equiv.trans[OF pref-rel switch])
show ?thesis
  using step True ts us uv-eq bs-cons Cons by simp
qed
next
case False
show ?thesis
proof (cases u < v)
case True
note uv-lt = True
have part2-tv: svk-partition p (t # v # us') (c # cs')
  using part2 Cons us by simp
have split2: svk-partition p (t # u # v # us') (c # c # cs')

```

```

    by (rule svk-partition-split-head[OF part2-tv tu True puUV])
  have split-tail2: svk-partition p (u # v # us') (c # cs')
    using split2 by simp
  have split-tail2-in:
    fpw-in-space G1 G2 (partition-word p (u # v # us') (c # cs'))
    by (rule svk-partition-partition-word-in-space[OF p-loop split-tail2])
  have tail2-in:
    fpw-in-space G1 G2 (partition-word p (v # us') cs')
    by (rule svk-partition-partition-word-in-space[OF p-loop tail2])
  have split2-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # v # us') (c # c # cs'))
    (partition-word p (t # v # us') (c # cs'))
  proof (cases c)
  case True
  have seg-tvU: subpathin t v p ' {0..1} ⊆ U
    using seg-tv True by auto
  have raw:
    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))
    (partition-word p (v # us') cs')))
    (WordLeft (loop-class U x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
    by (rule segment-word-split-left-equiv[OF p-loop t01 u01 v01 tu uv-lt
ptUV puUV pvUV seg-tvU tail2-in])
  then show ?thesis
    using True by simp
  next
  case False
  have seg-tvV: subpathin t v p ' {0..1} ⊆ V
    using seg-tv False by auto
  have raw:
    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))
    (partition-word p (v # us') cs')))
    (WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
    by (rule segment-word-split-right-equiv[OF p-loop t01 u01 v01 tu uv-lt
ptUV puUV pvUV seg-tvV tail2-in])
  then show ?thesis
    using False by simp
  qed
  have smaller: length bs' + length (c # cs') < length (b # bs') + length (c
# cs')
    by simp
  have tail-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2

```

$(\text{partition-word } p (u \# ts') bs')$   
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$

**proof** –

*cs*

**have** *smaller'*:  $\text{length } bs' + \text{length } (c \# cs') < \text{length } (b \# bs') + \text{length}$   
**using** *smaller Cons* **by** *simp*  
**have** *bs-eq*:  $bs = b \# bs'$   
**by** (*rule bs-cons*)  
**have** *smaller''*:  $\text{length } bs' + \text{length } (c \# cs') < \text{length } bs + \text{length } cs$   
**using** *smaller' bs-eq* **by** *simp*  
**have** *ih*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
 $(\text{partition-word } p (u \# ts') bs')$   
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$   
**by** (*rule less.hyps*[*OF smaller'' p-loop tail1 split-tail2*])  
**show** *?thesis*  
**by** (*rule ih*)  
**qed**  
**have** *pref-rel*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
 $(\text{partition-word } p (t \# u \# ts') (b \# bs'))$   
*if*  $b$  **then** *WordLeft* (*loop-class*  $U\ x0$  (*segment-loop*  $p\ t\ u$ ))  
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$   
**else** *WordRight* (*loop-class*  $V\ x0$  (*segment-loop*  $p\ t\ u$ ))  
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$   
**proof** (*cases b*)  
**case** *True*  
**have** *class-in*: *loop-class*  $U\ x0$  (*segment-loop*  $p\ t\ u$ )  $\in G1$   
**using** *first-in1 True* **by** *simp*  
**have** *rel'*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
 $(\text{WordLeft } (\text{loop-class } U\ x0\ (\text{segment-loop } p\ t\ u)))$   
 $(\text{partition-word } p (u \# ts') bs')$   
 $(\text{WordLeft } (\text{loop-class } U\ x0\ (\text{segment-loop } p\ t\ u)))$   
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$   
**by** (*rule carrier-full-amalg-equiv-left-context*[*OF tail-rel class-in*])  
**from** *rel'* **show** *?thesis*  
**using** *True ts* **by** *simp*  
**next**  
**case** *False*  
**have** *class-in*: *loop-class*  $V\ x0$  (*segment-loop*  $p\ t\ u$ )  $\in G2$   
**using** *first-in1 False* **by** *simp*  
**have** *rel'*:  
*carrier-full-amalg-equiv*  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$   
 $(\text{WordRight } (\text{loop-class } V\ x0\ (\text{segment-loop } p\ t\ u)))$   
 $(\text{partition-word } p (u \# ts') bs')$   
 $(\text{WordRight } (\text{loop-class } V\ x0\ (\text{segment-loop } p\ t\ u)))$   
 $(\text{partition-word } p (u \# v \# us') (c \# cs'))$   
**by** (*rule carrier-full-amalg-equiv-right-context*[*OF tail-rel class-in*])

```

then show ?thesis
  using False ts by simp
qed
show ?thesis
proof (cases b = c)
  case True
  have pref-to-split:
    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))
      (partition-word p (u # v # us') (c # cs'))
    else WordRight (loop-class V x0 (segment-loop p t u))
      (partition-word p (u # v # us') (c # cs')))
    (partition-word p (t # u # v # us') (c # c # cs'))
  using True by simp
  have step:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # ts') (b # bs'))
    (partition-word p (t # u # v # us') (c # c # cs'))
  by (rule carrier-full-amalg-equiv.trans[OF pref-rel pref-to-split])
  have final:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # ts') (b # bs'))
    (partition-word p (t # v # us') (c # cs'))
  by (rule carrier-full-amalg-equiv.trans[OF step split2-rel])
  show ?thesis
  using final ts us bs-cons Cons by simp
next
  case False
  have segUV: subpathin t u p ‘ {0..1}  $\subseteq$  U  $\cap$  V
    using seg-tu split2 False by (cases b; cases c) auto
  have switch-raw:
    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))
      (partition-word p (u # v # us') (c # cs'))
    else WordRight (loop-class V x0 (segment-loop p t u))
      (partition-word p (u # v # us') (c # cs')))
    (if c then WordLeft (loop-class U x0 (segment-loop p t u))
      (partition-word p (u # v # us') (c # cs'))
    else WordRight (loop-class V x0 (segment-loop p t u))
      (partition-word p (u # v # us') (c # cs')))
  by (rule segment-word-switch[where rest = partition-word p (u # v #
us') (c # cs'),
    OF p-path p-imgW t01 u01 tu ptUV puUV segUV split-tail2-in
False])
  have switch:
    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))
      (partition-word p (u # v # us') (c # cs'))
    else WordRight (loop-class V x0 (segment-loop p t u))

```

```

      (partition-word p (u # v # us') (c # cs'))
      (partition-word p (t # u # v # us') (c # c # cs'))
    using switch-raw by (cases c) simp-all
  have step:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word p (t # u # ts') (b # bs'))
      (partition-word p (t # u # v # us') (c # c # cs'))
    by (rule carrier-full-amalg-equiv.trans[OF pref-rel switch])
  have final:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word p (t # u # ts') (b # bs'))
      (partition-word p (t # v # us') (c # cs'))
    by (rule carrier-full-amalg-equiv.trans[OF step split2-rel])
  show ?thesis
    using final ts us bs-cons Cons by simp
qed
next
case False-lt: False
have vu: v < u
  using False False-lt by linarith
have part1-tu: svk-partition p (t # u # ts') (b # bs')
  using part1 bs-cons ts by simp
have split1: svk-partition p (t # v # u # ts') (b # b # bs')
  by (rule svk-partition-split-head[OF part1-tu tv vu pvUV])
have split-tail1: svk-partition p (v # u # ts') (b # bs')
  using split1 by simp
have split1-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # v # u # ts') (b # b # bs'))
    (partition-word p (t # u # ts') (b # bs'))
proof (cases b)
case True
have seg-tuU: subpathin t u p ' {0..1} ⊆ U
  using seg-tu True by simp
have raw:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordLeft (loop-class U x0 (segment-loop p t v))
      (WordLeft (loop-class U x0 (segment-loop p v u))
        (partition-word p (u # ts') bs')))
    (WordLeft (loop-class U x0 (segment-loop p t u))
      (partition-word p (u # ts') bs'))
  by (rule segment-word-split-left-equiv[where rest = partition-word p (u
# ts') bs',
    OF p-loop t01 v01 u01 tv vu ptUV pvUV puUV seg-tuU
    svk-partition-partition-word-in-space[OF p-loop tail1]])
then show ?thesis
  using True by simp
next
case False

```

```

have seg-tuV: subpathin t u p ‘ {0..1} ⊆ V
using seg-tu False by simp
have raw:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordRight (loop-class V x0 (segment-loop p t v))
      (WordRight (loop-class V x0 (segment-loop p v u))
        (partition-word p (u # ts^) bs^)))
    (WordRight (loop-class V x0 (segment-loop p t u))
      (partition-word p (u # ts^) bs^))
  by (rule segment-word-split-right-equiv[where rest = partition-word p
    (u # ts^) bs^,
      OF p-loop t01 v01 u01 tv vu ptUV pvUV puUV seg-tuV
      svk-partition-partition-word-in-space[OF p-loop tail1]])
then show ?thesis
using False by simp
qed
have smaller: length (b # bs^) + length cs^ < length (b # bs^) + length (c
# cs^)
by simp
have tail-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (v # u # ts^) (b # bs^))
    (partition-word p (v # us^) cs^)
proof –
  have smaller': length (b # bs^) + length cs^ < length bs + length (c #
cs^)
using smaller bs-cons by simp
have cs-eq: cs = c # cs^
by (rule Cons)
have smaller'': length (b # bs^) + length cs^ < length bs + length cs
using smaller' cs-eq by simp
have ih:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (v # u # ts^) (b # bs^))
    (partition-word p (v # us^) cs^)
  by (rule less.hyps[OF smaller'' p-loop split-tail1 tail2])
show ?thesis
by (rule ih)
qed
have first-in2:
  (if c then loop-class U x0 (segment-loop p t v) ∈ G1
  else loop-class V x0 (segment-loop p t v) ∈ G2)
proof (cases c)
case True
have seg-loop: segment-loop p t v ∈ loop-space U x0
by (rule segment-loop-in-U[OF p-path p-imgW t01 v01 ptUV pvUV])
(use seg-tv True in auto)
show ?thesis
using True by (auto intro: loop-class-in-space[OF seg-loop])

```

```

next
  case False
  have seg-loop: segment-loop p t v ∈ loop-space V x0
    by (rule segment-loop-in-V[OF p-path p-imgW t01 v01 ptUV pvUV])
  (use seg-tv False in auto)
  show ?thesis
    using False by (auto intro: loop-class-in-space[OF seg-loop])
qed
show ?thesis
proof (cases b = c)
  case True
  from True have bc: b = c .
  have pref-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word p (t # v # u # ts') (b # b # bs'))
      (partition-word p (t # v # us') (c # cs'))
  proof (cases b)
    case True
    have class-in: loop-class U x0 (segment-loop p t v) ∈ G1
      using first-in2 bc True by simp
    from bc True have c-true: c
      by simp
    have tail-rel':
      carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
        (partition-word p (v # u # ts') (True # bs'))
        (partition-word p (v # us') cs')
      using tail-rel True c-true by simp
    have ctx-rel:
      carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
        (WordLeft (loop-class U x0 (segment-loop p t v))
          (partition-word p (v # u # ts') (True # bs')))
        (WordLeft (loop-class U x0 (segment-loop p t v))
          (partition-word p (v # us') cs'))
      by (rule carrier-full-amalg-equiv-left-context[OF tail-rel' class-in])
    show ?thesis
      using bc True ctx-rel by simp
  next
  case False
  have class-in: loop-class V x0 (segment-loop p t v) ∈ G2
    using first-in2 bc False by simp
  from bc False have c-false: ¬ c
    by simp
  have tail-rel':
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word p (v # u # ts') (False # bs'))
      (partition-word p (v # us') cs')
    using tail-rel False c-false by simp
  have ctx-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2

```

```

      (WordRight (loop-class V x0 (segment-loop p t v))
        (partition-word p (v # u # ts') (False # bs')))
      (WordRight (loop-class V x0 (segment-loop p t v))
        (partition-word p (v # us') cs'))
    by (rule carrier-full-amalg-equiv-right-context[OF tail-rel' class-in])
  show ?thesis
    using bc False ctx-rel by simp
qed
have from-orig:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (t # u # ts') (b # bs'))
  (partition-word p (t # v # u # ts') (b # b # bs'))
  by (rule carrier-full-amalg-equiv.sym[OF split1-rel])
have step:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (t # u # ts') (b # bs'))
  (partition-word p (t # v # us') (c # cs'))
  by (rule carrier-full-amalg-equiv.trans[OF from-orig pref-rel])
show ?thesis
  using step us ts bs-cons Cons by simp
next
case False
have pref-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (t # v # u # ts') (b # b # bs'))
  (if b then WordLeft (loop-class U x0 (segment-loop p t v))
    (partition-word p (v # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
proof -
  have first-in-split:
    (if b then loop-class U x0 (segment-loop p t v) ∈ G1
    else loop-class V x0 (segment-loop p t v) ∈ G2)
  using split1 by (cases b) (auto intro: loop-class-in-space
    segment-loop-in-U[OF p-path p-imgW t01 v01 ptUV pvUV]
    segment-loop-in-V[OF p-path p-imgW t01 v01 ptUV pvUV])
  show ?thesis
proof (cases b)
case True
  have class-in: loop-class U x0 (segment-loop p t v) ∈ G1
  using first-in-split True by simp
  have tail-rel':
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (v # u # ts') (True # bs'))
    (partition-word p (v # us') cs')
  using tail-rel True by simp
  have ctx-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordLeft (loop-class U x0 (segment-loop p t v))

```

```

      (partition-word p (v # u # ts') (True # bs'))
      (WordLeft (loop-class U x0 (segment-loop p t v))
        (partition-word p (v # us') cs'))
    by (rule carrier-full-amalg-equiv-left-context[OF tail-rel' class-in])
  show ?thesis
  using True ctx-rel by simp
next
case False
have class-in: loop-class V x0 (segment-loop p t v) ∈ G2
  using first-in-split False by simp
have tail-rel':
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p (v # u # ts') (False # bs'))
  (partition-word p (v # us') cs')
  using tail-rel False by simp
have ctx-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # u # ts') (False # bs')))
  (WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
  by (rule carrier-full-amalg-equiv-right-context[OF tail-rel' class-in])
show ?thesis
  using False ctx-rel by simp
qed
qed
have segUV: subpathin t v p ' {0..1} ⊆ U ∩ V
  using split1 seg-tv False by (cases b; cases c) auto
have switch-raw:
  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 v))
    (partition-word p (v # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
  (if c then WordLeft (loop-class U x0 (segment-loop p t v))
    (partition-word p (v # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
  by (rule segment-word-switch[where rest = partition-word p (v # us')
cs',
      OF p-path p-imgW t01 v01 tv ptUV pvUV segUV tail2-in False])
have switch:
  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 v))
    (partition-word p (v # us') cs')
  else WordRight (loop-class V x0 (segment-loop p t v))
    (partition-word p (v # us') cs'))
  (partition-word p (t # v # us') (c # cs'))
  using switch-raw by (cases c) simp-all

```

```

have from-orig:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # ts') (b # bs'))
    (partition-word p (t # v # u # ts') (b # b # bs'))
  by (rule carrier-full-amalg-equiv.sym[OF split1-rel])
have step1:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # ts') (b # bs'))
    (if b then WordLeft (loop-class U x0 (segment-loop p t v))
      (partition-word p (v # us') cs')
      else WordRight (loop-class V x0 (segment-loop p t v))
      (partition-word p (v # us') cs'))
  by (rule carrier-full-amalg-equiv.trans[OF from-orig pref-rel])
have final:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word p (t # u # ts') (b # bs'))
    (partition-word p (t # v # us') (c # cs'))
  by (rule carrier-full-amalg-equiv.trans[OF step1 switch])
show ?thesis
  using final ts us bs-cons Cons by simp
qed
qed
qed
qed
qed
qed

```

**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** –

