Path Equivalence and Automation for Integration Contours

Manuel Eberl

October 16, 2026

Abstract

In complex analysis, one often has to manipulate *paths*, i.e. curves in the complex plane. This entry defines three useful relations on paths:

- an equivalence relation $\equiv_{\text{p}}$ that describes that two paths are the same up to reparametrisation
- a preorder $\leq_{\text{p}}$ that expresses the notion that one path is a subpath of another
- an equivalence relation $\equiv_{\circ}$ that describes equivalence of closed paths up to reparametrisation and "shifting" (e.g. if we have a rectangular path, it does not matter which corner we start in)

It also provides the `path` tactic, which proves or simplifies some common proof obligations for composite paths. Namely:

- proving $\equiv_{\text{p}}$, $\leq_{\text{p}}$, $\equiv_{\circ}$
- proving well-definedness of paths (`path`, `valid_path`)
- determining the *image* of a path (`path_image`)
- showing that a path is not self-intersecting (`arc`, `simple_path`)
- decomposing integrals on a composite path into the integrals on the constituent paths

Contents

1 Auxiliary material **3**

1.1 Miscellaneous . . . 3
1.2 Some facts about strict monotonicity . . . 3
1.3 General lemmas about topology . . . 7
1.4 General lemmas about real functions . . . 9
1.5 Rounding and fractional part . . . 12
1.6 General lemmas about paths . . . 16
1.7 Some facts about betweenness . . . 19
1.8 Simple loops and orientation . . . 21
1.9 More about circular arcs . . . 24
1.10 Reparametrisation of loops by shifting . . . 26

2 Some useful relations on paths **32**

2.1 Equivalence of paths up to reparametrisation . . . 32
2.2 Splitting lines and circular arcs . . . 43
2.3 The subpath relation . . . 48
2.4 Equivalence of closed paths . . . 57
2.5 Notation . . . 69
2.6 Examples . . . 70

3 Automation for paths **70**

3.1 Joining a list of paths together . . . 70
3.2 Representing a sequence of path joins as a tree . . . 79
3.3 Equivalence of two join trees . . . 83
3.4 Implementation . . . 84
3.5 Examples . . . 95

1 Auxiliary material

```
theory Path_Automation_Library
  imports "HOL-Complex_Analysis.Complex_Analysis"
begin
```

1.1 Miscellaneous

```
lemma cis_multiple_2pi':
  "n ∈ ℤ ⟹ cis (2 * n * pi) = 1"
  "n ∈ ℤ ⟹ cis (2 * (n * pi)) = 1"
  "n ∈ ℤ ⟹ cis (pi * (2 * n)) = 1"
  "n ∈ ℤ ⟹ cis (pi * (n * 2)) = 1"
  using cis_multiple_2pi[of n] by (simp_all only: mult_ac)

lemma dist_linepath1: "dist (linepath a b x) a = |x| * dist a b"
proof -
  have "dist (linepath a b x) a = norm (x *_R (b - a))"
    unfolding scaleR_diff_right
    by (simp add: linepath_def dist_norm algebra_simps)
  also have "... = |x| * dist a b"
    by (subst norm_scaleR) (auto simp: dist_norm norm_minus_commute)
  finally show ?thesis .
qed

lemma dist_linepath2: "dist (linepath a b x) b = |1 - x| * dist a b"
proof -
  have "dist (linepath a b x) b = norm ((x - 1) *_R (b - a))"
    unfolding scaleR_diff_right
    by (simp add: linepath_def dist_norm algebra_simps)
  also have "... = |x - 1| * dist a b"
    by (subst norm_scaleR) (auto simp: dist_norm norm_minus_commute)
  finally show ?thesis
    by (simp add: abs_minus_commute)
qed
```

1.2 Some facts about strict monotonicity

```
lemma strict_mono_on_atLeastAtMost_combine:
  fixes f :: "'a :: linorder ⇒ 'b :: linorder"
  assumes "strict_mono_on {a..b} f" "strict_mono_on {b..c} f"
  shows   "strict_mono_on {a..c} f"
proof (rule strict_mono_onI)
  fix r s
  assume rs: "r ∈ {a..c}" "s ∈ {a..c}" "r < s"
  consider "r ∈ {a..b}" "s ∈ {a..b}" | "r ∈ {a..b}" "s ∈ {b<..c}" |
"r ∈ {b..c}" "s ∈ {b..c}"
    using rs by force
  thus "f r < f s"
  proof cases
```

```
    assume rs: "r ∈ {a..b}" "s ∈ {b<..c}"
    have "f r ≤ f b"
      using rs by (intro strict_mono_on_leD[OF assms(1)]) auto
    also have "f b < f s"
      using rs by (intro strict_mono_onD[OF assms(2)]) auto
    finally show "f r < f s" .
  qed (use assms ‹r < s› in ‹auto simp: strict_mono_on_def›)
qed


lemma mono_on_compose:
  assumes "mono_on A f" "mono_on B g" "f ‘ A ⊆ B"
  shows   "mono_on A (g ∘ f)"
  unfolding o_def
  by (intro mono_onI mono_onD[OF assms(1)] mono_onD[OF assms(2)]) (use
assms(3) in auto)


lemma strict_mono_on_compose:
  assumes "strict_mono_on B g" "strict_mono_on A f" "f ‘ A ⊆ B"
  shows   "strict_mono_on A (g ∘ f)"
  unfolding strict_mono_on_def using assms(3)
  by (auto simp: strict_mono_on_def intro!: assms(1,2)[THEN strict_mono_onD])


lemma strict_mono_on_less:
  assumes "strict_mono_on S (f::'a :: linorder ⇒ 'b::preorder)"
  assumes "x ∈ S" "y ∈ S"
  shows "f x < f y ⟷ x < y"
  using strict_mono_onD[OF assms(1,2,3)] strict_mono_onD[OF assms(1,3,2)]
        order_less_imp_not_less[of "f x" "f y"]
  by (cases x y rule: linorder_cases) auto


lemma strict_mono_on_imp_strict_mono_on_inv:
  fixes f :: "'a :: linorder ⇒ 'b :: preorder"
  assumes "strict_mono_on {a..b} f"
  assumes "⋀x. x ∈ {a..b} ⟹ g (f x) = x"
  shows   "strict_mono_on (f ‘ {a..b}) g"
proof (rule strict_mono_onI, safe)
  fix r s assume rs: "r ∈ {a..b}" "s ∈ {a..b}" "f r < f s"
  thus "g (f r) < g (f s)"
    using strict_mono_on_less[OF assms(1)] rs by (auto simp: assms(2))
qed


lemma strict_mono_on_imp_strict_mono_on_inv_into:
  fixes f :: "'a :: linorder ⇒ 'b :: preorder"
  assumes "strict_mono_on {a..b} f"
```

```
  shows   "strict_mono_on (f ` {a..b}) (inv_into {a..b} f)"
  using strict_mono_on_imp_strict_mono_on_inv[OF
        assms inv_into_f_f[OF strict_mono_on_imp_inj_on[OF assms]]]
  by blast
```

Nice lemma taken from Austin, A. K. (1985). 69.8 Two Curiosities. The Mathematical Gazette, 69(447), 4244. https://doi.org/10.2307/3616452.