```

obtain t ts' where ts: ts = t # ts'
  using valid-partition-hd(1)[OF part1] by (cases ts) auto
obtain u us' where us: us = u # us'
  using valid-partition-hd(1)[OF part2] by (cases us) auto
have valid-ts: valid-partition p (t # ts') bs
  using part1 unfolding ts by simp
have t0: t = 0
  by (rule valid-partition-cases(1)[OF valid-ts])
have part1': svk-partition p (t # ts') bs
  by (rule valid-partition-cases(2)[OF valid-ts])
have valid-us: valid-partition p (u # us') cs
  using part2 unfolding us by simp
have u0: u = 0
  by (rule valid-partition-cases(1)[OF valid-us])
have part2': svk-partition p (u # us') cs

```

**by** (*rule valid-partition-cases*(2)[*OF valid-us*])  
**have** *rel*:  
*carrier-full-amalg-equiv* *G1 G2 H i1 i2 mult1 one1 mult2 one2*  
*(partition-word* *p (t # ts') bs)*  
*(partition-word* *p (u # us^ cs)*)  
**proof** –  
**have** *part1-0*: *svk-partition* *p (0 # ts') bs*  
**using** *part1' t0* **by** *simp*  
**have** *part2-0*: *svk-partition* *p (0 # us^ cs)*  
**using** *part2' u0* **by** *simp*  
**show** *?thesis*  
**by** (*subst* *t0*, *subst* *u0*, *rule svk-partition-same-start-equiv*[*OF p-loop part1-0*  
*part2-0*])  
**qed**  
**then show** *?thesis*  
**using** *ts us t0 u0* **by** *simp*  
**qed**

**lemma** *strip-neighbourhood*:  
**fixes** *h* :: (*real* × *real*) ⇒ '*a*  
**assumes** *h-cont*: *continuous-map* (*top-of-set* ( $\{0..1\} \times \{0..1\}$ )) (*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* ' ( $\{a..b\} \times \{y0\}$ ) ⊆ *S*  
**and** *openS*: *open* *S*  
**shows** ∃ *N*. *openin* (*top-of-set*  $\{0..1\}$ ) *N* ∧ *y0* ∈ *N* ∧ *h* ' ( $\{a..b\} \times N$ ) ⊆ *S*  
**proof** –  
**have** *h-on*: *continuous-on* ( $\{0..1\} \times \{0..1\}$ ) *h*  
**using** *h-cont* **by** *simp*  
**have** *strip-subset*:  $\{a..b\} \times \{0..1\} \subseteq \{0..1\} \times \{0..1\}$   
**using** *a01 b01 ab* **by** *auto*  
**have** *strip-cont*: *continuous-on* ( $\{a..b\} \times \{0..1\}$ ) *h*  
**by** (*rule continuous-on-subset*[*OF h-on strip-subset*])  
**have** *strip-open*:  
*openin* (*top-of-set* ( $\{a..b\} \times \{0..1\}$ ))  
( $\{a..b\} \times \{0..1\}$ ) ∩ *h* -' *S*  
**by** (*rule continuous-openin-preimage-gen*[*OF strip-cont openS*])  
**have** *line-in*:  
( $\lambda x. (x, y0)$ ) '  $\{a..b\} \subseteq ((\{a..b\} \times \{0..1\}) \cap h -' S)$   
**using** *rowS y0-01* **by** *auto*  
**obtain** *N* **where** *N-open*: *openin* (*top-of-set*  $\{0..1\}$ ) *N*  
**and** *y0N*: *y0* ∈ *N*  
**and** *stripN*:  $\{a..b\} \times N \subseteq ((\{a..b\} \times \{0..1\}) \cap h -' S)$   
**proof** –  
**have** *strip-open-prod*:  
*openin* (*prod-topology* (*top-of-set*  $\{a..b\}$ ) (*top-of-set*  $\{0..1\}$ ))

```

      (({a..b} × {0..1}) ∩ h - ' S)
    using strip-open by simp
  have line-in-prod:
    {a..b} × {y0} ⊆ (({a..b} × {0..1}) ∩ h - ' S)
    using line-in by auto
  have compact-ab: compactin (top-of-set {a..b}) {a..b}
    using compact-Icc by simp
  obtain M N where M-open: openin (top-of-set {a..b}) M
    and N-open: openin (top-of-set {0..1}) N
    and M-cover: {a..b} ⊆ M
    and y0N: y0 ∈ N
    and MN-strip: M × N ⊆ (({a..b} × {0..1}) ∩ h - ' S)
  proof -
    have local-boxes:
      ∃ M N. openin (top-of-set {a..b}) M ∧ openin (top-of-set {0..1}) N ∧
        x ∈ M ∧ y0 ∈ N ∧ M × N ⊆ (({a..b} × {0..1}) ∩ h - ' S)
    if x-in: x ∈ {a..b} for x
  proof -
    have xy-pair: (x, y0) ∈ {a..b} × {y0}
      using x-in by auto
    have xy-in: (x, y0) ∈ (({a..b} × {0..1}) ∩ h - ' S)
      by (rule subsetD[OF line-in-prod xy-pair])
    show ?thesis
      using strip-open-prod xy-in by (metis openin-prod-topology-alt)
  qed
  then obtain U V where UV:
    ∧ x. x ∈ {a..b} ⇒
      openin (top-of-set {a..b}) (U x) ∧
      openin (top-of-set {0..1}) (V x) ∧
      x ∈ U x ∧ y0 ∈ V x ∧
      U x × V x ⊆ (({a..b} × {0..1}) ∩ h - ' S)
    by metis
  then obtain D where D: finite D D ⊆ {a..b} {a..b} ⊆ ∪ (U ' D)
    using compactinD[OF compact-ab, of U ' {a..b}]
    by (smt (verit) UN-I finite-subset-image imageE subsetI)
  show ?thesis
  proof (intro that[of ∪ (U ' D) ∩ (V ' D)])
    show openin (top-of-set {a..b}) (∪ (U ' D))
      using D UV by blast
    show openin (top-of-set {0..1}) (∩ (V ' D))
  proof (rule openin-Inter)
    show finite (V ' D)
      using D by simp
    show V ' D ≠ {}
  proof
    assume V ' D = {}
    then have D = {}
      by auto
    with D(3) ab show False
  end
end

```

by *simp*  
 qed  
 show *openin* (*top-of-set*  $\{0..1\}$ )  $T$  if  $T \in V \text{ ' } D$  for  $T$   
 proof –  
   from that obtain  $x$  where  $xD: x \in D$  and  $T\text{-def}: T = V x$   
   by *auto*  
   show *?thesis*  
   using  $D(2)$   $UV$   $xD$   $T\text{-def}$  by *blast*  
 qed  
 qed  
 show  $\{a..b\} \subseteq \bigcup (U \text{ ' } D)$   
   using  $D$  by *blast*  
 show  $y0 \in \bigcap (V \text{ ' } D)$   
   using  $D$   $UV$  by *force*  
 show  $(\bigcup (U \text{ ' } D)) \times (\bigcap (V \text{ ' } D)) \subseteq ((\{a..b\} \times \{0..1\}) \cap h \text{ - ' } S)$   
 proof  
   fix  $xy$   
   assume  $xy\text{-in}: xy \in (\bigcup (U \text{ ' } D)) \times (\bigcap (V \text{ ' } D))$   
   then obtain  $x$   $y$  where  $xy\text{-eq}: xy = (x, y)$   
   and  $x\text{-in}: x \in \bigcup (U \text{ ' } D)$   
   and  $y\text{-in}: y \in \bigcap (V \text{ ' } D)$   
   by *blast*  
   obtain  $d$  where  $dD: d \in D$  and  $xUd: x \in U d$   
   using  $x\text{-in}$  by *auto*  
   have  $yVd: y \in V d$   
   using  $y\text{-in}$   $dD$  by *auto*  
   have  $UdVd\text{-strip}: U d \times V d \subseteq ((\{a..b\} \times \{0..1\}) \cap h \text{ - ' } S)$   
   using  $D(2)$   $UV$   $dD$  by *blast*  
   have  $(x, y) \in U d \times V d$   
   using  $xUd$   $yVd$  by *auto*  
   then have  $(x, y) \in ((\{a..b\} \times \{0..1\}) \cap h \text{ - ' } S)$   
   using  $UdVd\text{-strip}$  by *blast*  
   then show  $xy \in ((\{a..b\} \times \{0..1\}) \cap h \text{ - ' } S)$   
   using  $xy\text{-eq}$  by *simp*  
 qed  
 qed  
 qed  
 have  $stripN0: \{a..b\} \times N \subseteq ((\{a..b\} \times \{0..1\}) \cap h \text{ - ' } S)$   
   using *M-cover* *MN-strip* by *blast*  
 show *thesis*  
   by (*rule that*[of  $N$ ], *rule N-open*, *rule y0N*, use *stripN0* in *auto*)  
 qed  
 show *?thesis*  
 proof (*intro exI conjI*)  
   show *openin* (*top-of-set*  $\{0..1\}$ )  $N$   
   by (*rule N-open*)  
   show  $y0 \in N$   
   by (*rule y0N*)  
   show  $h \text{ ' } (\{a..b\} \times N) \subseteq S$

```

    using stripN by auto
  qed
qed

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 (λx. 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)
  using part
  proof (induction ts arbitrary: bs)
    case Nil
    then show ?case
      by simp
  next
    case (Cons t ts)
    show ?case
    proof (cases ts)
      case Nil
      have bs-nil: bs = []
        using Cons.prem1 Nil by (cases bs) simp-all
      have t1: t = 1
        using Cons.prem1 Nil bs-nil by simp
      have t01: t ∈ {0..1}
        using t1 by simp
      have tt: t ≤ t
        by simp
      have point-row: h ' ({t..t} × {y0}) ⊆ U ∩ V
        using Cons.prem1 Nil bs-nil y0-01 by auto
      have UV-open: open (U ∩ V)
        using U-open V-open by auto
      have point-neigh:
        ∃ N. openin (top-of-set {0..1}) N ∧ y0 ∈ N ∧ h ' ({t..t} × N) ⊆ U ∩ V
        by (rule strip-neighbourhood[where a = t and b = t and S = U ∩ V, OF
h-cont t01 t01 y0-01 tt point-row UV-open])
      then obtain N where N-open: openin (top-of-set {0..1}) N
        and y0N: y0 ∈ N
        and pointN: h ' ({t..t} × N) ⊆ U ∩ V
        by blast
      show ?thesis
    proof (intro exI conjI ballI allI impI)
      show openin (top-of-set {0..1}) N
        by (rule N-open)
      show y0 ∈ N
        by (rule y0N)
    end
    end
  end

```

```

show  $h \text{ ' } (\{x\} \times N) \subseteq U \cap V$  if  $x \in \text{set } (t \# ts)$  for  $x$ 
  using that Nil pointN by auto
show  $\text{rectangle-partition } h \ y \ z \ (t \# ts) \ bs$  if  $y \leq z \ \{y..z\} \subseteq N$  for  $y \ z$ 
  using t1 Nil bs-nil by simp
qed
next
case (Cons u us)
obtain  $b \ bs'$  where  $bs: bs = b \# \ bs'$ 
  using Cons.premis Cons by (cases bs) auto
have t01:  $t \in \{0..1\}$  and u01:  $u \in \{0..1\}$  and tu:  $t < u$ 
  and seg-side:
  (if b
    then subpathin t u ( $\lambda x. h \ (x, y0)$ ) '  $\{0..1\} \subseteq U$ 
    else subpathin t u ( $\lambda x. h \ (x, y0)$ ) '  $\{0..1\} \subseteq V$ )
  and tail: svk-partition ( $\lambda x. h \ (x, y0)$ ) ( $u \# \ us$ )  $bs'$ 
  and ptUV:  $h \ (t, y0) \in U \cap V$ 
  using Cons.premis Cons bs by simp-all
have seg-row:  $h \ \{t..u\} \times \{y0\} \subseteq (\text{if } b \ \text{then } U \ \text{else } V)$ 
proof -
  have seg-image: ( $\lambda x. h \ (x, y0)$ ) '  $\{t..u\} \subseteq (\text{if } b \ \text{then } U \ \text{else } V)$ 
    using seg-side tu by (cases b) (auto simp: subpathin-image-eq)
  show ?thesis
  proof
    fix z
    assume z-in:  $z \in h \ \{t..u\} \times \{y0\}$ 
    then obtain  $x$  where x-in:  $x \in \{t..u\}$  and z-eq:  $z = h \ (x, y0)$ 
      by auto
    have  $h \ (x, y0) \in (\text{if } b \ \text{then } U \ \text{else } V)$ 
    proof (cases b)
      case True
      then show ?thesis
        using seg-image x-in by auto
    next
      case False
      then show ?thesis
        using seg-image x-in by auto
    qed
    then show  $z \in (\text{if } b \ \text{then } U \ \text{else } V)$ 
      using z-eq by simp
    qed
  qed
have point-row:  $h \ \{t..t\} \times \{y0\} \subseteq U \cap V$ 
  using ptUV by auto
obtain Nseg where Nseg-open:  $\text{openin } (\text{top-of-set } \{0..1\}) \ Nseg$ 
  and y0Nseg:  $y0 \in Nseg$ 
  and segN:  $h \ \{t..u\} \times Nseg \subseteq (\text{if } b \ \text{then } U \ \text{else } V)$ 
proof (cases b)
  case True
  have seg-rowU:  $h \ \{t..u\} \times \{y0\} \subseteq U$ 

```

```

    using seg-row True by simp
  obtain Nseg where Nseg-open0: openin (top-of-set {0..1}) Nseg
    and y0Nseg0: y0 ∈ Nseg
    and segN0: h ‘ ({t..u} × Nseg) ⊆ U
      using strip-neighbourhood[OF h-cont t01 u01 y0-01 less-imp-le[OF tu]
seg-rowU U-open] by blast
    show thesis
  proof (rule that[of Nseg])
    show openin (top-of-set {0..1}) Nseg
      by (rule Nseg-open0)
    show y0 ∈ Nseg
      by (rule y0Nseg0)
    show h ‘ ({t..u} × Nseg) ⊆ (if b then U else V)
      using True segN0 by simp
  qed
next
case False
have seg-rowV: h ‘ ({t..u} × {y0}) ⊆ V
  using seg-row False by simp
obtain Nseg where Nseg-open0: openin (top-of-set {0..1}) Nseg
  and y0Nseg0: y0 ∈ Nseg
  and segN0: h ‘ ({t..u} × Nseg) ⊆ V
    using strip-neighbourhood[OF h-cont t01 u01 y0-01 less-imp-le[OF tu]
seg-rowV V-open] by blast
  show thesis
  proof (rule that[of Nseg])
    show openin (top-of-set {0..1}) Nseg
      by (rule Nseg-open0)
    show y0 ∈ Nseg
      by (rule y0Nseg0)
    show h ‘ ({t..u} × Nseg) ⊆ (if b then U else V)
      using False segN0 by simp
  qed
qed
obtain Nt where Nt-open: openin (top-of-set {0..1}) Nt
  and y0Nt: y0 ∈ Nt
  and pointN: h ‘ ({t..t} × Nt) ⊆ U ∩ V
    using strip-neighbourhood[OF h-cont t01 t01 y0-01 order-refl point-row] U-open
V-open by blast
obtain Ntail where Ntail-open: openin (top-of-set {0..1}) Ntail
  and y0Ntail: y0 ∈ Ntail
  and tail-points-raw: ∀ x∈set ts. h ‘ ({x} × Ntail) ⊆ U ∩ V
  and tail-rect-raw:
    ∀ y z. y ≤ z → {y..z} ⊆ Ntail → rectangle-partition h y z ts bs'
  proof –
    have tail-ts: svk-partition (λx. h (x, y0)) ts bs'
      using tail Cons by simp
    obtain N where N-open: openin (top-of-set {0..1}) N
      and y0N: y0 ∈ N

```

```

and pointsN:  $\forall x \in \text{set } ts. h'(\{x\} \times N) \subseteq U \cap V$ 
and rectN:
   $\forall y z. y \leq z \longrightarrow \{y..z\} \subseteq N \longrightarrow \text{rectangle-partition } h y z ts bs'$ 
using Cons.IH[OF tail-ts] by blast
show thesis
proof (rule that[of N])
  show openin (top-of-set {0..1}) N
    by (rule N-open)
  show  $y0 \in N$ 
    by (rule y0N)
  show  $\forall x \in \text{set } ts. h'(\{x\} \times N) \subseteq U \cap V$ 
    by (rule pointsN)
  show  $\forall y z. y \leq z \longrightarrow \{y..z\} \subseteq N \longrightarrow \text{rectangle-partition } h y z ts bs'$ 
    by (rule rectN)
qed
qed
have tail-points:  $\forall x \in \text{set } (u \# us). h'(\{x\} \times Ntail) \subseteq U \cap V$ 
  using tail-points-raw Cons by simp
have tail-rect:
   $\forall y z. y \leq z \longrightarrow \{y..z\} \subseteq Ntail \longrightarrow \text{rectangle-partition } h y z (u \# us) bs'$ 
  using tail-rect-raw Cons by simp
let  $?N = Nseg \cap Nt \cap Ntail$ 
have N-open: openin (top-of-set {0..1}) ?N
  by (intro openin-Int Nseg-open Nt-open Ntail-open)
have  $y0N: y0 \in ?N$ 
  using  $y0Nseg y0Nt y0Ntail$  by auto
have points:
   $h'(\{x\} \times ?N) \subseteq U \cap V$  if x-in:  $x \in \text{set } (t \# u \# us)$  for  $x$ 
proof (cases x = t)
  case True
  have point-t:  $h'(\{t..t\} \times Nt) \subseteq U \cap V$ 
    by (rule pointN)
  have subset-t:  $\{x\} \times ?N \subseteq \{t..t\} \times Nt$ 
    using True by auto
  have img-subset-t:  $h'(\{x\} \times ?N) \subseteq h'(\{t..t\} \times Nt)$ 
    by (rule image-mono[OF subset-t])
  show ?thesis
    using img-subset-t point-t by blast
next
  case False
  then have  $x \in \text{set } (u \# us)$ 
    using x-in by auto
  then have point-x:  $h'(\{x\} \times Ntail) \subseteq U \cap V$ 
    using tail-points by blast
  have subset-x:  $\{x\} \times ?N \subseteq \{x\} \times Ntail$ 
    by auto
  have img-subset-x:  $h'(\{x\} \times ?N) \subseteq h'(\{x\} \times Ntail)$ 
    by (rule image-mono[OF subset-x])
  show ?thesis

```

```

    using img-subset-x point-x by blast
qed
have rect:
  rectangle-partition h y z (t # u # us) (b # bs')
  if yz:  $y \leq z$  and yzN:  $\{y..z\} \subseteq ?N$  for y z
proof -
  have yzNseg:  $\{y..z\} \subseteq Nseg$ 
  using yzN by auto
  have seg-rect:  $h'(\{t..u\} \times \{y..z\}) \subseteq (if\ b\ then\ U\ else\ V)$ 
  proof (cases b)
    case True
    have prod-subset:  $\{t..u\} \times \{y..z\} \subseteq \{t..u\} \times Nseg$ 
    using yzNseg by auto
    have h'( $\{t..u\} \times \{y..z\}) \subseteq h'(\{t..u\} \times Nseg)$ 
    by (rule image-mono[OF prod-subset])
    also have ...  $\subseteq U$ 
    using segN True by simp
    finally show ?thesis
    using True by simp
  next
  case False
  have prod-subset:  $\{t..u\} \times \{y..z\} \subseteq \{t..u\} \times Nseg$ 
  using yzNseg by auto
  have h'( $\{t..u\} \times \{y..z\}) \subseteq h'(\{t..u\} \times Nseg)$ 
  by (rule image-mono[OF prod-subset])
  also have ...  $\subseteq V$ 
  using segN False by simp
  finally show ?thesis
  using False by simp
qed
have tail-rect': rectangle-partition h y z (u # us) bs'
  by (rule tail-rect[rule-format, OF yz]) (use yzN in auto)
show ?thesis
  using t01 u01 tu seg-rect tail-rect' by (cases b) simp-all
qed
show ?thesis
proof (intro exI conjI ballI allI impI)
  show openin (top-of-set {0..1}) ?N
  by (rule N-open)
  show  $y0 \in ?N$ 
  by (rule y0N)
  show  $h'(\{x\} \times ?N) \subseteq U \cap V$  if  $x \in set(t \# ts)$  for x
  using that Cons points by simp
  show rectangle-partition h y z (t # ts) bs if  $y \leq z$   $\{y..z\} \subseteq ?N$  for y z
  using rect[OF that] Cons bs by simp
qed
qed
qed

```

```

lemma rectangle-partition-svk-partition-row:
  fixes  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$ 
  assumes part: rectangle-partition  $h\ c\ d\ ts\ bs$ 
    and edgeUV:  $\bigwedge x. x \in \text{set } ts \implies h\ ' (\{x\} \times \{c..d\}) \subseteq U \cap V$ 
    and y-in:  $y \in \{c..d\}$ 
  shows svk-partition  $(\lambda x. h\ (x, y))\ ts\ bs$ 
  using part edgeUV
proof (induction ts arbitrary: bs)
  case Nil
  then show ?case
    by simp
next
  case (Cons t ts)
  show ?case
  proof (cases ts)
  case Nil
  have bs-nil:  $bs = []$ 
    using Cons.prem1 Nil by (cases bs) simp-all
  have t1:  $t = 1$ 
    using Cons.prem1 Nil bs-nil by simp
  have ptUV:  $h\ (t, y) \in U \cap V$ 
  proof –
  have t-edge:  $h\ ' (\{t\} \times \{c..d\}) \subseteq U \cap V$ 
    using Cons.prem2[of t] Nil by simp
  have  $(t, y) \in \{t\} \times \{c..d\}$ 
    using y-in by auto
  then show ?thesis
    using t-edge by auto
  qed
  then show ?thesis
    using Nil bs-nil t1 by simp
next
  case (Cons u us)
  obtain  $b\ bs'$  where  $bs = b \# bs'$ 
    using Cons.prem1 Cons by (cases bs) auto
  have t01:  $t \in \{0..1\}$  and u01:  $u \in \{0..1\}$  and tu:  $t < u$ 
    and rect-side: (if  $b$  then  $h\ ' (\{t..u\} \times \{c..d\}) \subseteq U$  else  $h\ ' (\{t..u\} \times \{c..d\})$ 
 $\subseteq V$ )
    and tail: rectangle-partition  $h\ c\ d\ (u \# us)\ bs'$ 
    using Cons.prem1 Cons bs by simp-all
  have edge-t:  $h\ ' (\{t\} \times \{c..d\}) \subseteq U \cap V$ 
    using Cons.prem2[of t] Cons by simp
  have ptUV:  $h\ (t, y) \in U \cap V$ 
  proof –
  have  $(t, y) \in \{t\} \times \{c..d\}$ 
    using y-in by auto
  then show ?thesis
    using edge-t by auto
  qed

```

```

have seg-side:
  (if b
    then subpathin t u ( $\lambda x. h(x, y)$ ) ‘ $\{0..1\} \subseteq U$ 
    else subpathin t u ( $\lambda x. h(x, y)$ ) ‘ $\{0..1\} \subseteq V$ )
proof –
  have row-subset:  $\{t..u\} \times \{y\} \subseteq \{t..u\} \times \{c..d\}$ 
    using y-in by auto
  have row-rect:  $h ‘(\{t..u\} \times \{y\}) \subseteq (if\ b\ then\ U\ else\ V)$ 
proof (cases b)
  case True
  have  $h ‘(\{t..u\} \times \{y\}) \subseteq h ‘(\{t..u\} \times \{c..d\})$ 
    by (rule image-mono[OF row-subset])
  also have  $\dots \subseteq U$ 
    using rect-side True by simp
  finally show ?thesis
    using True by simp
next
  case False
  have  $h ‘(\{t..u\} \times \{y\}) \subseteq h ‘(\{t..u\} \times \{c..d\})$ 
    by (rule image-mono[OF row-subset])
  also have  $\dots \subseteq V$ 
    using rect-side False by simp
  finally show ?thesis
    using False by simp
qed
then show ?thesis
  using tu by (cases b) (auto simp: subpathin-image-eq)
qed
have tail-svk-ts: svk-partition ( $\lambda x. h(x, y)$ ) ts bs'
proof (rule Cons.IH[where  $bs = bs'$ ])
  show rectangle-partition h c d ts bs'
    using tail Cons by simp
  show  $\bigwedge x. x \in set\ ts \implies h ‘(\{x\} \times \{c..d\}) \subseteq U \cap V$ 
proof –
  fix x
  assume x-in:  $x \in set\ ts$ 
  have  $x \in set\ (t \# ts)$ 
    using x-in by simp
  then show  $h ‘(\{x\} \times \{c..d\}) \subseteq U \cap V$ 
    using Cons.prem1(2) by blast
qed
qed
have tail-svk: svk-partition ( $\lambda x. h(x, y)$ ) (u # us) bs'
  using tail-svk-ts Cons by simp
have step-svk: svk-partition ( $\lambda x. h(x, y)$ ) (t # u # us) (b # bs')
  using t01 u01 tu seg-side ptUV tail-svk by simp
show ?thesis
  using step-svk Cons bs by simp
qed

```

qed

**lemma** *rectangle-partition-valid-partition-row*:

**fixes**  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$

**assumes** *part*: *rectangle-partition*  $h\ c\ d\ ts\ bs$

**and** *edgeUV*:  $\bigwedge x. x \in \text{set } ts \implies h\ '(\{x\} \times \{c..d\}) \subseteq U \cap V$

**and** *y-in*:  $y \in \{c..d\}$

**and** *ts-ne*:  $ts \neq []$

**and** *hd0*:  $hd\ ts = 0$

**shows** *valid-partition*  $(\lambda x. h\ (x, y))\ ts\ bs$

**unfolding** *valid-partition-def*

**using** *rectangle-partition-svk-partition-row*[*OF part edgeUV y-in*] *ts-ne hd0* **by** *simp*

**lemma** *homotopy-row-in-loop-space*:

**fixes**  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$

**assumes** *h-cont*: *continuous-map*  $(\text{top-of-set } (\{0..1\} \times \{0..1\}))\ (\text{top-of-set } W)$

*h*

**and** *y01*:  $y \in \{0..1\}$

**and** *start*:  $h\ (0, y) = x0$

**and** *finish*:  $h\ (1, y) = x0$

**shows**  $(\lambda x. h\ (x, y)) \in \text{loop-space } W\ x0$

**proof** –

**have** *h-on*: *continuous-on*  $(\{0..1\} \times \{0..1\})\ h$

**and** *h-into*:  $h \in (\{0..1\} \times \{0..1\}) \rightarrow W$

**using** *h-cont* **by** *simp-all*

**have** *row-cont*: *continuous-on*  $\{0..1\}\ (\lambda x. h\ (x, y))$

**proof** (*rule continuous-on-compose2*[*OF h-on*])

**show** *continuous-on*  $\{0..1\}\ (\lambda x. (x, y))$

**by** (*intro continuous-intros*)

**show**  $(\lambda x. (x, y))\ '(\{0..1\} \subseteq \{0..1\} \times \{0..1\})$

**using** *y01* **by** *auto*

qed

**have** *row-path*: *path*  $(\lambda x. h\ (x, y))$

**using** *row-cont* **by** (*simp add: path-def*)

**have** *row-img*: *path-image*  $(\lambda x. h\ (x, y)) \subseteq W$

**using** *h-into y01* **by** (*auto simp: path-image-def*)

**show** *?thesis*

**unfolding** *loop-space-def pathstart-def pathfinish-def*

**using** *row-path row-img start finish* **by** *simp*

qed

**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*  $b\ (\text{fundamental-group-one } (U \cap V)\ x0)\ rest)\ rest$

**proof** (*cases b*)

**case** *True*

**have** *i1-one*:  $i1\ (\text{fundamental-group-one } (U \cap V)\ x0) = one1$

```

    by (rule fundamental-group-map-one[OF x0-in-UV]) auto
  have one1-in: one1 ∈ G1
    by (rule fundamental-group-one-in-space[OF x0-in-U])
  have red:
    carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2
    (WordLeft one1 rest) rest
    by (rule carrier-fpw-reduction.step,
        rule carrier-fpw-reduction-step.remove-left-one[OF one1-in], rule rest-in)
  show ?thesis
    using True i1-one by (simp add: carrier-full-amalg-equiv.of-reduction[OF red])
next
case False
have i2-one: i2 (fundamental-group-one (U ∩ V) x0) = one2
  by (rule fundamental-group-map-one[OF x0-in-UV]) auto
have one2-in: one2 ∈ G2
  by (rule fundamental-group-one-in-space[OF x0-in-V])
have red:
  carrier-fpw-reduction G1 G2 mult1 one1 mult2 one2
  (WordRight one2 rest) rest
  by (rule carrier-fpw-reduction.step,
      rule carrier-fpw-reduction-step.remove-right-one[OF one2-in], rule rest-in)
show ?thesis
  using False i2-one by (simp add: carrier-full-amalg-equiv.of-reduction[OF red])
qed

```

```

lemma partition-word-with-tail-respects:
  assumes p-loop: p ∈ 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)
  using part rel
proof (induction ts arbitrary: bs r s)
case Nil
then show ?case
  by simp
next
case (Cons t ts)
from p-loop have p-path: path p and p-img: path-image p ⊆ W
  unfolding loop-space-def by auto
show ?case
proof (cases ts)
case Nil
have bs-nil: bs = []
  using Cons.prem1 Nil by (cases bs) simp-all
have pw-r: partition-word-with-tail p (t # ts) bs r = r
  using Nil bs-nil by simp
have pw-s: partition-word-with-tail p (t # ts) bs s = s

```

```

    using Nil bs-nil by simp
  show ?thesis
    unfolding pw-r pw-s
    by (rule Cons.prem2)
next
case (Cons u us)
obtain b bs' where bs: bs = b # bs'
  using Cons.prem1 Cons by (cases bs) auto
have tail: svk-partition p (u # us) bs'
  using Cons.prem1 Cons bs by simp
have ts-eq: ts = u # us
  using Cons by simp
have tail-rel-ts:
  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)
  by (rule Cons.IH) (use tail Cons.prem2) ts-eq in simp-all
have tail-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word-with-tail p (u # us) bs' r)
  (partition-word-with-tail p (u # us) bs' s)
  using tail-rel-ts ts-eq by simp
have i1-back: fundamental-group-map (U ∩ V) x0 U x0 id = i1
  by simp
have i2-back: fundamental-group-map (U ∩ V) x0 V x0 id = i2
  by simp
have t01: t ∈ {0..1} and ptUV: p t ∈ U ∩ V
  using Cons.prem1 Cons bs by simp-all
have u01: u ∈ {0..1} and seg-side:
  (if b then subpathin t u p ' {0..1} ⊆ U else subpathin t u p ' {0..1} ⊆ V)
  using Cons.prem1 Cons bs by simp-all
have puUV: p u ∈ U ∩ V
  using Cons.prem1 Cons bs svk-partition-next-in-intersection[of p t u us b
bs'] by simp
show ?thesis
proof (cases b)
case True
  have segU: segment-loop p t u ∈ loop-space U x0
    by (rule segment-loop-in-U[OF p-path p-img t01 u01 ptUV puUV]) (use
seg-side True in auto)
  have class-in: loop-class U x0 (segment-loop p t u) ∈ G1
    by (rule loop-class-in-space[OF segU])
  have ctx-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (WordLeft (loop-class U x0 (segment-loop p t u))
(partition-word-with-tail p (u # us) bs' r))
    (WordLeft (loop-class U x0 (segment-loop p t u))
(partition-word-with-tail p (u # us) bs' s))
    by (rule carrier-full-amalg-equiv-left-context[OF tail-rel class-in])

```

```

show ?thesis
  using True ts-eq ctx-rel
  by (subst i1-back, subst i2-back, simp add: bs)
next
  case False
  have segV: segment-loop p t u ∈ loop-space V x0
    by (rule segment-loop-in-V[OF p-path p-img t01 u01 ptUV puUV]) (use
seg-side False in auto)
  have class-in: loop-class V x0 (segment-loop p t u) ∈ G2
    by (rule loop-class-in-space[OF segV])
  have ctx-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (WordRight (loop-class V x0 (segment-loop p t u))
        (partition-word-with-tail p (u # us) bs' r))
      (WordRight (loop-class V x0 (segment-loop p t u))
        (partition-word-with-tail p (u # us) bs' s))
    by (rule carrier-full-amalg-equiv-right-context[OF tail-rel class-in])
  show ?thesis
    using False ts-eq ctx-rel
    by (subst i1-back, subst i2-back, simp add: bs)
qed
qed
qed

```

```

lemma valid-partition-tail-bridge-one-equiv:
  assumes p-loop: p ∈ 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 ∩ V) x0) WordNil)
    (partition-word p (t # u # ts) (b # bs))
proof –
  have svk: svk-partition p (t # u # ts) (b # bs)
    using part by (rule valid-partition-cases(2))
  have tail-rel:
    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 ∩ V) x0) WordNil)
      (partition-word-with-tail p (t # u # ts) (b # bs) WordNil)
    by (rule partition-word-with-tail-respects[OF p-loop svk bridge-word-one-equiv])
  simp
  then show ?thesis
    by simp
qed

```

```

lemma bridge-loop-constant-one:
  fixes h :: (real × real) ⇒ 'a
  assumes h-cont: continuous-map (top-of-set ({0..1} × {0..1})) (top-of-set W)
  h

```