A strictly monotonic function `f` on some closed real interval has a continuous (and strictly monotonic) inverse function `g` – even if `f` itself is not continuous.

```
lemma strict_mono_on_imp_continuous_on_inv:
  fixes f :: "real ⇒ real"
  assumes "strict_mono_on {a..b} f"
  assumes "⋀x. x ∈ {a..b} ⟹ g (f x) = x"
  shows   "continuous_on (f ` {a..b}) g"
proof (cases "a < b")
  case False
  thus ?thesis
    by (cases "a = b") auto
next
  case ab: True
  show ?thesis
  proof (rule continuous_onI, safe)
    fix x ε :: real
    assume ε: "ε > 0" and x: "x ∈ {a..b}"

    consider "x = a" | "x = b" | "x ∈ {a<..<b}"
      using x by force
    thus "∃d>0. ∀x'∈f ` {a..b}. dist x' (f x) < d ⟶ dist (g x') (g
(f x)) ≤ ε"
    proof cases
      assume [simp]: "x = a"
      define ε' where "ε' = min ε ((b - a) / 2)"
      have ε': "ε' > 0" "ε' ≤ ε" "ε' < b - a"
        using ε ‹a < b› by (auto simp: ε'_def min_less_iff_disj)
      define δ where "δ = f (a + ε') - f a"
      show ?thesis
      proof (rule exI[of _ δ], safe)
        show "δ > 0"
          using ‹a < b› ε' by (auto simp: δ_def intro!: strict_mono_onD[OF
assms(1)])
      next
        fix t assume t: "t ∈ {a..b}" "dist (f t) (f x) < δ"
        have "f t ≥ f a"
          using ‹a < b› t by (intro strict_mono_on_leD[OF assms(1)])
auto
        with t have "f t - f a < δ"
          by (simp add: dist_norm)
        hence "f t < f (a + ε')"
          unfolding δ_def by linarith
```

```
        hence "t < a + ε'"
          using t ε' by (subst (asm) strict_mono_on_less[OF assms(1)])
auto
        thus "dist (g (f t)) (g (f x)) ≤ ε"
          using ‹a < b› t ‹f t ≥ f a› ε' by (simp add: assms dist_norm)
      qed

    next

      assume [simp]: "x = b"
      define ε' where "ε' = min ε ((b - a) / 2)"
      have ε': "ε' > 0" "ε' ≤ ε" "ε' < b - a"
        using ε ‹a < b› by (auto simp: ε'_def min_less_iff_disj)
      define δ where "δ = f b - f (b - ε')"
      show ?thesis
      proof (rule exI[of _ δ], safe)
        show "δ > 0"
          using ‹a < b› ε' by (auto simp: δ_def intro!: strict_mono_onD[OF
assms(1)])
      next
        fix t assume t: "t ∈ {a..b}" "dist (f t) (f x) < δ"
        have "f t ≤ f b"
          using ‹a < b› t by (intro strict_mono_on_leD[OF assms(1)])
auto
        with t have "f b - f t < δ"
          by (simp add: dist_norm)
        hence "f t > f (b - ε')"
          unfolding δ_def by linarith
        hence "t > b - ε'"
          using t ε' by (subst (asm) strict_mono_on_less[OF assms(1)])
auto
        thus "dist (g (f t)) (g (f x)) ≤ ε"
          using ‹a < b› t ‹f t ≤ f b› ε' by (simp add: assms dist_norm)
      qed

    next

      assume x: "x ∈ {a<..<b}"
      define ε' where "ε' = min ε (min (x - a) (b - x) / 2)"
      have ε': "ε' > 0" "ε' ≤ ε" "ε' < x - a" "ε' < b - x"
        using ε x by (auto simp: ε'_def min_less_iff_disj)
      define δ where "δ = min (f x - f (x - ε')) (f (x + ε') - f x)"
      have *: "f (x - ε') < f x" "f x < f (x + ε')"
        using ε' by (intro strict_mono_onD[OF assms(1)]; simp)+
      show ?thesis
      proof (rule exI[of _ δ], safe)
        show "δ > 0"
          using * ε' by (auto simp add: δ_def)
      next
```

```
        fix t assume t: "t ∈ {a..b}" "dist (f t) (f x) < δ"
        have "dist (g (f t)) (g (f x)) ≤ ε'"
        proof (cases "t ≥ x")
          case True
          hence "f t ≥ f x"
            by (intro strict_mono_on_leD[OF assms(1)]) (use x t in auto)
          with t have "f t - f x < δ"
            by (simp add: dist_norm)
          hence "f t < f (x + ε')"
            unfolding δ_def by linarith
          hence "t < x + ε'"
            by (subst (asm) strict_mono_on_less[OF assms(1)]) (use x t
ε' in auto)
          thus ?thesis
            using x t True by (simp add: assms dist_norm)
        next
          case False
          hence "f t ≤ f x"
            by (intro strict_mono_on_leD[OF assms(1)]) (use x t in auto)
          with t have "f x - f t < δ"
            by (simp add: dist_norm)
          hence "f t > f (x - ε')"
            unfolding δ_def by linarith
          hence "t > x - ε'"
            by (subst (asm) strict_mono_on_less[OF assms(1)]) (use x t
ε' in auto)
          thus ?thesis
            using x t False by (simp add: assms dist_norm)
        qed
        also have "... ≤ ε"
          by fact
        finally show "dist (g (f t)) (g (f x)) ≤ ε" .
      qed
    qed
  qed
qed

lemma strict_mono_on_imp_continuous_on_inv_into:
  fixes f :: "real ⇒ real"
  assumes "strict_mono_on {a..b} f"
  shows   "continuous_on (f ` {a..b}) (inv_into {a..b} f)"
proof (rule strict_mono_on_imp_continuous_on_inv)
  show "inv_into {a..b} f (f x) = x" if "x ∈ {a..b}" for x
    using inv_into_f_f[OF strict_mono_on_imp_inj_on[OF assms] that] .
qed fact+
```

1.3 General lemmas about topology

```
lemma continuous_cong:
```

assumes *"eventually (λx. f x = g x) F" "f (netlimit F) = g (netlimit F)"*
shows *"continuous F f ⟷ continuous F g"*
unfolding *continuous_def* **using** *assms* **by** *(intro filterlim_cong) simp_all*

lemma *at_within_atLeastAtMost_eq_bot_iff_real:*
"at x within {a..b} = bot ⟷ x ∉ {a..b::real} ∨ a = b"
by *(cases a b rule: linorder_cases) (auto simp: trivial_limit_within islimpt_finite)*

lemma *eventually_in_pointed_at: "eventually (λx. x ∈ A - {y}) (at y within A)"*
by *(simp add: eventually_at_filter)*

lemma (**in** *order_topology*) *at_within_Icc_Icc_right:*
assumes *"a ≤ x" "x < b" "b ≤ c"*
shows *"at x within {a..c} = at x within {a..b}"*
by *(cases "x = a") (use assms* **in** *<simp_all add: at_within_Icc_at_right at_within_Icc_at>)*

lemma (**in** *order_topology*) *at_within_Icc_Icc_left:*
assumes *"a ≤ b" "b < x" "x ≤ c"*
shows *"at x within {a..c} = at x within {b..c}"*
by *(cases "x = c") (use assms* **in** *<simp_all add: at_within_Icc_at_left at_within_Icc_at>)*

lemma (**in** *order_topology*)
assumes *"a < b"*
shows *at_within_Ico_at_right: "at a within {a..<b} = at_right a"*
and *at_within_Ico_at_left: "at b within {a..<b} = at_left b"*
using *order_tendstoD(2)[OF tendsto_ident_at assms, of "{a<..}"]*
using *order_tendstoD(1)[OF tendsto_ident_at assms, of "{..<b}"]*
by *(auto intro!: order_class.order_antisym filter_leI*
simp: eventually_at_filter less_le
elim: eventually_elim2)

lemma (**in** *order_topology*)
assumes *"a < b"*
shows *at_within_Ioc_at_right: "at a within {a<..b} = at_right a"*
and *at_within_Ioc_at_left: "at b within {a<..b} = at_left b"*
using *order_tendstoD(2)[OF tendsto_ident_at assms, of "{a<..}"]*
using *order_tendstoD(1)[OF tendsto_ident_at assms, of "{..<b}"]*
by *(auto intro!: order_class.order_antisym filter_leI*
simp: eventually_at_filter less_le

```
    elim: eventually_elim2)

lemma (in order_topology) at_within_Ico_at: "a < x ⟹ x < b ⟹ at
x within {a..<b} = at x"
  by (rule at_within_open_subset[where S="{a<..<b}"]) auto

lemma (in order_topology) at_within_Ioc_at: "a < x ⟹ x < b ⟹ at
x within {a<..b} = at x"
  by (rule at_within_open_subset[where S="{a<..<b}"]) auto

lemma (in order_topology) at_within_Ioo_at: "a < x ⟹ x < b ⟹ at
x within {a<..<b} = at x"
  by (rule at_within_open_subset[where S="{a<..<b}"]) auto

lemma (in order_topology) at_within_Icc_Ico:
  assumes "a ≤ x" "x < b"
  shows   "at x within {a..b} = at x within {a..<b}"
  by (cases "x = a")
     (use assms in ‹simp_all add: at_within_Icc_at_right at_within_Ico_at_right
                                  at_within_Ico_at at_within_Icc_at›)

lemma (in order_topology) at_within_Icc_Ioc:
  assumes "a < x" "x ≤ b"
  shows   "at x within {a..b} = at x within {a<..b}"
  by (cases "x = b")
     (use assms in ‹simp_all add: at_within_Icc_at_left at_within_Ioc_at_left
                                  at_within_Ioc_at at_within_Icc_at›)
```

1.4 General lemmas about real functions

```
lemma isCont_real_If_combine:
  fixes x :: real
  assumes [simp]: "f x = h x" "g x = h x"
  assumes contf: "continuous (at_left x) f"
  assumes contg: "continuous (at_right x) g"
  assumes f: "eventually (λy. h y = f y) (at_left x)"
  assumes g: "eventually (λy. h y = g y) (at_right x)"
  shows   "continuous (at x) h"
  unfolding continuous_at_split
proof
  have "continuous (at_left x) f ⟷ continuous (at_left x) h"
    by (intro continuous_cong eventually_mono[OF f]) (auto simp: Lim_ident_at)
  with contf show "continuous (at_left x) h" by blast
next
  have "continuous (at_right x) g ⟷ continuous (at_right x) h"
    by (intro continuous_cong eventually_mono[OF g]) (auto simp: Lim_ident_at)
  with contg show "continuous (at_right x) h" by blast
qed
```

```
lemma continuous_on_real_If_combine:
  fixes f g :: "real ⇒ 'a :: topological_space"
  assumes "continuous_on {a..b} f"
  assumes "continuous_on {b..c} g"
  assumes "f b = g b" "a ≤ b" "b ≤ c"
  defines "h ≡ (λx. if x ≤ b then f x else g x)"
  shows   "continuous_on {a..c} h"
proof (cases "a = b ∨ b = c")
  case True
  thus ?thesis
  proof
    assume [simp]: "a = b"
    have "continuous_on {a..c} g"
      using assms by (simp add: continuous_on_imp_continuous_within)
    also have "?this ⟷ continuous_on {a..c} h"
      by (intro continuous_on_cong) (auto simp: assms)
    finally show ?thesis .
  next
    assume [simp]: "b = c"
    have "continuous_on {a..c} f"
      using assms by (simp add: continuous_on_imp_continuous_within)
    also have "?this ⟷ continuous_on {a..c} h"
      by (intro continuous_on_cong) (auto simp: assms)
    finally show ?thesis .
  qed
next
  case False
  hence abc: "a < b" "b < c"
    using assms by auto
  note [simp] = at_within_atLeastAtMost_eq_bot_iff_real Lim_ident_at
  have "continuous (at x within {a..c}) h" if x: "x ∈ {a..c}" for x
  proof (cases x b rule: linorder_cases)
    case [simp]: equal
    have "continuous (at b) h"
      unfolding continuous_at_split
    proof
      have "continuous (at b within {a..b}) f"
        using assms x by (simp add: continuous_on_imp_continuous_within)
      also have "at b within {a..b} = at_left b"
        using abc by (simp add: at_within_Icc_at_left)
      also have ev: "eventually (λx. x < b) (at_left b)"
        using eventually_at_topological by blast
      have "continuous (at_left b) f ⟷ continuous (at_left b) h"
        using assms by (intro continuous_cong eventually_mono[OF ev])
(auto simp: h_def)
      finally show "continuous (at_left b) h" .
    next
      have "continuous (at b within {b..c}) g"
```

```
      using assms x by (simp add: continuous_on_imp_continuous_within)
    also have "at b within {b..c} = at_right b"
      using abc by (simp add: at_within_Icc_at_right)
    also have ev: "eventually (λx. x > b) (at_right b)"
      using eventually_at_topological by blast
    have "continuous (at_right b) g ⟷ continuous (at_right b) h"
      using assms by (intro continuous_cong eventually_mono[OF ev])
(auto simp: h_def)
    finally show "continuous (at_right b) h" .
  qed
  thus ?thesis
    using continuous_at_imp_continuous_at_within local.equal by blast
next
  case less
  have "continuous (at x within {a..b}) f"
    using assms less x by (simp add: continuous_on_imp_continuous_within)
  also have "eventually (λy. y ∈ {a..b} - {x}) (at x within {a..b})"
    by (rule eventually_in_pointed_at)
  hence "eventually (λy. f y = h y) (at x within {a..b})"
    by eventually_elim (auto simp: h_def)
  hence "continuous (at x within {a..b}) f ⟷ continuous (at x within
{a..b}) h"
    using assms less x by (intro continuous_cong) simp_all
  also have "at x within {a..b} = at x within {a..c}"
    by (rule sym, rule at_within_Icc_Icc_right) (use x less assms in
auto)
  finally show ?thesis .
next
  case greater
  have "continuous (at x within {b..c}) g"
    using assms greater x by (simp add: continuous_on_imp_continuous_within)
  also {
    have "eventually (λy. y ∈ {b<..c} - {x}) (at x within {b<..c})"
      by (rule eventually_in_pointed_at)
    also have "at x within {b<..c} = at x within {b..c}"
      using greater assms x by (metis atLeastAtMost_iff at_within_Icc_Ioc)
    finally have "eventually (λy. g y = h y) (at x within {b..c})"
      by eventually_elim (use greater in ‹auto simp: h_def›)
  }
  hence "continuous (at x within {b..c}) g ⟷ continuous (at x within
{b..c}) h"
    using assms greater x by (intro continuous_cong) simp_all
  also have "at x within {b..c} = at x within {a..c}"
    by (rule sym, rule at_within_Icc_Icc_left) (use x greater assms
in auto)
  finally show ?thesis .
qed
thus ?thesis
  using continuous_on_eq_continuous_within by blast
```

qed

lemma *continuous_on_real_If_combine':*
fixes *f g :: "real ⇒ 'a :: topological_space"*
assumes *"continuous_on {a..b} f"*
assumes *"continuous_on {b..c} g"*
assumes *"f b = g b" "a ≤ b" "b ≤ c"*
defines *"h ≡ (λx. if x < b then f x else g x)"*
shows *"continuous_on {a..c} h"*
proof -
have *"continuous_on {-c..-a} ((λx. if x ≤ -b then (g ∘ uminus) x else (f ∘ uminus) x))"*
(is *"continuous_on _ ?h'"***)**
using *assms*
by *(intro continuous_on_real_If_combine continuous_on_compose continuous_intros) auto*
hence *"continuous_on {a..c} (?h' ∘ uminus)"*
by *(intro continuous_on_compose continuous_intros) auto*
also have *"?h' ∘ uminus = h"*
by *(auto simp: h_def fun_eq_iff)*
finally show *?thesis* **.**
qed

lemma *continuous_on_linepath [continuous_intros]:*
assumes *"continuous_on A f" "continuous_on A g" "continuous_on A h"*
shows *"continuous_on A (λx. linepath (f x) (g x) (h x))"*
unfolding *linepath_def* **by** *(intro continuous_intros assms)*