```

    and t01: t ∈ {0..1}
    and c01: c ∈ {0..1}
    and d01: d ∈ {0..1}
    and cd: c ≤ d
    and const: ∀ y ∈ {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 -
  have edgeUV: h ‘ ({t} × {c..d}) ⊆ U ∩ V
    using const x0-in-UV by auto
  have bridge-loop-in: bridge-loop h t c d ∈ loop-space (U ∩ V) x0
    by (rule vertical-bridge-loop-in-set[OF h-cont t01 c01 d01 cd edgeUV]) simp
  have hc: h (t, c) = x0 and hd: h (t, d) = x0
    using const cd by auto
  have conn-c-img: path-image (connector (h (t, c))) ⊆ {x0}
    using hc by (auto simp: path-image-def connector-def)
  have conn-d-img: path-image (reversepath (connector (h (t, d)))) ⊆ {x0}
    using hd by (auto simp: path-image-def connector-def reversepath-def)
  have vert-img: path-image (vertical-strip-path h t c d) ⊆ {x0}
proof -
  have vert-img-raw: vertical-strip-path h t c d ‘ {0..1} ⊆ {x0}
proof
  fix x
  assume x ∈ vertical-strip-path h t c d ‘ {0..1}
  then obtain u where u01: u ∈ {0..1} and x-eq: x = vertical-strip-path h t
c d u
    by blast
  have (d - c) * u + c ∈ {c..d}
    by (rule affine-subinterval-member[OF cd u01])
  then show x ∈ {x0}
    using const x-eq by (auto simp: vertical-strip-path-def)
qed
from vert-img-raw show ?thesis
  by (simp add: path-image-def)
qed
have join-img:
  path-image (connector (h (t, c)) +++ vertical-strip-path h t c d) ⊆ {x0}
  by (rule subset-path-image-join[OF conn-c-img vert-img])
have bridge-img-single: path-image (bridge-loop h t c d) ⊆ {x0}
  unfolding bridge-loop-def by (rule subset-path-image-join[OF join-img
conn-d-img])
have bridge-path: path (bridge-loop h t c d)
  and bridge-img: path-image (bridge-loop h t c d) ⊆ U ∩ V
  using bridge-loop-in unfolding loop-space-def by auto
have bridge-const:
  homotopic-paths (U ∩ V) (bridge-loop h t c d) (λ-. x0)
proof (rule homotopic-paths-eq[OF bridge-path bridge-img])
  fix u :: real
  assume u01: u ∈ {0..1}

```

```

then have bridge-loop h t c d u ∈ path-image (bridge-loop h t c d)
  by (auto simp: path-image-def)
then show bridge-loop h t c d u = x0
  using bridge-imp-single by auto
qed
show ?thesis
  unfolding fundamental-group-one-def
  by (rule loop-class-eqI[OF bridge-loop-in constant-loop-in-space[OF x0-in-UV]
bridge-const])
qed

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 ∈ {0..1}
  and d01: d ∈ {0..1}
  and cd: c ≤ d
  and part: rectangle-partition h c d ts bs
  and ts-ne: ts ≠ []
  and hd0: hd ts = 0
  and edgeUV:  $\bigwedge x. x \in \text{set } ts \implies h \text{ ` } (\{x\} \times \{c..d\}) \subseteq U \cap V$ 
  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 ( $\lambda x. h (x, c)$ ) ts bs)
    (partition-word ( $\lambda x. h (x, d)$ ) ts bs)
proof -
  have rowc-loop: ( $\lambda x. h (x, c)$ ) ∈ loop-space W x0
    by (rule homotopy-row-in-loop-space[OF h-cont c01]) (use end0 end1 cd in
auto)
  have rowd-loop: ( $\lambda x. h (x, d)$ ) ∈ loop-space W x0
    by (rule homotopy-row-in-loop-space[OF h-cont d01]) (use end0 end1 cd in
auto)
  have valid-c: valid-partition ( $\lambda x. h (x, c)$ ) ts bs
    by (rule rectangle-partition-valid-partition-row[OF part edgeUV - ts-ne hd0])
(use c01 cd in auto)
  have valid-d: valid-partition ( $\lambda x. h (x, d)$ ) ts bs
    by (rule rectangle-partition-valid-partition-row[OF part edgeUV - ts-ne hd0])
(use d01 cd in auto)
  obtain t ts' where ts: ts = t # ts'
    using ts-ne by (cases ts) auto
  have valid-c-ts: valid-partition ( $\lambda x. h (x, c)$ ) ts bs
    using valid-c by simp
  have t0: t = 0
    using valid-c-ts unfolding ts by (rule valid-partition-cases(1))
  have svk-c: svk-partition ( $\lambda x. h (x, c)$ ) (t # ts') bs
    using valid-c-ts unfolding ts by (rule valid-partition-cases(2))
  have ts'-ne: ts' ≠ []

```

```

proof
  assume  $ts' = []$ 
  with svk-c t0 show False
  by (cases bs) auto
qed
obtain  $u\ us$  where  $ts': ts' = u \# us$ 
  using  $ts'-ne$  by (cases ts') auto
obtain  $b\ bs'$  where  $bs: bs = b \# bs'$ 
  using svk-c ts' by (cases bs) auto
have part-shape: rectangle-partition  $h\ c\ d\ (t \# u \# us)\ (b \# bs')$ 
  using part unfolding  $ts\ ts'\ bs$  by simp
have last1:  $last\ (t \# u \# us) = 1$ 
  using valid-partition-last-props[OF valid-c] unfolding  $ts\ ts'$  by simp
have rowd-in:
  fpw-in-space  $G1\ G2\ (partition-word\ (\lambda x. h\ (x, d))\ (t \# u \# us)\ (b \# bs'))$ 
  by (rule valid-partition-partition-word-in-space[OF rowd-loop]) (use valid-d ts
ts' bs in simp)
have edgeUV-shape:
   $\bigwedge x. x \in set\ (t \# u \# us) \implies h\ '(\{x\} \times \{c..d\}) \subseteq U \cap V$ 
  using edgeUV unfolding  $ts\ ts'$  by simp
have wordnil-in: fpw-in-space  $G1\ G2\ WordNil$ 
  by simp
have rect-rel-raw:
  carrier-full-amalg-equiv  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$ 
  (partition-word-with-tail  $(\lambda x. h\ (x, c))\ (t \# u \# us)\ (b \# bs')$ )
  (bridge-word  $(last\ (b \# bs'))$ )
  (loop-class  $(U \cap V)\ x0\ (bridge-loop\ h\ (last\ (t \# u \# us))\ c\ d))\ WordNil$ )
  (bridge-word  $b\ (loop-class\ (U \cap V)\ x0\ (bridge-loop\ h\ t\ c\ d))$ )
  (partition-word-with-tail  $(\lambda x. h\ (x, d))\ (t \# u \# us)\ (b \# bs')\ WordNil$ )
  by (rule rectangle-partition-partition-word-with-tail-equiv[OF h-cont c01 d01 cd
part-shape edgeUV-shape wordnil-in])
have rect-rel:
  carrier-full-amalg-equiv  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$ 
  (partition-word-with-tail  $(\lambda x. h\ (x, c))\ (t \# u \# us)\ (b \# bs')$ )
  (bridge-word  $(last\ (b \# bs'))$ )
  (loop-class  $(U \cap V)\ x0\ (bridge-loop\ h\ (last\ (t \# u \# us))\ c\ d))\ WordNil$ )
  (bridge-word  $b\ (loop-class\ (U \cap V)\ x0\ (bridge-loop\ h\ t\ c\ d))$ )
  (partition-word  $(\lambda x. h\ (x, d))\ (t \# u \# us)\ (b \# bs'))$ )
  using rect-rel-raw by simp
have start-one:
  loop-class  $(U \cap V)\ x0\ (bridge-loop\ h\ t\ c\ d) = fundamental-group-one\ (U \cap V)$ 
x0
  by (subst t0, rule bridge-loop-constant-one[OF h-cont]) (use c01 d01 cd end0
in auto)
have end-one:
  loop-class  $(U \cap V)\ x0\ (bridge-loop\ h\ (last\ (t \# u \# us))\ c\ d) =$ 
fundamental-group-one  $(U \cap V)\ x0$ 
  by (subst last1, rule bridge-loop-constant-one[OF h-cont]) (use c01 d01 cd end1
in auto)

```

```

have rect-rel':
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (bridge-word (last ( $b \# bs'$ )) (fundamental-group-one ( $U \cap V$ )  $x0$ ) WordNil)
  (bridge-word  $b$  (fundamental-group-one ( $U \cap V$ )  $x0$ )
    (partition-word ( $\lambda x. h(x, d)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ )))
proof (cases b)
  case True
  then show ?thesis
    using rect-rel start-one end-one by simp
next
  case False
  then show ?thesis
    using rect-rel start-one end-one by simp
qed
have tail-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (bridge-word (last ( $b \# bs'$ )) (fundamental-group-one ( $U \cap V$ )  $x0$ ) WordNil)
  (partition-word ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  by (rule valid-partition-tail-bridge-one-equiv[OF rowc-loop]) (use valid-c ts ts'
bs in simp)
have prefix-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (bridge-word  $b$  (fundamental-group-one ( $U \cap V$ )  $x0$ )
    (partition-word ( $\lambda x. h(x, d)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ )))
  (partition-word ( $\lambda x. h(x, d)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  by (rule bridge-word-one-equiv[OF rowd-in])
have step1:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (partition-word-with-tail ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (bridge-word (last ( $b \# bs'$ )) (fundamental-group-one ( $U \cap V$ )  $x0$ ) WordNil)
  by (rule carrier-full-amalg-equiv.sym[OF tail-rel])
have step2:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (bridge-word  $b$  (fundamental-group-one ( $U \cap V$ )  $x0$ )
    (partition-word ( $\lambda x. h(x, d)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ )))
  by (rule carrier-full-amalg-equiv.trans[OF step1 rect-rel'])
have step3:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word ( $\lambda x. h(x, c)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  (partition-word ( $\lambda x. h(x, d)$ ) ( $t \# u \# us$ ) ( $b \# bs'$ ))
  by (rule carrier-full-amalg-equiv.trans[OF step2 prefix-rel])
then show ?thesis
  using ts ts' bs by simp
qed

```

**lemma** *valid-partition-nearby-partition-word-equiv*:  
**fixes**  $h :: (\text{real} \times \text{real}) \Rightarrow 'a$   
**assumes**  $h\text{-cont}$ : *continuous-map* (*top-of-set* ( $\{0..1\} \times \{0..1\}$ )) (*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**  $y0\text{-}01$ :  $y0 \in \{0..1\}$   
**and**  $part0$ : *valid-partition* ( $\lambda x. h(x, y0)$ )  $ts$   $bs$   
**shows**  $\exists e > 0. \forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow$   
*valid-partition* ( $\lambda x. h(x, z)$ )  $ts$   $bs \wedge$   
*carrier-full-amalg-equiv*  $G1$   $G2$   $H$   $i1$   $i2$   $mult1$   $one1$   $mult2$   $one2$   
(*partition-word* ( $\lambda x. h(x, y0)$ )  $ts$   $bs$ )  
(*partition-word* ( $\lambda x. h(x, z)$ )  $ts$   $bs$ )  
**proof** –  
**obtain**  $t$   $ts'$  **where**  $ts$ :  $ts = t \# ts'$   
**using** *valid-partition-hd(1)*[*OF*  $part0$ ] **by** (*cases*  $ts$ ) *auto*  
**have**  $hd\text{-}ts0$ :  $hd\ ts = 0$   
**by** (*rule* *valid-partition-hd(2)*)[*OF*  $part0$ ]  
**have**  $valid\text{-}ts$ : *valid-partition* ( $\lambda x. h(x, y0)$ )  $ts$   $bs$   
**using**  $part0$  **by** *simp*  
**have**  $t0$ :  $t = 0$   
**using** *valid-ts unfolding*  $ts$  **by** (*rule* *valid-partition-cases(1)*)  
**have**  $svk0$ : *svk-partition* ( $\lambda x. h(x, y0)$ ) ( $t \# ts'$ )  $bs$   
**using** *valid-ts unfolding*  $ts$  **by** (*rule* *valid-partition-cases(2)*)  
**have**  $local\text{-}neigh$ :  
 $\exists N. \text{openin}$  (*top-of-set*  $\{0..1\}$ )  $N \wedge y0 \in N \wedge$   
 $(\forall x \in \text{set } (t \# ts'). h'(\{x\} \times N) \subseteq U \cap V) \wedge$   
 $(\forall y\ z. y \leq z \longrightarrow \{y..z\} \subseteq N \longrightarrow \text{rectangle-partition } h\ y\ z\ (t \# ts')\ bs)$   
**by** (*rule* *svk-partition-local-neighbourhood*)[*OF*  $h\text{-cont}$   $y0\text{-}01$   $svk0$ ]  
**obtain**  $N$  **where**  $N\text{-open}$ : *openin* (*top-of-set*  $\{0..1\}$ )  $N$   
**and**  $y0N$ :  $y0 \in N$   
**and**  $pointN$ :  $\forall x \in \text{set } ts. h'(\{x\} \times N) \subseteq U \cap V$   
**and**  $rectN$ :  $\forall y\ z. y \leq z \longrightarrow \{y..z\} \subseteq N \longrightarrow \text{rectangle-partition } h\ y\ z\ ts\ bs$   
**using**  $local\text{-}neigh$  *unfolding*  $ts$  **by** *blast*  
**from** *openin-euclidean-subtopology-iff*[*THEN* *iffD1*, *OF*  $N\text{-open}$ ]  $y0N$   
**obtain**  $e$  **where**  $e\text{-pos}$ :  $e > 0$   
**and**  $e\text{-ball}$ :  $\forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow z \in N$   
**by** *blast*  
**show** *?thesis*  
**proof** (*intro* *exI* *conjI* *ballI* *impI*)  
**show**  $e > 0$   
**by** (*rule*  $e\text{-pos}$ )  
**fix**  $z$   
**assume**  $z01$ :  $z \in \{0..1\}$   
**and**  $dz$ :  $\text{dist } z \ y0 < e$   
**have**  $zN$ :  $z \in N$   
**by** (*rule*  $e\text{-ball}$ )[*rule-format*, *OF*  $z01$   $dz$ ]  
**have**  $dyz$ :  $\text{dist } y0\ z < e$   
**using**  $dz$  **by** (*simp* *add*: *dist-commute*)

```

have seg-ball: closed-segment  $y_0 z \subseteq \text{ball } y_0 e$ 
  by (rule closed-segment-subset) (use y0-01 z01 e-pos dyz in auto)
have seg-unit: closed-segment  $y_0 z \subseteq \{0..1\}$ 
  by (rule closed-segment-subset) (use y0-01 z01 in auto)
have segN: closed-segment  $y_0 z \subseteq N$ 
proof
  fix w
  assume wseg:  $w \in \text{closed-segment } y_0 z$ 
  have w01:  $w \in \{0..1\}$ 
    using seg-unit wseg by auto
  have w-ball:  $w \in \text{ball } y_0 e$ 
    using seg-ball wseg by auto
  have dist w y0 < e
    using w-ball unfolding ball-def by (simp add: dist-commute)
  then show  $w \in N$ 
    by (rule e-ball[rule-format, OF w01])
qed
have intervalN:  $\{\min y_0 z.. \max y_0 z\} \subseteq N$ 
proof
  fix y
  assume y-in:  $y \in \{\min y_0 z.. \max y_0 z\}$ 
  have  $y \in \text{closed-segment } y_0 z$ 
  proof (cases  $y_0 \leq z$ )
    case True
      with y-in show ?thesis
        by (simp add: closed-segment-eq-real-ivl)
    next
      case False
        with y-in show ?thesis
          by (simp add: closed-segment-eq-real-ivl)
  qed
  then show  $y \in N$ 
    using segN by blast
qed
have edgeUV-interval:
   $h \text{ ' } (\{x\} \times \{\min y_0 z.. \max y_0 z\}) \subseteq U \cap V$  if x-in:  $x \in \text{set } ts$  for x
proof –
  have edgeN:  $h \text{ ' } (\{x\} \times N) \subseteq U \cap V$ 
    using pointN x-in by auto
  have subset-x:  $\{x\} \times \{\min y_0 z.. \max y_0 z\} \subseteq \{x\} \times N$ 
    using intervalN by auto
  show ?thesis
    by (rule order-trans[OF image-mono[OF subset-x] edgeN])
qed
have rect:
  rectangle-partition  $h (\min y_0 z) (\max y_0 z) ts bs$ 
  by (rule rectN[rule-format]) (use intervalN in auto)
have valid-z: valid-partition  $(\lambda x. h (x, z)) ts bs$ 
  by (rule rectangle-partition-valid-partition-row[OF rect edgeUV-interval -

```