1.5 Rounding and fractional part

lemma *frac_of_Int [simp]: "x ∈ ℤ ⟹ frac x = 0"*
by *(subst frac_eq_0_iff)*

lemma *floor_less_not_int: "x ∉ ℤ ⟹ of_int (floor x) < x"*
by *(metis Ints_of_int floor_correct order_less_le)*

lemma *less_ceiling_not_int: "x ∉ ℤ ⟹ of_int (ceiling x) > x"*
by *(meson floor_less_iff floor_less_not_int less_ceiling_iff)*

lemma *image_frac_atLeastLessThan:*
assumes *"y ≥ x + (1 :: 'a :: floor_ceiling)"*
shows *"frac ‘ {x..<y} = {0..<1}"*
proof *safe*
fix *t :: 'a* **assume** *t: "t ∈ {0..<1}"*

```
  define u where "u = (if t ≥ frac x then t + of_int ⌊x⌋ else t + of_int
⌊x⌋ + 1)"
  have "frac u = t"
    using t by (auto simp: u_def frac_def floor_unique)
  moreover {
    have "x ≤ t + of_int ⌊x⌋ + 1"
      using assms t unfolding atLeastLessThan_iff by linarith
    moreover have "t + of_int ⌊x⌋ < y"
      using assms t unfolding atLeastLessThan_iff by linarith
    ultimately have "u ∈ {x..<y}"
      using assms by (auto simp: u_def frac_def)
  }
  ultimately show "t ∈ frac ‘ {x..<y}"
    by blast
qed (auto simp: frac_lt_1)


lemma image_frac_atLeastAtMost:
  assumes "y ≥ x + 1"
  shows   "frac ‘ {x..y} = {0..<1}"
proof
  have "{0..<1} = frac ‘ {x..<y}"
    by (rule sym, intro image_frac_atLeastLessThan assms)
  also have "... ⊆ frac ‘ {x..y}"
    by (intro image_mono) auto
  finally show "{0..<1} ⊆ frac ‘ {x..y}" .
qed (auto simp: frac_lt_1)




lemma tendsto_frac_real [tendsto_intros]:
  assumes "(x :: real) ∉ ℤ"
  shows   "(frac ⟶ frac x) (at x within A)"
  using assms continuous_at_imp_continuous_at_within continuous_frac continuous_within
by blast

lemma tendsto_frac_at_left_int_real:
  assumes "(x :: real) ∈ ℤ"
  shows   "(frac ⟶ 1) (at_left x)"
proof -
  have "((λy. y - real_of_int ⌊x⌋ + 1) ⟶ 1) (at_left x)"
    by (rule tendsto_eq_intros refl)+ (use assms in ‹auto elim!: Ints_cases›)
  moreover have "eventually (λy. y ∈ {x-1<..<x}) (at_left x)"
    using eventually_at_left_real by force
  hence "eventually (λy. frac y = y - real_of_int ⌊x⌋ + 1) (at_left x)"
  proof eventually_elim
    case (elim y)
    show "frac y = y - real_of_int ⌊x⌋ + 1"
```

```
    using assms elim by (subst frac_unique_iff) (auto elim!: Ints_cases)
  qed
  ultimately show ?thesis
    by (simp add: filterlim_cong)
qed

lemma filterlim_at_frac_at_left_int_real:
  assumes "(x :: real) ∈ ℤ"
  shows   "filterlim frac (at_left 1) (at_left x)"
  unfolding filterlim_at
proof
  show "∀\<^sub>F y in at_left x. frac y ∈ {..<1} ∧ frac y ≠ 1"
  proof (intro always_eventually allI)
    fix y :: real
    show "frac y ∈ {..<1} ∧ frac y ≠ 1"
      using frac_lt_1[of y] by auto
  qed
qed (auto intro!: tendsto_frac_at_left_int_real assms)

lemma tendsto_frac_at_right_int_real:
  assumes "(x :: real) ∈ ℤ"
  shows   "(frac ⟶ 0) (at_right x)"
proof -
  have "((λy. y - real_of_int ⌊x⌋) ⟶ 0) (at_right x)"
    by (rule tendsto_eq_intros refl)+ (use assms in <auto elim!: Ints_cases>)
  moreover have "eventually (λy. y ∈ {x<..<x+1}) (at_right x)"
    using eventually_at_right_real by force
  hence "eventually (λy. frac y = y - real_of_int ⌊x⌋) (at_right x)"
  proof eventually_elim
    case (elim y)
    show "frac y = y - real_of_int ⌊x⌋"
      using assms elim by (subst frac_unique_iff) (auto elim!: Ints_cases)
  qed
  ultimately show ?thesis
    by (simp add: filterlim_cong)
qed

lemma filterlim_at_frac_at_right_int_real [tendsto_intros]:
  assumes "(x :: real) ∈ ℤ"
  shows   "filterlim frac (at_right 0) (at_right x)"
  unfolding filterlim_at
proof
  have "eventually (λy. y ∈ {x<..<x+1}) (at_right x)"
    using eventually_at_right_real by force
  thus "∀\<^sub>F x in at_right x. frac x ∈ {0<..} ∧ frac x ≠ 0"
  proof eventually_elim
    case (elim y)
    hence "y ∉ ℤ"
      using assms by (auto elim!: Ints_cases)
```

```
    thus "frac y ∈ {0<..} ∧ frac y ≠ 0"
      using frac_ge_0[of x] by auto
  qed
qed (auto intro!:  tendsto_frac_at_right_int_real assms)


lemma continuous_on_frac_real:
  assumes "continuous_on {0..1} f" "f 0 = f 1"
  shows   "continuous_on A (λx::real. f (frac x))"
proof -
  have isCont_f: "isCont f x" if "x ∈ {0<..<1}" for x
    by (rule continuous_on_interior[OF assms(1)]) (use that in auto)
  note [continuous_intros] = continuous_at_compose[OF _ isCont_f, unfolded
o_def]

  have contfl: "(f ⟶ f 0) (at_right 0)"
    using assms(1) by (simp add: continuous_on_Icc_at_rightD)
  have contfr: "(f ⟶ f 1) (at_left 1)"
    using assms(1) by (simp add: continuous_on_Icc_at_leftD)
  note tendsto_intros = filterlim_compose[OF contfr] filterlim_compose[OF
contfl]

  have "continuous (at x) (λx. f (frac x))" for x
  proof (cases "x ∈ ℤ")
    case True
    have "((λx. f (frac x)) ⟶ f 1) (at_left x)"
      by (rule tendsto_intros filterlim_at_frac_at_left_int_real True)+
    moreover have "((λx. f (frac x)) ⟶ f 0) (at_right x)"
      by (rule tendsto_intros filterlim_at_frac_at_right_int_real True)+
    ultimately show ?thesis
      using assms(2) True unfolding continuous_at_split unfolding continuous_def
      by (auto simp: Lim_ident_at elim!: Ints_cases)
  next
    case False
    have "x < 1 + real_of_int ⌊x⌋"
      by linarith
    hence "continuous (at x) (λy. f (y - ⌊x⌋))"
      using floor_less_not_int[of x] False
      by (intro continuous_intros) (auto simp: algebra_simps)
    also have "eventually (λy. y ∈ {floor x<..<ceiling x}) (nhds x)"
      using eventually_floor_less[OF filterlim_ident False]
            eventually_less_ceiling[OF filterlim_ident False]
      by eventually_elim auto
    hence "eventually (λy. frac y = y - ⌊x⌋) (nhds x)"
    proof eventually_elim
      case (elim y)
      hence "y - real_of_int ⌊x⌋ < 1"
        unfolding greaterThanLessThan_iff using ceiling_diff_floor_le_1[of
x] by linarith
```

```
    thus ?case using elim
      by (subst frac_unique_iff) auto
  qed
  hence "eventually (λy. f (y - ⌊x⌋) = f (frac y)) (at x)"
    unfolding eventually_at_filter by eventually_elim (auto simp: frac_def)
  hence "isCont (λy. f (y - ⌊x⌋)) x = isCont (λy. f (frac y)) x"
    by (intro continuous_cong) (auto simp: frac_def)
  finally show ?thesis .
qed
thus ?thesis
  using continuous_at_imp_continuous_on by blast
qed

lemma continuous_on_frac_real':
  assumes "continuous_on {0..1} f" "continuous_on A g" "f 0 = f 1"
  shows   "continuous_on A (λx. f (frac (g x :: real)))"
  using continuous_on_compose2[OF continuous_on_frac_real[OF assms(1,3)]
assms(2)] by blast
```