```

valid-partition-hd(1)[OF part0] hd-ts0)
  auto
  have min01: min y0 z ∈ {0..1}
  using y0-01 z01 by auto
  have max01: max y0 z ∈ {0..1}
  using y0-01 z01 by auto
  have interval01: {min y0 z..max y0 z} ⊆ {0..1}
  using y0-01 z01 by auto
  have end0-interval: ∀ y∈{min y0 z..max y0 z}. h (0, y) = x0
  proof
    fix y
    assume y-in: y ∈ {min y0 z..max y0 z}
    then have y01: y ∈ {0..1}
      using interval01 by blast
    show h (0, y) = x0
      using end0 y01 by blast
  qed
  have end1-interval: ∀ y∈{min y0 z..max y0 z}. h (1, y) = x0
  proof
    fix y
    assume y-in: y ∈ {min y0 z..max y0 z}
    then have y01: y ∈ {0..1}
      using interval01 by blast
    show h (1, y) = x0
      using end1 y01 by blast
  qed
  have part-rel-raw:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word (λx. h (x, min y0 z)) ts bs)
      (partition-word (λx. h (x, max y0 z)) ts bs)
  by (rule rectangle-partition-partition-word-equiv[
    OF h-cont min01 max01 - rect valid-partition-hd(1)[OF part0] hd-ts0
    edgeUV-interval end0-interval end1-interval])
  simp
  have part-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word (λx. h (x, y0)) ts bs)
      (partition-word (λx. h (x, z)) ts bs)
  proof (cases y0 ≤ z)
    case True
    then show ?thesis
      using part-rel-raw by simp
  next
    case False
    then have carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (partition-word (λx. h (x, z)) ts bs)
      (partition-word (λx. h (x, y0)) ts bs)
    using part-rel-raw by simp
    then show ?thesis

```

```

    by (rule carrier-full-amalg-equiv.sym)
  qed
  show valid-partition ( $\lambda x. h (x, z)$ ) ts bs
    using valid-z by simp
  show carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word ( $\lambda x. h (x, y0)$ ) ts bs)
    (partition-word ( $\lambda x. h (x, z)$ ) ts bs)
    using part-rel by simp
  qed
  qed

```

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

**lemma** *some-valid-partition-spec*:  
**assumes** p-loop:  $p \in \text{loop-space } W \ x0$   
**shows** valid-partition p  
 (fst (some-valid-partition p))  
 (snd (some-valid-partition p))  
**proof** –  
**have** ex:  $\exists$  tb. case tb of (ts, bs)  $\Rightarrow$  valid-partition p ts bs  
 using loop-has-valid-partition[OF p-loop] by force  
**have** case some-valid-partition p of (ts, bs)  $\Rightarrow$  valid-partition p ts bs  
 unfolding some-valid-partition-def by (rule someI-ex[OF ex])  
**then show** ?thesis  
 by (cases some-valid-partition p) auto  
 qed

### 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:  $p \in \text{loop-space } W \ x0$   
 and q-loop:  $q \in \text{loop-space } W \ x0$   
 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** –  
**obtain** h0 ::  $real \times real \Rightarrow 'a$  **where** h0-cont: *continuous-on* ( $\{0..1\} \times \{0..1\}$ )  
 h0  
 and h0-into:  $h0 \in (\{0..1\} \times \{0..1\}) \rightarrow W$   
 and h0-p:  $\bigwedge x. h0 (0, x) = p \ x$   
 and h0-q:  $\bigwedge x. h0 (1, x) = q \ x$

```

and h0-left:  $\bigwedge y. y \in \{0..1\} \implies h0 (y, 0) = p\ 0$ 
and h0-right:  $\bigwedge y. y \in \{0..1\} \implies h0 (y, 1) = p\ 1$ 
using pq
by (auto simp: homotopic-paths-def homotopic-with-def
      pathstart-def pathfinish-def image-subset-iff-funcset)
define h where h = ( $\lambda xy. h0 (snd\ xy, fst\ xy)$ )
let ?row =  $\lambda y. \lambda x. h (x, y)$ 
let ?enc =  $\lambda y. case\ some\ valid\ partition\ (?row\ y)\ of\ (vs, ds) \Rightarrow partition\ word$ 
(?row y vs ds)
let ?P =
  { $y \in \{0..1\}. carrier\ full\ amalg\ equiv\ G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2$ }
(?enc 0) (?enc y)}
have p-start: pathstart p = x0 and p-finish: pathfinish p = x0
using p-loop unfolding loop-space-def by auto
have h-cont: continuous-map (top-of-set ( $\{0..1\} \times \{0..1\}$ )) (top-of-set W) h
proof –
  have swap-cont:
    continuous-on ( $\{0..1\} \times \{0..1\}$ ) ( $\lambda xy. (snd\ xy, fst\ xy)$ )
  by (intro continuous-intros)
  have swap-in:
    ( $\lambda xy. (snd\ xy, fst\ xy)$ ) ‘ ( $\{0..1\} \times \{0..1\}$ )  $\subseteq$  ( $\{0..1\} \times \{0..1\}$ )
  by auto
  have h-on: continuous-on ( $\{0..1\} \times \{0..1\}$ ) h
proof –
  have continuous-on ( $\{0..1\} \times \{0..1\}$ ) ( $\lambda xy. h0 (snd\ xy, fst\ xy)$ )
  by (rule continuous-on-compose2[OF h0-cont]) (use swap-cont swap-in in
auto)
  then show ?thesis
    unfolding h-def .
qed
have h-into:  $h \in (\{0..1\} \times \{0..1\}) \rightarrow W$ 
using h0-into unfolding h-def by auto
show ?thesis
using h-on h-into by simp
qed
have end0:  $\forall y \in \{0..1\}. h (0, y) = x0$ 
proof
  fix y :: real
  assume y01:  $y \in \{0..1\}$ 
  have h0 (y, 0::real) = p 0
  by (rule h0-left[OF y01])
  then show  $h (0, y) = x0$ 
  using p-start unfolding h-def pathstart-def by simp
qed
have end1:  $\forall y \in \{0..1\}. h (1, y) = x0$ 
proof
  fix y :: real
  assume y01:  $y \in \{0..1\}$ 
  have h0 (y, 1::real) = p 1

```

```

    by (rule h0-right[OF y01])
  then show  $h(1, y) = x0$ 
    using p-finish unfolding h-def pathfinish-def by simp
qed
have row-loop: ?row  $y \in \text{loop-space } W x0$  if  $y01: y \in \{0..1\}$  for  $y$ 
  by (rule homotopy-row-in-loop-space[OF h-cont y01]) (use end0 end1 y01 in
auto)
have openP-local:  $\forall y0 \in ?P. \exists e > 0. \forall z \in \{0..1\}. \text{dist } z y0 < e \longrightarrow z \in ?P$ 
proof
  fix y0
  assume y0P:  $y0 \in ?P$ 
  then have y0-01:  $y0 \in \{0..1\}$ 
    by simp
  obtain vs ds where tb0: some-valid-partition (?row y0) = (vs, ds)
    by (cases some-valid-partition (?row y0)) auto
  have part0: valid-partition (?row y0) vs ds
    using some-valid-partition-spec[OF row-loop[OF y0-01]] unfolding tb0 by
simp
  obtain e where e-pos:  $e > 0$ 
  and near:
     $\forall z \in \{0..1\}. \text{dist } z y0 < e \longrightarrow$ 
    valid-partition (?row z) vs ds  $\wedge$ 
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (?row y0) vs ds)
    (partition-word (?row z) vs ds)
  using valid-partition-nearby-partition-word-equiv[OF h-cont end0 end1 y0-01
part0]
  by blast
  have nearP:  $\forall z \in \{0..1\}. \text{dist } z y0 < e \longrightarrow z \in ?P$ 
proof
  fix z :: real
  assume z01:  $z \in \{0..1\}$ 
  show  $\text{dist } z y0 < e \longrightarrow z \in ?P$ 
proof
  assume dz:  $\text{dist } z y0 < e$ 
  have z-part: valid-partition (?row z) vs ds
  and fixed-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (?row y0) vs ds)
    (partition-word (?row z) vs ds)
  using near[rule-format, OF z01 dz] by blast+
  obtain us' cs' where tbz: some-valid-partition (?row z) = (us', cs')
    by (cases some-valid-partition (?row z)) auto
  have chosen-z: valid-partition (?row z) us' cs'
    using some-valid-partition-spec[OF row-loop[OF z01]] unfolding tbz by
simp
  have z-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (?row z) vs ds) (?enc z)

```

```

    using valid-partition-same-loop-partition-word-equiv[OF row-loop[OF z01]
z-part chosen-z]
    unfolding tbz by simp
    have y0-eq: ?enc y0 = partition-word (?row y0) vs ds
    unfolding tb0 by simp
    have base-rel0:
      carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
        (?enc 0) (?enc y0)
    using y0P by simp
    have base-rel:
      carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
        (?enc 0) (partition-word (?row y0) vs ds)
    using base-rel0 unfolding y0-eq by simp
    have carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
      (?enc 0) (?enc z)
    by (rule carrier-full-amalg-equiv.trans[OF base-rel])
      (rule carrier-full-amalg-equiv.trans[OF fixed-rel z-rel])
    then show z ∈ ?P
      using z01 by simp
  qed
qed
have witness: e > 0 ∧ (∀ z ∈ {0..1}. dist z y0 < e ⟶ z ∈ ?P)
proof
  show e > 0
    by (rule e-pos)
  show ∀ z ∈ {0..1}. dist z y0 < e ⟶ z ∈ ?P
    by (rule nearP)
qed
have ex-witness: ∃ eps. eps > 0 ∧ (∀ z ∈ {0..1}. dist z y0 < eps ⟶ z ∈ ?P)
proof (rule exI[where x = e], rule conjI)
  show e > 0
    using witness by simp
  show ∀ z ∈ {0..1}. dist z y0 < e ⟶ z ∈ ?P
    using witness by simp
qed
have ex-witness-set: ∃ eps. eps ∈ {0<..} ∧ (∀ z ∈ {0..1}. dist z y0 < eps ⟶ z
∈ ?P)
proof –
  from ex-witness obtain eps where
    eps-pos: eps > 0 and eps-near: ∀ z ∈ {0..1}. dist z y0 < eps ⟶ z ∈ ?P
  by auto
  show ?thesis
proof (rule exI[where x = eps], rule conjI)
  show eps ∈ {0<..}
    using eps-pos by simp
  show ∀ z ∈ {0..1}. dist z y0 < eps ⟶ z ∈ ?P
    by (rule eps-near)
qed
qed

```

```

show  $\exists e > 0. \forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow z \in ?P$ 
proof –
  from ex-witness
  show ?thesis
    unfolding Bex-def greaterThan-def
    by blast
  qed
qed
have P-subset: ?P  $\subseteq \{0..1\}$ 
  by auto
have openP: openin (top-of-set {0..1}) ?P
  using openP-local P-subset by (auto simp: openin-euclidean-subtopology-iff)
have open-notP-local:
   $\forall y0 \in \{0..1\} - ?P. \exists e > 0. \forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow z \in \{0..1\} - ?P$ 
proof
  fix y0
  assume y0-notP: y0  $\in \{0..1\} - ?P$ 
  then have y0-01: y0  $\in \{0..1\}$  and y0-not: y0  $\notin ?P$ 
    by auto
  obtain vs ds where tb0: some-valid-partition (?row y0) = (vs, ds)
    by (cases some-valid-partition (?row y0)) auto
  have part0: valid-partition (?row y0) vs ds
    using some-valid-partition-spec[OF row-loop[OF y0-01]] unfolding tb0 by
simp
  obtain e where e-pos: e > 0
  and near:
     $\forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow$ 
    valid-partition (?row z) vs ds  $\wedge$ 
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (?row y0) vs ds)
    (partition-word (?row z) vs ds)
  using valid-partition-nearby-partition-word-equiv[OF h-cont end0 end1 y0-01
part0]
  by blast
have near-notP:  $\forall z \in \{0..1\}. \text{dist } z \ y0 < e \longrightarrow z \in \{0..1\} - ?P$ 
proof
  fix z :: real
  assume z01: z  $\in \{0..1\}$ 
  show dist z y0 < e  $\longrightarrow z \in \{0..1\} - ?P$ 
proof
  assume dz: dist z y0 < e
  have z-part: valid-partition (?row z) vs ds
  and fixed-rel:
    carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
    (partition-word (?row y0) vs ds)
    (partition-word (?row z) vs ds)
  using near[rule-format, OF z01 dz] by blast+
  obtain us' cs' where tbz: some-valid-partition (?row z) = (us', cs')
    by (cases some-valid-partition (?row z)) auto

```

```

have chosen-z: valid-partition (?row z) us' cs'
  using some-valid-partition-spec[OF row-loop[OF z01]] unfolding tbz by
simp
have z-rel:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word (?row z) vs ds) (?enc z)
  using valid-partition-same-loop-partition-word-equiv[OF row-loop[OF z01]
z-part chosen-z]
  unfolding tbz by simp
have y0-eq: ?enc y0 = partition-word (?row y0) vs ds
  unfolding tb0 by simp
show z ∈ {0..1} – ?P
proof
  show z ∈ {0..1}
    by (rule z01)
  show z ∉ ?P
    proof
      assume zP: z ∈ ?P
      have base-to-z:
        carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (?enc 0)
(?enc z)
        using zP by simp
      have z-to-y0:
        carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (?enc z)
(?enc y0)
      proof –
        have step1:
          carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
          (?enc z) (partition-word (?row z) vs ds)
          by (rule carrier-full-amalg-equiv.sym[OF z-rel])
        show ?thesis
          unfolding y0-eq
          by (rule carrier-full-amalg-equiv.trans[OF step1])
            (rule carrier-full-amalg-equiv.sym[OF fixed-rel])
        qed
      have carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (?enc
0) (?enc y0)
        by (rule carrier-full-amalg-equiv.trans[OF base-to-z z-to-y0])
      then show False
        using y0-not y0-01 by simp
      qed
    qed
  qed
qed
have witness: e > 0 ∧ (∀ z∈{0..1}. dist z y0 < e → z ∈ {0..1} – ?P)
proof
  show e > 0
    by (rule e-pos)
  show ∀ z∈{0..1}. dist z y0 < e → z ∈ {0..1} – ?P

```

```

    by (rule near-notP)
  qed
  have ex-witness:  $\exists eps. eps > 0 \wedge (\forall z \in \{0..1\}. dist\ z\ y0 < eps \longrightarrow z \in \{0..1\})$ 
- ?P)
  proof (rule exI[where x = e], rule conjI)
    show  $e > 0$ 
    using witness by simp
    show  $\forall z \in \{0..1\}. dist\ z\ y0 < e \longrightarrow z \in \{0..1\}$  - ?P
    using witness by simp
  qed
  have ex-witness-set:  $\exists eps. eps \in \{0<..\} \wedge (\forall z \in \{0..1\}. dist\ z\ y0 < eps \longrightarrow z \in \{0..1\})$  - ?P)
- ?P)
  proof -
    from ex-witness obtain eps where
      eps-pos:  $eps > 0$  and eps-near:  $\forall z \in \{0..1\}. dist\ z\ y0 < eps \longrightarrow z \in \{0..1\}$ 
- ?P)
    by auto
  show ?thesis
  proof (rule exI[where x = eps], rule conjI)
    show  $eps \in \{0<..\}$ 
    using eps-pos by simp
    show  $\forall z \in \{0..1\}. dist\ z\ y0 < eps \longrightarrow z \in \{0..1\}$  - ?P
    by (rule eps-near)
  qed
  qed
  show  $\exists e > 0. \forall z \in \{0..1\}. dist\ z\ y0 < e \longrightarrow z \in \{0..1\}$  - ?P
  proof -
    from ex-witness
  show ?thesis
    unfolding Bex-def greaterThan-def
  by blast
  qed
  qed
  have open-notP:  $openin\ (top-of-set\ \{0..1\})\ (\{0..1\})$  - ?P)
  using open-notP-local by (auto simp: openin-euclidean-subtopology-iff)
  have closedP:  $closedin\ (top-of-set\ \{0..1\})\ ?P$ 
  proof -
    have  $\{0..1\} - (\{0..1\} - ?P) = ?P$ 
    using P-subset by auto
    with open-notP show ?thesis
    by (auto simp: openin-closedin-eq)
  qed
  have P-all:  $?P = \{0..1\}$ 
  proof -
    have connected ( $\{0..1\}$  :: real set)
    by simp
    then have P-cases:  $?P = \{\} \vee ?P = \{0..1\}$ 
    using openP closedP unfolding connected-clopen by blast
    have ( $0::real$ )  $\in ?P$ 

```