1.6 General lemmas about paths

```
lemma linepath_scaleR: "(*R) c ∘ linepath a b = linepath (c *R a) (c
*R b)"
  by (simp add: linepath_def fun_eq_iff algebra_simps)


lemma linepath_mult_complex: "(*) c ∘ linepath a b = linepath (c * a)
(c * b :: complex)"
  by (simp add: linepath_def fun_eq_iff algebra_simps)


lemma linepath_translate: "(+) c ∘ linepath a b = linepath (c + a) (c
+ b)"
  by (simp add: linepath_def fun_eq_iff algebra_simps)


lemma part_circlepath_translate:
  "(+) c ∘ part_circlepath x r a b = part_circlepath (c + x) r a b"
  by (simp add: part_circlepath_def fun_eq_iff algebra_simps)


lemma circlepath_translate:
  "(+) c ∘ circlepath x r = circlepath (c + x) r"
  by (simp add: circlepath_def part_circlepath_translate)


lemma rectpath_translate:
  "(+) c ∘ rectpath a b = rectpath (c + a) (c + b)"
  by (simp add: rectpath_def linepath_translate Let_def path_compose_join
```

```
plus_complex.ctr)


lemma path_image_cong: "(⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ path_image
p = path_image q"
  by (auto simp: path_image_def)


lemma path_cong: "(⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ path p = path q"
  unfolding path_def by (intro continuous_on_cong) auto


lemma simple_path_cong:
  shows "(⋀x. x ∈ {0..1} ⟹ f x = g x) ⟹ simple_path f ⟷ simple_path
g"
  unfolding simple_path_def loop_free_def
  by (intro arg_cong2[of _ _ _ _ "(∧)"] path_cong) auto


lemma simple_path_reversepath_iff: "simple_path (reversepath g) ⟷
simple_path g"
  using simple_path_reversepath[of g] simple_path_reversepath[of "reversepath
g"]
  by auto


lemma path_image_loop:
  assumes "pathstart p = pathfinish p"
  shows   "path_image p = p ‘ {0..<1}"
  unfolding path_image_def
proof safe
  fix x :: real assume x: "x ∈ {0..1}"
  have "(if x = 1 then 0 else x) ∈ {0..<1}" "p x = p (if x = 1 then 0
else x)"
    using assms x by (auto simp: pathstart_def pathfinish_def)
  thus "p x ∈ p ‘ {0..<1}"
    by blast
qed auto


lemma simple_pathD:
  assumes "simple_path p" "x ∈ {0..1}" "y ∈ {0..1}" "x ≠ y" "p x = p
y"
  shows   "{x, y} = {0, 1}"
  using assms unfolding simple_path_def loop_free_def by blast


lemmas [trans] = homotopic_loops_trans
```

```
proposition homotopic_loops_reparametrize:
  assumes "path p" "pathstart p = pathfinish p"
      and pips: "path_image p ⊆ s"
      and contf: "continuous_on {0..1} f"
      and q: "⋀t. t ∈ {0..1} ⟹ q t = p (frac (f t))"
      and closed: "f 1 = f 0 + 1"
    shows "homotopic_loops s p q"
proof -
  note [continuous_intros] = continuous_on_frac_real'[OF continuous_on_path[OF
‹path p›]]
  note [continuous_intros] = continuous_on_compose2[OF contf]
  define h :: "real × real ⇒ 'a" where "h = (λ(u,v). p (frac (linepath
v (f v) u)))"

  have [simp]: "p (frac t) = p t" if "t ∈ {0..1}" for t
    using that assms(2) frac_eq[of t]
    by (cases "t = 1") (auto simp: pathstart_def pathfinish_def)

  show ?thesis
    unfolding homotopic_loops
  proof (rule exI[of _ h]; safe)
    fix v :: real assume v: "v ∈ {0..1}"
    show "h (0, v) = p v" and "h (1, v) = q v"
      using q v by (simp_all add: h_def linepath_def)
  next
    fix u v :: real assume uv: "u ∈ {0..1}" "v ∈ {0..1}"
    have "h (u, v) ∈ path_image p"
      by (auto simp: h_def path_image_def intro!: imageI less_imp_le[OF
frac_lt_1])
    also have "... ⊆ s"
      by fact
    finally show "h (u, v) ∈ s" .
  next
    fix t :: real assume t: "t ∈ {0..1}"
    show "pathfinish (h ∘ Pair t) = pathstart (h ∘ Pair t)"
      using t by (auto simp: h_def pathfinish_def pathstart_def linepath_def

                             closed algebra_simps frac_def)
  next
    show "continuous_on ({0..1}×{0..1}) h"
      unfolding h_def case_prod_unfold using ‹pathstart p = pathfinish
p›
      by (auto intro!: continuous_intros order.refl continuous_on_fst
less_imp_le[OF frac_lt_1]
               simp: pathstart_def pathfinish_def)
  qed
qed
```