```

    by (auto intro: carrier-full-amalg-equiv.refl)
  then have P-nonempty: ?P ≠ {}
    by blast
  from P-cases P-nonempty show ?thesis
    by blast
qed
have row0-eq: ?row 0 = p
proof
  fix x :: real
  show ?row 0 x = p x
    using h0-p[of x] unfolding h-def by simp
qed
have row1-eq: ?row 1 = q
proof
  fix x :: real
  show ?row 1 x = q x
    using h0-q[of x] unfolding h-def by simp
qed
have row0-hom: homotopic-paths W (?row 0) p
proof -
  have row0-path: path (?row 0) and row0-img: path-image (?row 0) ⊆ W
    using row-loop[of 0] unfolding loop-space-def by auto
  show ?thesis
  proof (rule homotopic-paths-eq[OF row0-path row0-img])
    fix t :: real
    assume t01: t ∈ {0..1}
    show ?row 0 t = p t
    proof -
      from row0-eq have ?row 0 t = p t
        by (rule fun-cong)
      then show ?thesis .
    qed
  qed
qed
have row1-hom: homotopic-paths W (?row 1) q
proof -
  have row1-path: path (?row 1) and row1-img: path-image (?row 1) ⊆ W
    using row-loop[of 1] unfolding loop-space-def by auto
  show ?thesis
  proof (rule homotopic-paths-eq[OF row1-path row1-img])
    fix t :: real
    assume t01: t ∈ {0..1}
    show ?row 1 t = q t
    proof -
      from row1-eq have ?row 1 t = q t
        by (rule fun-cong)
      then show ?thesis .
    qed
  qed
qed

```

```

qed
obtain vs0 ds0 where tb0: some-valid-partition (?row 0) = (vs0, ds0)
  by (cases some-valid-partition (?row 0)) auto
obtain vs1 ds1 where tb1: some-valid-partition (?row 1) = (vs1, ds1)
  by (cases some-valid-partition (?row 1)) auto
have tb0-p: some-valid-partition p = (vs0, ds0)
  using tb0 row0-eq by simp
have tb1-q: some-valid-partition q = (vs1, ds1)
  using tb1 row1-eq by simp
have enc0-eq: ?enc 0 = partition-word p vs0 ds0
  using tb0 row0-eq by simp
have enc1-eq: ?enc 1 = partition-word q vs1 ds1
  using tb1 row1-eq by simp
have part-row0: valid-partition p vs0 ds0
  using some-valid-partition-spec[OF p-loop] unfolding tb0-p by simp
have part-row1: valid-partition q vs1 ds1
  using some-valid-partition-spec[OF q-loop] unfolding tb1-q by simp
have p-to-row0:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (partition-word p ts bs) (?enc 0)
  using valid-partition-same-loop-partition-word-equiv[OF p-loop p-part
part-row0]
  using enc0-eq by simp
have row1-to-q:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2
  (?enc 1) (partition-word q us cs)
  using valid-partition-same-loop-partition-word-equiv[OF q-loop part-row1 q-part]
  using enc1-eq by simp
have row0-to-row1:
  carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (?enc 0) (?enc
1)
proof –
  have (1::real)  $\in$  ?P
  using P-all by simp
  then show ?thesis
  by simp
qed
show ?thesis
  by (rule carrier-full-amalg-equiv.trans[OF p-to-row0])
  (rule carrier-full-amalg-equiv.trans[OF row0-to-row1 row1-to-q])
qed

```

```

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 –
  have hom: homotopic-paths W p q
    using p-loop q-loop eq by (simp add: loop-class-eq-iff)
  show ?thesis
    by (rule valid-partition-homotopic-partition-word-equiv[OF p-loop q-loop hom
p-part q-part])
qed

```

```

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 –
  have w-loop: word-loop w ∈ loop-space W x0
    by (rule word-loop-in-W[OF w-in])
  have w-part:
    valid-partition (word-loop w) (word-partition-times w) (word-partition-bits w)
    by (rule word-loop-valid-partition[OF w-in])
  have decode-part:
    svk-decode (partition-word (word-loop w) (word-partition-times w)
(word-partition-bits w)) =
    loop-class W x0 (word-loop w)
    by (rule valid-partition-decode-partition-word-eq-loop-class[OF w-loop w-part])
  have rel:
    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
    by (rule partition-word-word-loop-equiv[OF w-in])
  have svk-decode (partition-word (word-loop w) (word-partition-times w)
(word-partition-bits w)) =
    svk-decode w
    by (rule svk-decode-respects[OF rel])
  then show ?thesis
    using decode-part by simp
qed

```

## 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 ⇒ 'a) set ⇒ ((real ⇒ 'a) set, (real ⇒ '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*:  $A \in \text{fundamental-group-space } W \ x0$   
**shows** *fpw-in-space*  $G1 \ G2$  (*svk-encode*  $A$ )  
**proof** –  
**have** *p-loop*:  $\text{some-loop } W \ x0 \ A \in \text{loop-space } W \ x0$   
**and** *A-eq*:  $A = \text{loop-class } W \ x0 \ (\text{some-loop } W \ x0 \ A)$   
**using** *some-loop-spec*[*OF* *A-in*] **by** *auto*  
**have** *part*: *valid-partition* ( $\text{some-loop } W \ x0 \ A$ )  
(*fst* (*some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )))  
(*snd* (*some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )))  
**by** (*rule* *some-valid-partition-spec*[*OF* *p-loop*])  
**show** *?thesis*  
**unfolding** *svk-encode-def* *Let-def*  
**using** *valid-partition-partition-word-in-space*[*OF* *p-loop* *part*]  
**by** (*cases* *some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )) *simp-all*  
**qed**

**lemma** *svk-decode-encode*:  
**assumes** *A-in*:  $A \in \text{fundamental-group-space } W \ x0$   
**shows** *svk-decode* (*svk-encode*  $A$ ) =  $A$   
**proof** –  
**have** *p-loop*:  $\text{some-loop } W \ x0 \ A \in \text{loop-space } W \ x0$   
**and** *A-eq*:  $A = \text{loop-class } W \ x0 \ (\text{some-loop } W \ x0 \ A)$   
**using** *some-loop-spec*[*OF* *A-in*] **by** *auto*  
**have** *part*: *valid-partition* ( $\text{some-loop } W \ x0 \ A$ )  
(*fst* (*some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )))  
(*snd* (*some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )))  
**by** (*rule* *some-valid-partition-spec*[*OF* *p-loop*])  
**show** *?thesis*  
**unfolding** *svk-encode-def* *Let-def*  
**using** *valid-partition-decode-partition-word-eq-loop-class*[*OF* *p-loop* *part*] *A-eq*  
**by** (*cases* *some-valid-partition* ( $\text{some-loop } W \ x0 \ A$ )) *simp-all*  
**qed**

**lemma** *svk-encode-decode*:  
**assumes** *w-in*:  $\text{fpw-in-space } G1 \ G2 \ w$   
**shows** *carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
(*svk-encode* (*svk-decode*  $w$ ))  $w$   
**proof** –  
**let** *?p* =  $\text{some-loop } W \ x0 \ (\text{svk-decode } w)$   
**let** *?tb* = *some-valid-partition* *?p*  
**obtain** *ts* *bs* **where** *tb*:  $?tb = (ts, bs)$   
**by** (*cases* *?tb*) *auto*  
**have** *p-loop*:  $?p \in \text{loop-space } W \ x0$   
**and** *p-class*:  $\text{svk-decode } w = \text{loop-class } W \ x0 \ ?p$   
**using** *some-loop-spec*[*OF* *svk-decode-in-space*] **by** *auto*  
**have** *p-part*: *valid-partition*  $?p \ ts \ bs$   
**using** *some-valid-partition-spec*[*OF* *p-loop*] **unfolding** *tb* **by** *simp*

**have** *w-loop*: *word-loop*  $w \in \text{loop-space } W \ x0$   
**by** (*rule word-loop-in-W*[*OF w-in*])  
**have** *w-part*:  
*valid-partition* (*word-loop*  $w$ ) (*word-partition-times*  $w$ ) (*word-partition-bits*  $w$ )  
**by** (*rule word-loop-valid-partition*[*OF w-in*])  
**have** *same-class*: *loop-class*  $W \ x0 \ ?p = \text{loop-class } W \ x0$  (*word-loop*  $w$ )  
**using** *p-class svk-decode-word-loop*[*OF w-in*] **by** *simp*  
**have** *part-rel*:  
*carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
(*partition-word*  $?p \ ts \ bs$ )  
(*partition-word* (*word-loop*  $w$ ) (*word-partition-times*  $w$ ) (*word-partition-bits*  $w$ ))  
**by** (*rule valid-partition-loop-class-partition-word-equiv*[*OF p-loop w-loop same-class p-part w-part*])  
**have** *word-rel*:  
*carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
(*partition-word* (*word-loop*  $w$ ) (*word-partition-times*  $w$ ) (*word-partition-bits*  $w$ ))  $w$   
**by** (*rule partition-word-word-loop-equiv*[*OF w-in*])  
**have** *encode-word*: *svk-encode* (*svk-decode*  $w$ ) = *partition-word*  $?p \ ts \ bs$   
**unfolding** *svk-encode-def* *Let-def* *tb* **by** *simp*  
**show** *?thesis*  
**unfolding** *encode-word*  
**by** (*rule carrier-full-amalg-equiv.trans*[*OF part-rel word-rel*])  
**qed**

**lemma** *svk-decode-surjective*:  
**assumes** *A-in*:  $A \in \text{fundamental-group-space } W \ x0$   
**shows**  $\exists w. \text{fpw-in-space } G1 \ G2 \ w \wedge \text{svk-decode } w = A$   
**using** *svk-encode-in-space*[*OF A-in*] *svk-decode-encode*[*OF A-in*] **by** *blast*

**theorem** *svk-decode-image*:  
*svk-decode* ‘ $\{w. \text{fpw-in-space } G1 \ G2 \ w\}$  = *fundamental-group-space*  $W \ x0$   
**proof**  
**show** *svk-decode* ‘ $\{w. \text{fpw-in-space } G1 \ G2 \ w\} \subseteq \text{fundamental-group-space } W \ x0$   
**using** *svk-decode-in-space* **by** *blast*  
**next**  
**show** *fundamental-group-space*  $W \ x0 \subseteq \text{svk-decode}$  ‘ $\{w. \text{fpw-in-space } G1 \ G2 \ w\}$   
**using** *svk-decode-surjective* **by** *blast*  
**qed**

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

**lemma** *classical-svk-quotient-map-in-space*:

**assumes** *A-in*:  $A \in \text{fundamental-group-space } W \ x0$   
**shows** *classical-svk-quotient-map*  $A \in$   
*carrier-full-amalgamated-free-product-space*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2}$   
*one2*  
**unfolding** *classical-svk-quotient-map-def*  
**by** (*rule carrier-full-amalg-class-in-space*[*OF svk-encode-in-space*[*OF A-in*]])

**lemma** *classical-svk-quotient-map-injective*:  
**assumes** *AB*: *classical-svk-quotient-map*  $A = \text{classical-svk-quotient-map } B$   
**and** *A-in*:  $A \in \text{fundamental-group-space } W \ x0$   
**and** *B-in*:  $B \in \text{fundamental-group-space } W \ x0$   
**shows**  $A = B$   
**proof** –  
**have** *carrier-full-amalg-equiv*  $G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
*(svk-encode*  $A)$  *(svk-encode*  $B)$   
**using** *AB* **unfolding** *classical-svk-quotient-map-def*  
**by** (*simp add: carrier-full-amalg-class-eq-iff*)  
**then have** *svk-decode*  $(\text{svk-encode } A) = \text{svk-decode } (\text{svk-encode } B)$   
**by** (*rule svk-decode-respects*)  
**then show** *?thesis*  
**using** *A-in B-in* **by** (*simp add: svk-decode-encode*)  
**qed**

**lemma** *classical-svk-quotient-map-surjective*:  
**assumes** *Q-in*:  
 $Q \in \text{carrier-full-amalgamated-free-product-space } G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1}$   
 $\text{mult2} \ \text{one2}$   
**shows**  $\exists A \in \text{fundamental-group-space } W \ x0. \text{classical-svk-quotient-map } A = Q$   
**proof** –  
**from** *Q-in* **obtain**  $w$  **where** *w-in*: *fpw-in-space*  $G1 \ G2 \ w$   
**and** *Q-rep*:  $Q = \text{carrier-full-amalg-class } G1 \ G2 \ H \ i1 \ i2 \ \text{mult1} \ \text{one1} \ \text{mult2} \ \text{one2}$   
 $w$   
**unfolding** *carrier-full-amalgamated-free-product-space-def* **by** *auto*  
**have** *A-in*: *svk-decode*  $w \in \text{fundamental-group-space } W \ x0$   
**by** (*rule svk-decode-in-space*)  
**have** *classical-svk-quotient-map*  $(\text{svk-decode } w) = Q$   
**unfolding** *classical-svk-quotient-map-def* *Q-rep*  
**by** (*simp add: carrier-full-amalg-class-eq-iff svk-encode-decode*[*OF w-in*])  
**then show** *?thesis*  
**using** *A-in* **by** *blast*  
**qed**

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*  $(\text{fundamental-group-space } W \ x0)$

```

    (carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2
one2)
  unfolding bij-betw-def
proof
  show inj-on classical-svk-quotient-map (fundamental-group-space W x0)
    unfolding inj-on-def
    using classical-svk-quotient-map-injective by blast
next
  show 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
    show classical-svk-quotient-map ' fundamental-group-space W x0 ⊆
      carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1 mult2
one2
    using classical-svk-quotient-map-in-space by blast
  next
    show carrier-full-amalgamated-free-product-space G1 G2 H i1 i2 mult1 one1
mult2 one2
      ⊆ classical-svk-quotient-map ' fundamental-group-space W x0
    using classical-svk-quotient-map-surjective by blast
  qed
qed
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 (λU.
    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 (λU.
  U ⊆ Inl ' topspace X ∪ Inr ' topspace Y ∧

```

$openin\ X\ \{x.\ Inl\ x \in U\} \wedge$   
 $openin\ Y\ \{y.\ Inr\ y \in U\}$   
**proof** –  
**have**  $inl\text{-}Int: \{x.\ Inl\ x \in S \cap T\} = \{x.\ Inl\ x \in S\} \cap \{x.\ Inl\ x \in T\}$  **for**  $S\ T$   
**by** *auto*  
**have**  $inr\text{-}Int: \{y.\ Inr\ y \in S \cap T\} = \{y.\ Inr\ y \in S\} \cap \{y.\ Inr\ y \in T\}$  **for**  $S\ T$   
**by** *auto*  
**have**  $inl\text{-}Union: \{x.\ Inl\ x \in \bigcup \mathcal{K}\} = (\bigcup U \in \mathcal{K}.\ \{x.\ Inl\ x \in U\})$  **for**  $\mathcal{K}$   
**by** *auto*  
**have**  $inr\text{-}Union: \{y.\ Inr\ y \in \bigcup \mathcal{K}\} = (\bigcup U \in \mathcal{K}.\ \{y.\ Inr\ y \in U\})$  **for**  $\mathcal{K}$   
**by** *auto*  
**show** *?thesis*  
**unfolding** *istopology-def inl-Int inr-Int inl-Union inr-Union*  
**by** *blast*  
**qed**

**lemma** *openin-binary-sum-topology*:  
 $openin\ (binary\text{-}sum\text{-}topology\ X\ Y)\ U \longleftrightarrow$   
 $U \subseteq Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y \wedge$   
 $openin\ X\ \{x.\ Inl\ x \in U\} \wedge$   
 $openin\ Y\ \{y.\ Inr\ y \in U\}$   
**by** (*auto simp: binary-sum-topology-def is-binary-sum-topology*)

**lemma** *topspace-binary-sum-topology [simp]*:  
 $topspace\ (binary\text{-}sum\text{-}topology\ X\ Y) = Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y$   
**proof**  
**have**  $top\text{-}open: openin\ (binary\text{-}sum\text{-}topology\ X\ Y)\ (topspace\ (binary\text{-}sum\text{-}topology\ X\ Y))$   
**by** *simp*  
**show**  $topspace\ (binary\text{-}sum\text{-}topology\ X\ Y) \subseteq Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y$   
**using** *top-open unfolding openin-binary-sum-topology by blast*  
**have**  $eqX: \{x.\ Inl\ x \in Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y\} = topspace\ X$   
**by** *auto*  
**have**  $eqY: \{y.\ Inr\ y \in Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y\} = topspace\ Y$   
**by** *auto*  
**have**  $openin\ (binary\text{-}sum\text{-}topology\ X\ Y)\ (Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y)$   
**unfolding** *openin-binary-sum-topology eqX eqY by simp*  
**then show**  $Inl\ 'topspace\ X \cup Inr\ 'topspace\ Y \subseteq topspace\ (binary\text{-}sum\text{-}topology\ X\ Y)$   
**by** (*rule openin-subset*)  
**qed**

**lemma** *subset-topspace-binary-sum-inl*:  
 $(Inl :: 'a \Rightarrow 'a + 'b)\ 'topspace\ X \subseteq topspace\ (binary\text{-}sum\text{-}topology\ X\ Y)$   
**by** *simp*

**lemma** *subset-topspace-binary-sum-inr*:  
 $(Inr :: 'b \Rightarrow 'a + 'b)\ 'topspace\ Y \subseteq topspace\ (binary\text{-}sum\text{-}topology\ X\ Y)$   
**by** *simp*

**lemma** *continuous-map-binary-sum-inl*:  
*continuous-map X (binary-sum-topology X Y) (Inl :: 'a  $\Rightarrow$  'a + 'b)*  
**proof** –  
**have** *image*: (*Inl :: 'a  $\Rightarrow$  'a + 'b*) ‘ *topspace X  $\subseteq$  topspace (binary-sum-topology X Y)*  
**by** *simp*  
**have** *preimage*:  
 $\forall U. \text{openin } (\text{binary-sum-topology } X \ Y) \ U \longrightarrow \text{openin } X \ \{x \in \text{topspace } X. \text{Inl } x \in U\}$   
**proof** (*intro allI impI*)  
**fix** *U*  
**assume** *U*: *openin (binary-sum-topology X Y) U*  
**have** *openU*: *openin X {x. Inl x  $\in$  U}*  
**using** *U* **by** (*simp add: openin-binary-sum-topology*)  
**have** *subsetX*: *{x. Inl x  $\in$  U}  $\subseteq$  topspace X*  
**using** *openU* **by** (*rule openin-subset*)  
**have** *eq*: *{x  $\in$  topspace X. Inl x  $\in$  U} = {x. Inl x  $\in$  U}*  
**using** *subsetX* **by** *auto*  
**have** *openin X {x  $\in$  topspace X. Inl x  $\in$  U}*  
**unfolding** *eq* **using** *openU* **by** *simp*  
**then show** *openin X {x  $\in$  topspace X. Inl x  $\in$  U} .*  
**qed**  
**from** *image preimage* **show** *?thesis*  
**by** (*simp add: continuous-map*)  
**qed**

**lemma** *continuous-map-binary-sum-inr*:  
*continuous-map Y (binary-sum-topology X Y) (Inr :: 'b  $\Rightarrow$  'a + 'b)*  
**proof** –  
**have** *image*: (*Inr :: 'b  $\Rightarrow$  'a + 'b*) ‘ *topspace Y  $\subseteq$  topspace (binary-sum-topology X Y)*  
**by** *simp*  
**have** *preimage*:  
 $\forall U. \text{openin } (\text{binary-sum-topology } X \ Y) \ U \longrightarrow \text{openin } Y \ \{y \in \text{topspace } Y. \text{Inr } y \in U\}$   
**proof** (*intro allI impI*)  
**fix** *U*  
**assume** *U*: *openin (binary-sum-topology X Y) U*  
**have** *openU*: *openin Y {y. Inr y  $\in$  U}*  
**using** *U* **by** (*simp add: openin-binary-sum-topology*)  
**have** *subsetY*: *{y. Inr y  $\in$  U}  $\subseteq$  topspace Y*  
**using** *openU* **by** (*rule openin-subset*)  
**have** *eq*: *{y  $\in$  topspace Y. Inr y  $\in$  U} = {y. Inr y  $\in$  U}*  
**using** *subsetY* **by** *auto*  
**have** *openin Y {y  $\in$  topspace Y. Inr y  $\in$  U}*  
**unfolding** *eq* **using** *openU* **by** *simp*  
**then show** *openin Y {y  $\in$  topspace Y. Inr y  $\in$  U} .*  
**qed**

```

from image preimage show ?thesis
  by (simp add: continuous-map)
qed

lemma open-map-binary-sum-inl:
  open-map X (binary-sum-topology X Y) (Inl :: 'a ⇒ 'a + 'b)
  unfolding open-map-def openin-binary-sum-topology
  using openin-subset openin-subopen by (force simp: image-iff)

lemma open-map-binary-sum-inr:
  open-map Y (binary-sum-topology X Y) (Inr :: 'b ⇒ 'a + 'b)
  unfolding open-map-def openin-binary-sum-topology
  using openin-subset openin-subopen by (force simp: image-iff)

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 –
  from f have f-image: f ‘ topspace X ⊆ topspace Z
  by (simp add: continuous-map)
  from g have g-image: g ‘ topspace Y ⊆ topspace Z
  by (simp add: continuous-map)
  have case-image:
    case-sum f g ‘ topspace (binary-sum-topology X Y) = f ‘ topspace X ∪ g ‘
    topspace Y
  proof
    show case-sum f g ‘ topspace (binary-sum-topology X Y) ⊆ f ‘ topspace X ∪ g
    ‘ topspace Y
    by (auto simp: topspace-binary-sum-topology)
    show f ‘ topspace X ∪ g ‘ topspace Y ⊆ case-sum f g ‘ topspace
    (binary-sum-topology X Y)
  proof
    fix z
    assume z ∈ f ‘ topspace X ∪ g ‘ topspace Y
    then consider
      (left) x where x ∈ topspace X z = f x
    | (right) y where y ∈ topspace Y z = g y
    by auto
    then show z ∈ case-sum f g ‘ topspace (binary-sum-topology X Y)
  proof cases
    case (left x)
      have Inl x ∈ topspace (binary-sum-topology X Y)
      using left by (auto simp: topspace-binary-sum-topology)
    then have case-sum f g (Inl x) ∈ case-sum f g ‘ topspace (binary-sum-topology
    X Y)
      by (rule imageI)
    then show ?thesis
      using left by simp

```

```

next
  case (right y)
  have Inr y ∈ topspace (binary-sum-topology X Y)
    using right by (auto simp: topspace-binary-sum-topology)
  then have case-sum f g (Inr y) ∈ case-sum f g ‘ topspace (binary-sum-topology
X Y)
    by (rule imageI)
  then show ?thesis
    using right by simp
qed
qed
qed
have image: case-sum f g ‘ topspace (binary-sum-topology X Y) ⊆ topspace Z
proof -
  have case-subset: f ‘ topspace X ∪ g ‘ topspace Y ⊆ topspace Z
    using f-image g-image by blast
  from case-image case-subset show ?thesis
    by blast
qed
have preimage:
  ∀ U. openin Z U ⟶
    openin (binary-sum-topology X Y) {z ∈ topspace (binary-sum-topology X Y).
case-sum f g z ∈ U}
proof (intro allI impI)
  fix U
  assume U: openin Z U
  from f U have openX: openin X {x ∈ topspace X. f x ∈ U}
    by (simp add: continuous-map)
  from g U have openY: openin Y {y ∈ topspace Y. g y ∈ U}
    by (simp add: continuous-map)
  let ?W = {z ∈ topspace (binary-sum-topology X Y). case-sum f g z ∈ U}
  have subsetW: ?W ⊆ topspace (binary-sum-topology X Y)
    by auto
  have eqX: {x. Inl x ∈ ?W} = {x ∈ topspace X. f x ∈ U}
    by auto
  have eqY: {y. Inr y ∈ ?W} = {y ∈ topspace Y. g y ∈ U}
    by auto
  show openin (binary-sum-topology X Y) ?W
    unfolding openin-binary-sum-topology eqX eqY
    using subsetW openX openY by simp
qed
from image preimage show ?thesis
  by (simp add: continuous-map)
qed

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-topospace* ::

*'a topology => 'b topology => ('c => 'a) => ('c => 'b) => ('a + 'b) set set*

**where**

*pushout-topospace 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-topospace 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-topospace-alt*:

*pushout-topospace X Y f g = pushout-inl f g ' topspace X ∪ pushout-inr f g ' topspace Y*

**unfolding** *pushout-topospace-def pushout-inl-def pushout-inr-def topspace-binary-sum-topology*

**by** *auto*

**lemma** *pushout-topospace-subset-pushout-space*:

*pushout-topospace X Y f g ⊆ pushout-space f g*

**unfolding** *pushout-topospace-def pushout-space-def pushout-class-def quotient-space-def*

**by** *blast*

**lemma** *is-pushout-topology*:

*istopology (λU.*

*U ⊆ pushout-topospace X Y f g ∧*

*openin (binary-sum-topology X Y)*

*{z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ U})*

**proof** –

**have** *Int-eq*:

*{z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ S ∩ T} =*

*{z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ S} ∩*

*{z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ T} **for** S T*

**by** *auto*

**have** *Union-eq*:

*{z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ ∪K} =*

*(∪ U ∈ K. {z ∈ topspace (binary-sum-topology X Y). pushout-class f g z ∈ U})*

**for**  $\mathcal{K}$   
**by** *auto*  
**show** *?thesis*  
**unfolding** *istopology-def pushout-topospace-def Int-eq Union-eq*  
**by** *blast*  
**qed**

**lemma** *openin-pushout-topology*:  
 $openin (pushout-topology X Y f g) U \longleftrightarrow$   
 $U \subseteq pushout-topospace X Y f g \wedge$   
 $openin (binary-sum-topology X Y)$   
 $\{z \in topspace (binary-sum-topology X Y). pushout-class f g z \in U\}$

**proof** –  
**have**  $openin (pushout-topology X Y f g) U =$   
 $(U \subseteq pushout-topospace X Y f g \wedge$   
 $openin (binary-sum-topology X Y)$   
 $\{z \in topspace (binary-sum-topology X Y). pushout-class f g z \in U\})$   
**unfolding** *pushout-topology-def*  
**by** (*subst topology-inverse'[OF is-pushout-topology]*) *simp*  
**then show** *?thesis*  
**by** *simp*  
**qed**

**lemma** *topspace-pushout-topology [simp]*:  
 $topspace (pushout-topology X Y f g) = pushout-topospace X Y f g$

**proof**  
**have** *top-open*:  $openin (pushout-topology X Y f g) (topspace (pushout-topology X Y f g))$   
**by** *simp*  
**show**  $topspace (pushout-topology X Y f g) \subseteq pushout-topospace X Y f g$   
**using** *top-open* **unfolding** *openin-pushout-topology* **by** *blast*  
**have** *preimage-space*:  
 $\{z \in topspace (binary-sum-topology X Y). pushout-class f g z \in pushout-topospace X Y f g\} =$   
 $topspace (binary-sum-topology X Y)$   
**unfolding** *pushout-topospace-def* **by** *auto*  
**have** *open-top*:  $openin (binary-sum-topology X Y) (topspace (binary-sum-topology X Y))$   
**by** (*rule openin-topospace*)  
**have**  $openin (pushout-topology X Y f g) (pushout-topospace X Y f g)$   
**unfolding** *openin-pushout-topology preimage-space*  
**using** *open-top* **by** *simp*  
**then show**  $pushout-topospace X Y f g \subseteq topspace (pushout-topology X Y f g)$   
**by** (*rule openin-subset*)  
**qed**

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

**proof** (*unfold quotient-map-def, intro conjI allI impI*)  
**show**  $\text{pushout-class } f \ g \ \text{' } \text{topspace } (\text{binary-sum-topology } X \ Y) = \text{topspace } (\text{pushout-topology } X \ Y \ f \ g)$   
**by** (*auto simp: pushout-topospace-alt pushout-inl-def pushout-inr-def image-Un*)  
**next**  
**fix**  $U$   
**assume**  $U: U \subseteq \text{topspace } (\text{pushout-topology } X \ Y \ f \ g)$   
**then have**  $U': U \subseteq \text{pushout-topospace } X \ Y \ f \ g$   
**by** *simp*  
**show**  $\text{openin } (\text{binary-sum-topology } X \ Y) \ \{x \in \text{topspace } (\text{binary-sum-topology } X \ Y). \text{pushout-class } f \ g \ x \in U\} = \text{openin } (\text{pushout-topology } X \ Y \ f \ g) \ U$   
**using**  $U'$  **by** (*simp add: openin-pushout-topology*)  
**qed**

**lemma** *continuous-map-pushout-class*:  
 $\text{continuous-map } (\text{binary-sum-topology } X \ Y) \ (\text{pushout-topology } X \ Y \ f \ g) \ (\text{pushout-class } f \ g)$   
**using** *quotient-map-pushout-class* **by** (*rule quotient-imp-continuous-map*)

**lemma** *pushout-inl-in-topospace*:  
**assumes**  $a \in \text{topspace } X$   
**shows**  $\text{pushout-inl } f \ g \ a \in \text{pushout-topospace } X \ Y \ f \ g$   
**using** *assms*  
**unfolding** *pushout-topospace-def pushout-inl-def topspace-binary-sum-topology*  
**by** *auto*

**lemma** *pushout-inr-in-topospace*:  
**assumes**  $b \in \text{topspace } Y$   
**shows**  $\text{pushout-inr } f \ g \ b \in \text{pushout-topospace } X \ Y \ f \ g$   
**using** *assms*  
**unfolding** *pushout-topospace-def pushout-inr-def topspace-binary-sum-topology*  
**by** *auto*

**lemma** *continuous-map-pushout-inl* [*continuous-intros*]:  
 $\text{continuous-map } X \ (\text{pushout-topology } X \ Y \ f \ g) \ (\text{pushout-inl } f \ g)$   
**proof** –  
**have**  $\text{continuous-map } X \ (\text{pushout-topology } X \ Y \ f \ g) \ (\text{pushout-class } f \ g \circ (\text{Inl} :: 'a \Rightarrow 'a + 'b))$   
**using** *continuous-map-binary-sum-inl continuous-map-pushout-class*  
**by** (*rule continuous-map-compose*)  
**then show** *?thesis*  
**unfolding** *pushout-inl-def o-def*  
**by** (*rule continuous-map-eq*) *auto*  
**qed**

**lemma** *continuous-map-pushout-inr* [*continuous-intros*]:  
 $\text{continuous-map } Y \ (\text{pushout-topology } X \ Y \ f \ g) \ (\text{pushout-inr } f \ g)$   
**proof** –

```

have inr-cont: continuous-map  $Y$  (binary-sum-topology  $X$   $Y$ ) ( $\lambda b. \text{Inr } b$ )
  using continuous-map-binary-sum-inr
  by (rule continuous-map-eq) auto
have continuous-map  $Y$  (pushout-topology  $X$   $Y$   $f$   $g$ )
  (pushout-class  $f$   $g$   $\circ$  ( $\lambda b. \text{Inr } b$ ))
  using inr-cont continuous-map-pushout-class
  by (rule continuous-map-compose)
then show ?thesis
  unfolding pushout-inr-def o-def
  by (rule continuous-map-eq) auto
qed

```

```

lemma pushout-rec-pushout-class:
  assumes compat: pushout-case-compatible  $f$   $g$  left right
    and x-in:  $x \in \text{topspace}$  (binary-sum-topology  $X$   $Y$ )
  shows pushout-rec  $f$   $g$  left right (pushout-class  $f$   $g$   $x$ ) = case-sum left right  $x$ 
proof –
  from x-in have ( $\exists a \in \text{topspace } X. x = \text{Inl } a$ )  $\vee$  ( $\exists b \in \text{topspace } Y. x = \text{Inr } b$ )
    by (auto simp: topspace-binary-sum-topology)
  then show ?thesis
proof
  assume  $\exists a \in \text{topspace } X. x = \text{Inl } a$ 
  then obtain  $a$  where  $a \in \text{topspace } X$   $x = \text{Inl } a$ 
    by blast
  then show ?thesis
    using pushout-rec-inl[OF compat, of a]
    unfolding pushout-inl-def by simp
next
  assume  $\exists b \in \text{topspace } Y. x = \text{Inr } b$ 
  then obtain  $b$  where  $b \in \text{topspace } Y$   $x = \text{Inr } b$ 
    by blast
  then show ?thesis
    using pushout-rec-inr[OF compat, of b]
    unfolding pushout-inr-def by simp
qed
qed