1.7 Some facts about betweenness

```
lemma between_conv_linepath:
  fixes a b c :: "'a :: euclidean_space"
  assumes "between (a, c) b"
  shows   "b = linepath a c (dist a b / dist a c)" (is "_ = ?b'")
proof (cases "a = c")
  case False
  from assms obtain u where u: "u ∈ {0..1}" "b = (1 - u) *R a + u *R
c"
    using assms by (auto simp: between_def closed_segment_def)
  have "dist a b = norm (u *R (a - c))"
    unfolding scaleR_diff_right by (simp add: u dist_norm algebra_simps)
  hence ab: "dist a b = u * dist a c"
    using u(1) by (simp add: dist_norm norm_minus_commute)
  define t where "t = dist a b / dist a c"
  have "linepath a c t =
          (1 - dist a b / dist a c) *R a + (dist a b / dist a c) *R c"
    by (simp add: ab linepath_def t_def)
  also have "(1 - dist a b / dist a c) = 1 - u"
    using False by (simp add: ab)
  also have "dist a b / dist a c = u"
    using False by (simp add: ab)
  also have "(1 - u) *R a + u *R c = b"
    by (simp add: u)
  finally show ?thesis
    by (simp add: u t_def)
qed (use assms in auto)

lemma between_trans1:
  assumes "between (a, c) b" "between (b, d) c" "b ≠ c" "a ≠ d"
  shows   "between (a, d) b"
proof (cases "distinct [a, b, c, d]")
  case False
  with assms show ?thesis
    by (auto simp: between_def)
next
  case True
  from assms(1) obtain u where u: "u ∈ {0..1}" "b = (1 - u) *R a + u
*R c"
    by (auto simp: between_def closed_segment_def)
  from assms(2) obtain v where v: "v ∈ {0..1}" "c = (1 - v) *R b + v
*R d"
    by (auto simp: between_def closed_segment_def)

  have "u ≠ 0" "u ≠ 1" "v ≠ 0" "v ≠ 1"
    using u v True by auto
  with u(1) v(1) have uv: "u ∈ {0<..<1}" "v ∈ {0<..<1}"
    by auto
```

```
  define z where "z = 1 - u * (1 - v)"
  define t where "t = (u * v) / z"
  have "u * (1 - v) < 1 * 1"
    using uv by (intro mult_strict_mono) auto
  hence *: "z > 0"
    unfolding z_def by auto

  have "b = (1 - u) *R a + (u * (1 - v)) *R b + (u * v) *R d"
    by (subst u, subst v) (simp add: algebra_simps)
  hence "z *R b = (1 - u) *R a + (u * v) *R d"
    by (simp add: algebra_simps z_def)
  hence "inverse z *R z *R b = inverse z *R ((1 - u) *R a + (u * v) *R
d)"
    by (simp only: )
  also have "inverse z *R z *R b = b"
    using * by (simp add: field_simps)
  also have "inverse z *R ((1 - u) *R a + (u * v) *R d) =
             ((1 - u) / z) *R a + ((u * v) / z) *R d"
    using * by (simp add: algebra_simps divide_inverse)
  also have "(1 - u) / z = 1 - t"
    using * by (simp add: field_simps t_def z_def)
  also have "(u * v) / z = t"
    by (simp add: t_def)
  finally have "b = (1 - t) *R a + t *R d" .

  moreover have "t ≥ 0"
    unfolding t_def using u(1) v(1) *
    by (intro divide_nonneg_pos mult_nonneg_nonneg) auto
  moreover have "(1 - u) / z ≥ 0"
    using u(1) * by (intro divide_nonneg_pos) auto
  with ‹(1 - u) / z = 1 - t› have "t ≤ 1"
    by simp
  ultimately show ?thesis
    unfolding between_def prod.case closed_segment_def by blast
qed

lemma between_trans2:
  "between (a, c) b ⟹ between (b, d) c ⟹ b ≠ c ⟹ a ≠ d ⟹ between
(a, d) c"
  using between_trans1[of d b c a] by (simp add: between_commute)

lemma between_trans1':
  assumes "between (a :: 'a :: euclidean_space, c) b" "between (b, d)
c" "b ≠ c"
  shows   "between (a, d) b"
proof (cases "a = d")
  case True
  with assms show ?thesis
    using between_antisym between_commute by metis
```

```
qed (use between_trans1[OF assms] in simp)

lemma between_trans2':
  assumes "between (a :: 'a :: euclidean_space, c) b" "between (b, d)
c" "b ≠ c"
  shows   "between (a, d) c"
proof (cases "a = d")
  case True
  with assms show ?thesis
    using between_antisym between_commute by metis
qed (use between_trans2[OF assms] in simp)
```

The following expresses successive betweenness: e.g. `betweens [a,b,c,d]` means that the points `a`, `b`, `c`, `d` all lie on the same line in that order. Note that we do not have strict betweenness, i.e. some of the points might be identical.

```
fun betweens :: "'a :: euclidean_space list ⇒ bool" where
  "betweens (x # y # z # xs) ⟷ between (x, z) y ∧ betweens (y # z #
xs)"
| "betweens _ ⟷ True"
```

1.8 Simple loops and orientation

A simple loop is a continuous path whose start and end point coincide and which never intersects itself. In e.g. the complex plane, such a simple loop partitions the full complex plane into an inner and outer part by the Jordan Curve Theorem.

```
definition simple_loop :: "(real ⇒ 'a :: topological_space) ⇒ bool"
  where "simple_loop p ⟷ simple_path p ∧ pathstart p = pathfinish
p"

lemma simple_loop_reversepath [simp]: "simple_loop (reversepath p) ⟷
simple_loop p"
  by (auto simp: simple_loop_def simple_path_reversepath_iff)
```

The winding number of a simple loop is either 1 for any point inside the loop or -1 for any point inside the loop (and of course 0 for all the points outside, and undefined for all the points on it).

We refer to the winding number of the points inside a simple loop as their *orientation*, and we call simple loops with orientation 1 *counter-clockwise* and those with orientation -1 *clockwise*.

```
definition simple_loop_ccw :: "(real ⇒ complex) ⇒ bool" where
  "simple_loop_ccw p ⟷ simple_loop p ∧ (∃z. z ∉ path_image p ∧ winding_number
p z = 1)"

definition simple_loop_cw :: "(real ⇒ complex) ⇒ bool" where
```

```
  "simple_loop_cw p ⟷ simple_loop p ∧ (∃z. z ∉ path_image p ∧ winding_number
p z = -1)"

definition simple_loop_orientation :: "(real ⇒ complex) ⇒ int" where
  "simple_loop_orientation p =
     (if simple_loop_ccw p then 1 else if simple_loop_cw p then -1 else
0)"

lemma simple_loop_ccwI:
  "simple_loop p ⟹ z ∉ path_image p ⟹ winding_number p z = 1 ⟹
simple_loop_ccw p"
  unfolding simple_loop_ccw_def by auto

lemma simple_loop_cwI:
  "simple_loop p ⟹ z ∉ path_image p ⟹ winding_number p z = -1 ⟹
simple_loop_cw p"
  unfolding simple_loop_cw_def by auto

lemma simple_path_not_cw_and_ccw: "¬simple_loop_cw p ∨ ¬simple_loop_ccw
p"
  unfolding simple_loop_cw_def simple_loop_ccw_def simple_loop_def
  by (metis ComplI UnE inside_Un_outside one_neq_neg_one simple_closed_path_winding_number_
           simple_path_def winding_number_zero_in_outside zero_neq_neg_one
zero_neq_one)

lemma simple_loop_cw_or_ccw:
  assumes "simple_loop p"
  shows   "simple_loop_cw p ∨ simple_loop_ccw p"
  using assms unfolding simple_loop_cw_def simple_loop_ccw_def simple_loop_def
  by (metis Compl_iff UnCI inside_Un_outside simple_closed_path_winding_number_inside
       simple_closed_path_wn3)

lemma simple_loop_ccw_conv_cw:
  assumes "simple_loop p"
  shows   "simple_loop_ccw p ⟷ ¬simple_loop_cw p"
  using assms simple_path_not_cw_and_ccw simple_loop_cw_or_ccw by blast

lemma simple_loop_orientation_eqI:
  assumes "simple_loop p" "z ∉ path_image p"
  assumes "winding_number p z ∈ {-1, 1}"
  shows   "simple_loop_orientation p = winding_number p z"
  unfolding simple_loop_orientation_def
  using assms simple_loop_ccwI simple_loop_ccw_conv_cw simple_loop_cwI
by force

lemma simple_loop_winding_number_cases:
  assumes "simple_loop p" "z ∉ path_image p"
  shows   "winding_number p z = (if z ∈ inside (path_image p) then simple_loop_orientation
p else 0)"
```

```
proof (cases "z ∈ inside (path_image p)")
  case True
  hence "winding_number p z ∈ {-1, 1}"
    using simple_closed_path_winding_number_inside[of p] assms
    unfolding simple_loop_def by fast
  hence "simple_loop_orientation p = winding_number p z"
    by (intro simple_loop_orientation_eqI) (use assms in auto)
  thus ?thesis
    using True by simp
next
  case False
  hence "winding_number p z = 0"
    using assms unfolding simple_loop_def
    by (simp add: inside_outside simple_path_imp_path winding_number_zero_in_outside)
  thus ?thesis
    using False by auto
qed

lemma simple_loop_orientation_eq_0_iff [simp]:
  "simple_loop_orientation p = 0 ⟷ ¬simple_loop p"
  using simple_loop_cw_or_ccw[of p]
  by (auto simp: simple_loop_orientation_def simple_loop_cw_def simple_loop_ccw_def)

lemma simple_loop_ccw_reversepath_aux:
  assumes "simple_loop_ccw p"
  shows   "simple_loop_cw (reversepath p)"
proof -
  from assms obtain z where *: "simple_loop p" "z ∉ path_image p" "winding_number
p z = 1"
    by (auto simp: simple_loop_ccw_def)
  moreover from * have "winding_number (reversepath p) z = -winding_number
p z"
    by (subst winding_number_reversepath) (auto simp: simple_path_imp_path
simple_loop_def)
  ultimately show ?thesis using *
    by (auto simp: simple_loop_cw_def simple_loop_def simple_path_reversepath)
qed

lemma simple_loop_cw_reversepath_aux:
  assumes "simple_loop_cw p"
  shows   "simple_loop_ccw (reversepath p)"
proof -
  from assms obtain z where *: "simple_loop p" "z ∉ path_image p" "winding_number
p z = -1"
    by (auto simp: simple_loop_cw_def)
  moreover from * have "winding_number (reversepath p) z = -winding_number
p z"
    by (subst winding_number_reversepath) (auto simp: simple_path_imp_path
simple_loop_def)
```

```
  ultimately show ?thesis using *
    by (auto simp: simple_loop_ccw_def simple_loop_def simple_path_reversepath)
qed

lemma simple_loop_cases: "simple_loop_ccw p ∨ simple_loop_cw p ∨ ¬simple_loop
p"
  using simple_loop_cw_or_ccw[of p] by blast

lemma simple_loop_cw_reversepath [simp]: "simple_loop_cw (reversepath
p) ⟷ simple_loop_ccw p"
  using simple_loop_ccw_reversepath_aux reversepath_reversepath simple_loop_cw_reversepath_a
  by metis

lemma simple_loop_ccw_reversepath [simp]: "simple_loop_ccw (reversepath
p) ⟷ simple_loop_cw p"
  using simple_loop_ccw_reversepath_aux reversepath_reversepath simple_loop_cw_reversepath_a
  by metis

lemma simple_loop_orientation_reversepath [simp]:
  "simple_loop_orientation (reversepath p) = -simple_loop_orientation
p"
  using simple_path_not_cw_and_ccw[of p] by (auto simp: simple_loop_orientation_def)

lemma simple_loop_orientation_cases:
  assumes "simple_loop p"
  shows   "simple_loop_orientation p ∈ {-1, 1}"
  using simple_loop_cases[of p] assms by (auto simp: simple_loop_orientation_def)

lemma inside_simple_loop_iff:
  assumes "simple_loop p"
  shows   "z ∈ inside (path_image p) ⟷ z ∉ path_image p ∧ winding_number
p z ≠ 0"
  using assms
  by (smt (verit, best) disjoint_iff_not_equal inside_no_overlap norm_zero
of_int_0
    simple_closed_path_norm_winding_number_inside simple_loop_def simple_loop_winding_numbe

lemma outside_simple_loop_iff:
  assumes "simple_loop p"
  shows   "z ∈ outside (path_image p) ⟷ z ∉ path_image p ∧ winding_number
p z = 0"
  using assms by (metis Compl_iff Un_iff inside_Un_outside inside_outside
inside_simple_loop_iff)
```

1.9 More about circular arcs

```
lemma part_circlepath_altdef:
  "part_circlepath z r a b = (λt. z + rcis r (linepath a b t))"
  unfolding part_circlepath_def rcis_def cis_conv_exp ..
```

```
lemma part_circlepath_cong:
  assumes "x = x'" "r = r'" "cis a' = cis a" "b' = a' + b - a"
  shows   "part_circlepath x r a b = part_circlepath x' r' a' b'"
  by (simp add: part_circlepath_altdef rcis_def linepath_def algebra_simps
assms
          flip: cis_mult cis_divide)

lemma part_circlepath_empty: "part_circlepath x r a a = linepath (x +
rcis r a) (x + rcis r a)"
  by (auto simp: part_circlepath_altdef linepath_def algebra_simps fun_eq_iff)

lemma part_circlepath_radius_0 [simp]: "part_circlepath x 0 a b = linepath
x x"
  by (simp add: part_circlepath_altdef linepath_def)

lemma part_circlepath_scaleR:
  "(*_R) c ∘ part_circlepath x r a b = part_circlepath (c *_R x) (c * r)
a b"
proof (cases "c = 0")
  assume "c ≠ 0"
  thus ?thesis
    by (simp add: part_circlepath_altdef fun_eq_iff algebra_simps linepath_def
rcis_def cis_Arg
                  complex_sgn_def scaleR_conv_of_real flip: cis_divide
cis_mult)
qed (auto simp: fun_eq_iff part_circlepath_altdef)

lemma part_circlepath_mult_complex:
  "(*) c ∘ part_circlepath x r a b = part_circlepath (c * x :: complex)
(norm c * r) (a + Arg c) (b + Arg c)"
proof (cases "c = 0")
  assume "c ≠ 0"
  thus ?thesis
    by (simp add: part_circlepath_altdef fun_eq_iff algebra_simps linepath_def
rcis_def cis_Arg
                  complex_sgn_def scaleR_conv_of_real flip: cis_divide
cis_mult)
qed (auto simp: fun_eq_iff part_circlepath_altdef)

lemma part_circlepath_mult_complex':
  assumes "cis a' = cis (a + Arg c)" "b' = a' + b - a"
  shows   "(*) c ∘ part_circlepath x r a b = part_circlepath (c * x ::
complex) (norm c * r) a' b'"
  unfolding part_circlepath_mult_complex by (rule part_circlepath_cong)
(use assms in auto)

lemma circlepath_altdef: "circlepath x r t = x + rcis r (2 * pi * t)"
  by (simp add: circlepath_def part_circlepath_altdef mult_ac)
```

```
lemma reversepath_circlepath: "reversepath (circlepath x r) = part_circlepath
x r (2 * pi) 0"
  by (simp add: circlepath_def)

lemma pathstart_part_circlepath': "pathstart (part_circlepath z r a b)
= z + rcis r a"
  and pathfinish_part_circlepath': "pathfinish (part_circlepath z r a
b) = z + rcis r b"
  unfolding part_circlepath_altdef by (simp_all add: pathstart_def pathfinish_def
linepath_def)
```

1.10 Reparametrisation of loops by shifting

```
lemma shiftpath_loop_altdef:
  assumes "pathstart p = pathfinish p" "x ∈ {0..1}" "a ∈ {0..1}"
  shows   "shiftpath a p x = p (frac (x + a))"
proof -
  consider "x + a < 1" | "x + a = 1" | "x + a > 1" "x + a < 2" | "x +
a = 2"
    using assms(2,3) by fastforce
  thus ?thesis
  proof cases
    case 3
    hence [simp]: "frac (a + x) = x + a - 1"
      using assms unfolding atLeastAtMost_iff by (subst frac_unique_iff)
auto
    show ?thesis using assms 3
      by (auto simp: shiftpath_def pathstart_def pathfinish_def algebra_simps)
  qed (use assms frac_eq[of "a + x"]
       in ‹auto simp: shiftpath_def algebra_simps pathstart_def pathfinish_def›)
qed

lemma homotopic_loops_shiftpath_left:
  assumes "path p" "path_image p ⊆ A" "pathstart p = pathfinish p" "x
∈ {0..1}"
  shows   "homotopic_loops A (shiftpath x p) p"
proof (rule homotopic_loops_sym, rule homotopic_loops_reparametrize)
  show "continuous_on {0..1} ((+) x)"
    by (intro continuous_intros)
  show "shiftpath x p t = p (frac (x + t))" if "t ∈ {0..1}" for t
    using that assms by (simp add: shiftpath_loop_altdef add_ac)
qed (use assms in auto)

lemma homotopic_loops_shiftpath_right:
  assumes "path p" "path_image p ⊆ A" "pathstart p = pathfinish p" "x
∈ {0..1}"
  shows   "homotopic_loops A p (shiftpath x p)"
  using homotopic_loops_shiftpath_left[OF assms] by (simp add: homotopic_loops_sym)
```

```
lemma shiftpath_full_part_circlepath:
  "shiftpath c (part_circlepath x r a (a + 2 * of_int n * pi)) =
   part_circlepath x r (a + 2 * n * pi * c) (a + 2 * n * pi * (c + 1))"
  unfolding shiftpath_def
  by (simp add: shiftpath_def part_circlepath_altdef fun_eq_iff rcis_def
linepath_def
                field_simps cis_multiple_2pi' flip: cis_mult cis_divide)

lemma shiftpath_circlepath:
  "shiftpath c (circlepath x r) = part_circlepath x r (c * 2 * pi) ((c
+ 1) * 2 * pi)"
  unfolding circlepath_def using shiftpath_full_part_circlepath[of c x
r 0 1]
  by (simp add: algebra_simps)
```