```

```

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 –
  have sum-cont: continuous-map (binary-sum-topology  $X$   $Y$ )  $Z$  (case-sum left
right)
    using left-cont right-cont by (rule continuous-map-from-binary-sum-topology)
  have comp-eq:
    (pushout-rec  $f$   $g$  left right)  $\circ$  pushout-class  $f$   $g$ )  $x$  = case-sum left right  $x$ 
    if  $x \in \text{topspace}$  (binary-sum-topology  $X$   $Y$ ) for  $x$ 

```

```

    using pushout-rec-pushout-class[OF compat that] by (simp add: o-def)
  have comp-eq':
    case-sum left right x = ((pushout-rec f g left right) ◦ pushout-class f g) x
  if x ∈ topspace (binary-sum-topology X Y) for x
  using comp-eq[OF that] by simp
  have comp-cont:
    continuous-map (binary-sum-topology X Y) Z ((pushout-rec f g left right) ◦
pushout-class f g)
  using sum-cont comp-eq' by (rule continuous-map-eq)
  show ?thesis
    using quotient-map-pushout-class comp-cont by (rule continu-
ous-compose-quotient-map)
qed

```

**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  $\bigwedge a. \text{pushout-rec } f \text{ } g \text{ left right (pushout-inl } f \text{ } g \text{ } a) = \text{left } a$ 
    and  $\bigwedge b. \text{pushout-rec } f \text{ } g \text{ left right (pushout-inr } f \text{ } g \text{ } b) = \text{right } b$ 
  using assms
  by (auto intro: continuous-map-pushout-rec simp: pushout-rec-inl pushout-rec-inr)

```

**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

**unfolding** *loopin-image-def* **by** (rule *ext*) *simp*

**lemma** *fundamental-groupin-map-eq*:

**assumes** *A-in*:  $A \in \text{fundamental-groupin-space } X \text{ } x$

**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:  $\bigwedge u. f u = g u$ 
shows fundamental-groupin-map X x Y y f A = fundamental-groupin-map X x
Y y g A
proof –
  have repA: some-loopin X x A  $\in$  loopin-space X x A = loopin-class X x
(some-loopin X x A)
  using some-loopin-spec[OF A-in] by auto
  have left-eq:
    fundamental-groupin-map X x Y y f A =
      loopin-class Y y (loopin-image f (some-loopin X x A))
  by (rule fundamental-groupin-map-rep[OF A-in repA(1) repA(2) f-cont fx])
  have right-eq:
    fundamental-groupin-map X x Y y g A =
      loopin-class Y y (loopin-image g (some-loopin X x A))
  by (rule fundamental-groupin-map-rep[OF A-in repA(1) repA(2) g-cont gx])
  have loopin-image f (some-loopin X x A) = loopin-image g (some-loopin X x A)
  using fg by (simp add: loopin-image-def fun-eq-iff)
  then show ?thesis
  using left-eq right-eq by simp
qed

```

```

lemma fundamental-groupin-map-compose:
  assumes A-in: A  $\in$  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  $\circ$  f) A
proof –
  have repA: some-loopin X x A  $\in$  loopin-space X x A = loopin-class X x
(some-loopin X x A)
  using some-loopin-spec[OF A-in] by auto
  have mapA-in: fundamental-groupin-map X x Y y f A  $\in$  fundamen-
tal-groupin-space Y y
  by (rule fundamental-groupin-map-in-space[OF A-in f-cont fx])
  have mapA-eq:
    fundamental-groupin-map X x Y y f A =
      loopin-class Y y (loopin-image f (some-loopin X x A))
  by (rule fundamental-groupin-map-rep[OF A-in repA(1) repA(2) f-cont fx])
  have map-loop-in: loopin-image f (some-loopin X x A)  $\in$  loopin-space Y y
  by (rule loopin-image-in-space[OF repA(1) f-cont fx])
  have left-eq:
    fundamental-groupin-map Y y Z z g (fundamental-groupin-map X x Y y f A) =
      loopin-class Z z (loopin-image g (loopin-image f (some-loopin X x A)))
  by (rule fundamental-groupin-map-rep[OF mapA-in map-loop-in mapA-eq g-cont

```

```

gy])
have comp-cont: continuous-map X Z (g ∘ f)
  using f-cont g-cont by (rule continuous-map-compose)
have right-eq:
  fundamental-groupin-map X x Z z (g ∘ f) A =
    loopin-class Z z (loopin-image (g ∘ f) (some-loopin X x A))
  by (rule fundamental-groupin-map-rep[OF A-in repA(1) repA(2) comp-cont])
(simp add: fx gy)
show ?thesis
  using left-eq right-eq by simp
qed

```

```

locale topological-svk-setup =
  fixes X :: 'a topology
    and Y :: 'b topology
    and Z :: 'c topology
    and f :: 'c ⇒ 'a
    and g :: 'c ⇒ 'b
    and z0 :: 'c
  assumes z0-in: z0 ∈ topspace Z
    and f-cont: continuous-map Z X f
    and g-cont: continuous-map Z Y g
begin

```

```

abbreviation x0 where x0 ≡ f z0
abbreviation y0 where y0 ≡ g z0
abbreviation P where P ≡ pushout-topology X Y f g
abbreviation p0 where p0 ≡ pushout-inl f g x0

```

```

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

```

```

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

```

```

lemma x0-in [simp]: x0 ∈ topspace X
  using z0-in f-cont unfolding continuous-map-def by blast

```

**lemma** *y0-in* [*simp*]:  $y0 \in \text{topspace } Y$   
**using** *z0-in g-cont unfolding continuous-map-def* **by** *blast*

**lemma** *pushout-basepoint-eq* [*simp*]:  
 $\text{pushout-inr } f \ g \ y0 = p0$   
**by** (*simp add: pushout-glua*)

**lemma** *pushout-basepoint-in* [*simp*]:  
 $p0 \in \text{topspace } P$   
**by** (*simp add: pushout-inl-in-topspace[OF x0-in]*)

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

**proof** –

**have** *left-comp*:

$j1 \ (i1 \ h) = \text{fundamental-groupin-map } Z \ z0 \ P \ p0 \ ((\text{pushout-inl } f \ g) \circ f) \ h$

**by** (*simp add: fundamental-groupin-map-compose[OF h-in f-cont continuous-map-pushout-inl]*)

**have** *right-comp*:

$j2 \ (i2 \ h) = \text{fundamental-groupin-map } Z \ z0 \ P \ p0 \ ((\text{pushout-inr } f \ g) \circ g) \ h$

**by** (*simp add: fundamental-groupin-map-compose[OF h-in g-cont continuous-map-pushout-inr]*)

**have** *left-cont*:  $\text{continuous-map } Z \ P \ ((\text{pushout-inl } f \ g) \circ f)$

**by** (*rule continuous-map-compose[OF f-cont continuous-map-pushout-inl]*)

**have** *right-cont*:  $\text{continuous-map } Z \ P \ ((\text{pushout-inr } f \ g) \circ g)$

**by** (*rule continuous-map-compose[OF g-cont continuous-map-pushout-inr]*)

**have** *left-z0*:  $((\text{pushout-inl } f \ g) \circ f) \ z0 = p0$

**by** *simp*

**have** *right-z0*:  $((\text{pushout-inr } f \ g) \circ g) \ z0 = p0$

**by** *simp*

**have** *comp-fun-eq*:  $((\text{pushout-inl } f \ g) \circ f) \ u = ((\text{pushout-inr } f \ g) \circ g) \ u$  **for**  $u$

**by** (*simp add: pushout-glua*)

**have** *comp-eq*:

$\text{fundamental-groupin-map } Z \ z0 \ P \ p0 \ ((\text{pushout-inl } f \ g) \circ f) \ h =$

$\text{fundamental-groupin-map } Z \ z0 \ P \ p0 \ ((\text{pushout-inr } f \ g) \circ g) \ h$

**by** (*rule fundamental-groupin-map-eq[OF h-in left-cont right-cont left-z0 right-z0 comp-fun-eq]*)

**show** *?thesis*

**using** *left-comp right-comp comp-eq* **by** *simp*

**qed**

**lemma** *i1-in-G1*:

**assumes**  $h \in H$

**shows**  $i1 \ h \in G1$

**by** (*rule fundamental-groupin-map-in-space[OF assms f-cont]*) *simp*

**lemma** *i2-in-G2*:

**assumes**  $h \in H$

**shows**  $i2\ h \in G2$   
**by** (rule *fundamental-groupin-map-in-space*[*OF assms g-cont*]) *simp*

**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** (rule *carrier-full-amalg-word-eval.intro*)  
**show** *carrier-group*  
 $(fundamental-groupin-space\ X\ x0)$   
 $(fundamental-groupin-mult\ X\ x0)$   
 $(fundamental-groupin-one\ X\ x0)$   
 $(fundamental-groupin-inv\ X\ x0)$   
**by** (rule *fundamental-groupin-carrier-group*[*OF x0-in*])  
**show** *carrier-group*  
 $(fundamental-groupin-space\ Y\ y0)$   
 $(fundamental-groupin-mult\ Y\ y0)$   
 $(fundamental-groupin-one\ Y\ y0)$   
 $(fundamental-groupin-inv\ Y\ y0)$   
**by** (rule *fundamental-groupin-carrier-group*[*OF y0-in*])  
**show** *carrier-group*  
 $(fundamental-groupin-space\ P\ p0)\ multP\ oneP\ (fundamental-groupin-inv\ P\ p0)$   
**by** (rule *fundamental-groupin-carrier-group*[*OF pushout-basepoint-in*])  
**show** *carrier-group-hom*  
 $(fundamental-groupin-space\ X\ x0)$   
 $(fundamental-groupin-mult\ X\ x0)$   
 $(fundamental-groupin-one\ X\ x0)$   
 $(fundamental-groupin-inv\ X\ x0)$   
 $(fundamental-groupin-space\ P\ p0)\ multP\ oneP\ (fundamental-groupin-inv\ P\ p0)$   
 $(fundamental-groupin-map\ X\ x0\ P\ p0\ (pushout-inl\ f\ g))$   
**by** (rule *fundamental-groupin-map-carrier-group-hom*[*OF x0-in continuous-map-pushout-inl*]) *simp*  
**show** *carrier-group-hom*  
 $(fundamental-groupin-space\ Y\ y0)$   
 $(fundamental-groupin-mult\ Y\ y0)$   
 $(fundamental-groupin-one\ Y\ y0)$   
 $(fundamental-groupin-inv\ Y\ y0)$   
 $(fundamental-groupin-space\ P\ p0)\ multP\ oneP\ (fundamental-groupin-inv\ P\ p0)$   
 $(fundamental-groupin-map\ Y\ y0\ P\ p0\ (pushout-inr\ f\ g))$   
**by** (rule *fundamental-groupin-map-carrier-group-hom*[*OF y0-in continuous-map-pushout-inr*]) *simp*  
**show** *carrier-full-amalg-word-eval-axioms*  $G1\ G2\ H\ i1\ i2\ j1\ j2$   
**proof** (rule *carrier-full-amalg-word-eval-axioms.intro*)

```

show  $h \in H \implies i1\ h \in G1$  for  $h$ 
  by (rule i1-in-G1)
show  $h \in H \implies i2\ h \in G2$  for  $h$ 
  by (rule i2-in-G2)
show  $h \in H \implies j1\ (i1\ h) = j2\ (i2\ h)$  for  $h$ 
  by (rule pushout-fundamental-group-maps-agree)
qed
qed

```

**interpretation** *decode*:

```

carrier-full-amalg-word-eval
   $G1\ mult1\ one1\ fundamental\ groupin\ inv\ X\ x0$ 
   $G2\ mult2\ one2\ fundamental\ groupin\ inv\ Y\ y0$ 
   $H\ i1\ i2$ 
   $fundamental\ groupin\ space\ P\ p0\ multP\ oneP\ fundamental\ groupin\ inv\ P\ p0$ 
   $j1\ j2$ 
by (rule decode-locale)

```

**abbreviation** *svk-word-eval* **where** *svk-word-eval*  $\equiv$  *decode.carrier-full-amalg-eval*  
**abbreviation** *svk-decode* **where** *svk-decode*  $\equiv$  *decode.carrier-full-amalg-decode*

**lemma** *svk-decode-in-space*:

```

svk-decode  $w \in fundamental\ groupin\ space\ P\ p0$ 
by (rule decode.carrier-full-amalg-decode-in-carrier)

```

**lemma** *svk-decode-respects*:

```

assumes carrier-full-amalg-equiv  $G1\ G2\ H\ i1\ i2\ mult1\ one1\ mult2\ one2\ u\ v$ 
shows svk-decode  $u = svk-decode\ v$ 
using assms by (rule decode.carrier-full-amalg-decode-respects)

```

**lemma** *svk-decode-eq-eval*:

```

assumes fpw-in-space  $G1\ G2\ w$ 
shows svk-decode  $w = svk-word-eval\ w$ 
using assms by (rule decode.carrier-full-amalg-decode-eq-eval)

```

**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  $f :: 'c \Rightarrow 'a$ 
  and  $g :: 'c \Rightarrow 'b$ 
  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*:  $p \in \text{loopin-space } P \ p0$   
**shows**  $\exists n::\text{nat. } 0 < n \wedge$   
 $(\forall i < n.$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) \ p \ \{0..1\}$   
 $\subseteq \text{pushout-inl } f \ g \ \text{'topspace } X \ \vee$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) \ p \ \{0..1\}$   
 $\subseteq \text{pushout-inr } f \ g \ \text{'topspace } Y)$

**proof** –  
**let**  $?L = \text{pushout-inl } f \ g \ \text{'topspace } X$   
**let**  $?R = \text{pushout-inr } f \ g \ \text{'topspace } Y$   
**from** *p-loop* **obtain** *p-path* **where** *p-path*:  $\text{pathin } P \ p$   
**by** (*rule loopin-spaceE*)  
**have** *p-img*:  $p \ \{0..1\} \subseteq \text{topspace } P$   
**by** (*rule pathin-image-subset-topspace*[*OF p-path*])  
**have** *cover*:  $p \ \{0..1\} \subseteq ?L \cup ?R$   
**using** *p-img* **by** (*auto simp: pushout-topspace-alt*)  
**have** *cover'*:  $p \ \{0..1\} \subseteq \bigcup \{?L, ?R\}$   
**using** *cover* **by** *auto*  
**have** *open-cover*:  $\text{openin } P \ U$  **if**  $U \in \{?L, ?R\}$  **for**  $U$   
**using** *that left-open right-open* **by** *auto*  
**have** *subdiv*:  
 $\exists n::\text{nat. } 0 < n \wedge$   
 $(\forall i < n. \exists U \in \{?L, ?R\}.$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) \ p \ \{0..1\} \subseteq U)$   
**by** (*rule pathin-subdivision-open-cover*[*OF p-path cover' open-cover*])  
**then obtain**  $n :: \text{nat}$  **where** *n-pos*:  $0 < n$   
**and** *n-cover*:  
 $\forall i < n. \exists U \in \{?L, ?R\}.$   
 $\text{subpathin } (\text{real } i / \text{real } n) (\text{real } (\text{Suc } i) / \text{real } n) \ p \ \{0..1\} \subseteq U$   
**by** *blast*  
**show** *thesis*  
**using** *n-pos n-cover* **by** *auto*

**qed**

**end**

**locale** *topological-svk* =  
*topological-svk-setup*  $X \ Y \ Z \ f \ g \ z0$   
**for**  $X :: 'a \ \text{topology}$   
**and**  $Y :: 'b \ \text{topology}$   
**and**  $Z :: 'c \ \text{topology}$   
**and**  $f :: 'c \Rightarrow 'a$   
**and**  $g :: 'c \Rightarrow 'b$   
**and**  $z0 :: 'c +$   
**fixes** *encode*  
**assumes** *encode-in-space*:  $\text{fpw-in-space } G1 \ G2 \ (\text{encode } A)$   
**and** *decode-encode*:  $\text{svk-decode } (\text{encode } A) = A$   
**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** (*rule carrier-svk-encode-decode-interface.intro*)

**show** *fpw-in-space G1 G2 (encode x) for x*

**by** (*fact encode-in-space*)

**show** *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 u v  $\implies$  svk-decode u = svk-decode v for u v*

**by** (*rule svk-decode-respects*)

**show** *svk-decode (encode x) = x for x*

**by** (*fact decode-encode*)

**show** *carrier-full-amalg-equiv G1 G2 H i1 i2 mult1 one1 mult2 one2 (encode (svk-decode w)) w for w*

**by** (*fact encode-decode*)

**qed**

**interpretation** *svk: carrier-svk-encode-decode-interface*

*G1 G2 H i1 i2 mult1 one1 mult2 one2 encode svk-decode*

**by** (*rule svk-locale*)

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)*

**by** (*rule svk.carrier-seifert-van-kampen-bij-betw*)

**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/5100091>.