The following variant of *shiftpath* is more convenient for loops.

```
definition shiftpath' :: "real ⇒ (real ⇒ 'a) ⇒ (real ⇒ 'a)"
  where "shiftpath' a p = (λx. p (frac (x + a)))"

lemma shiftpath'_0 [simp]: "pathfinish p = pathstart p ⟹ t ∈ {0..1}
⟹ shiftpath' 0 p t = p t"
  using frac_eq[of t] by (cases "t = 1") (auto simp: pathfinish_def pathstart_def
shiftpath'_def)

lemma path_image_shiftpath':
  assumes "path p" "pathstart p = pathfinish p"
  shows   "path_image (shiftpath' c p) = path_image p"
proof -
  have "path_image (shiftpath' c p) = (λx. p (frac (x + c))) ` {0..1}"
    unfolding path_image_def shiftpath'_def ..
  also have "{0..1} = {0..<1 - frac c} ∪ {1 - frac c..1}"
    using frac_lt_1[of c] frac_ge_0[of c] by (auto simp del: frac_ge_0)
  also have "(λx. p (frac (x + c))) ` ... =
             (λx. p (frac (x + c))) ` {0..<1 - frac c} ∪ (λx. p (frac
(x + c))) ` {1 - frac c..1}"
    by (rule image_Un)

  also have "(λx. p (frac (x + c))) ` {0..<1 - frac c} = (λx. p (x + frac
c)) ` {0..<1 - frac c}"
  proof (intro image_cong refl)
    show "p (frac (x + c)) = p (x + frac c)" if "x ∈ {0..<1-frac c}"
for x
    proof -
      have "frac x = x"
        using frac_eq[of x] that frac_ge_0[of c] by (auto simp del: frac_ge_0)
      thus ?thesis
        using frac_lt_1[of c] frac_ge_0[of c] that
        by (auto simp: frac_add field_simps simp del: frac_ge_0)
```

```
    qed
  qed
  also have "{0..<1 - frac c} = (+) (-frac c) ‘ {frac c..<1}"
    by (subst image_add_atLeastLessThan) simp_all
  also have "(λx. p (x + frac c)) ‘ ... = p ‘ {frac c..<1}"
    by (subst image_image) simp

  also have "(λx. p (frac (x + c))) ‘ {1 - frac c..1} = (λx. p (x + frac
c - 1)) ‘ {1 - frac c..1}"
  proof (intro image_cong refl)
    fix x assume x: "x ∈ {1 - frac c..1}"
    have "frac (x + c) = x + frac c - 1"
    proof (cases "x = 1")
      case False
      with x have "x ∈ {0..<1}"
        using frac_lt_1[of c] by auto
      hence "frac x = x"
        by (subst frac_eq) auto
      thus ?thesis using x by (auto simp: algebra_simps frac_add)
    qed (auto simp: frac_def)
    thus "p (frac (x + c)) = p (x + frac c - 1)"
      by simp
  qed
  also have "{1 - frac c..1} = (+) (1 - frac c) ‘ {0..frac c}"
    by (subst image_add_atLeastAtMost) simp_all
  also have "(λx. p (x + frac c - 1)) ‘ ... = p ‘ {0..frac c}"
    by (subst image_image) simp

  also have "p ‘ {frac c..<1} ∪ p ‘ {0..frac c} = p ‘ ({frac c..<1} ∪
{0..frac c})"
    by (rule image_Un [symmetric])
  also have "{frac c..<1} ∪ {0..frac c} = {0..<1}"
    using frac_lt_1[of c] frac_ge_0[of c] by (auto simp del: frac_ge_0)
  also have "p ‘ {0..<1} = path_image p"
    by (rule path_image_loop [symmetric]) fact+
  finally show ?thesis .
qed

lemma path_shiftpath_0_iff [simp]: "path (shiftpath 0 p) ⟷ path p"
  unfolding path_def by (intro continuous_on_cong) (auto simp: shiftpath_def)

lemma path_shiftpath'_int_iff [simp]:
  assumes "pathstart p = pathfinish p" "c ∈ ℤ"
  shows   "path (shiftpath' c p) ⟷ path p"
  unfolding path_def
proof (intro continuous_on_cong)
  show "shiftpath' c p x = p x" if "x ∈ {0..1}" for x
  proof (cases "x = 1")
    case False
```

```
    hence "x ∈ {0..<1}"
      using that by auto
    moreover from this have "frac x = x"
      by (subst frac_eq) auto
    moreover have "frac (x + c) = frac x"
      using assms by (auto elim!: Ints_cases simp: frac_def)
    ultimately show ?thesis
      using assms that
      by (auto simp: shiftpath'_def pathstart_def pathfinish_def)
  qed (use assms in ‹auto simp: shiftpath'_def frac_def pathfinish_def
pathstart_def›)
qed auto

lemma shiftpath'_eq_shiftpath:
  assumes "pathstart p = pathfinish p" "c ∈ {0..1}" "t ∈ {0..1}"
  shows   "shiftpath' c p t = shiftpath c p t"
proof -
  consider "t + c < 1" | "t + c = 1" | "t + c > 1" "t + c < 2" | "t +
c ≥ 2"
    by linarith
  thus ?thesis
  proof cases
    case 1
    hence "frac (t + c) = t + c"
      using assms by (subst frac_unique_iff) auto
    thus ?thesis
      using assms 1 by (simp add: shiftpath'_def shiftpath_def add_ac)
  next
    case 3
    hence "frac (t + c) = t + c - 1"
      using assms by (subst frac_unique_iff) (auto simp: algebra_simps)
    thus ?thesis
      using assms 3 by (simp add: shiftpath'_def shiftpath_def add_ac)
  next
    case 4
    with assms have "t + c = 2"
      by auto
    thus ?thesis using assms
      by (simp add: shiftpath'_def shiftpath_def pathstart_def pathfinish_def
add_ac)
  qed (use assms in ‹auto simp: shiftpath'_def shiftpath_def add_ac pathstart_def
pathfinish_def›)
qed

lemma shiftpath'_frac: "shiftpath' (frac c) p = shiftpath' c p"
  unfolding shiftpath'_def by (simp add: frac_def algebra_simps)

lemma path_shiftpath' [intro]:
  "pathstart p = pathfinish p ⟹ path p ⟹ path (shiftpath' c p)"
```

```
  unfolding shiftpath'_def path_def
  by (rule continuous_on_frac_real')
     (auto intro!: continuous_intros simp: pathfinish_def pathstart_def)

lemma pathfinish_shiftpath':
  "pathfinish (shiftpath' c p) = pathstart (shiftpath' c p)"
  by (simp add: pathstart_def pathfinish_def shiftpath'_def frac_def)

lemma shiftpath'_shiftpath': "shiftpath' c (shiftpath' d p) = shiftpath'
(c + d) p"
proof
  fix x :: real
  have "shiftpath' c (shiftpath' d p) x = p (frac (frac (x + c) + d))"
    by (simp_all add: shiftpath'_def)
  also have "frac (frac (x + c) + d) =
              x + c - real_of_int ⌊x + c⌋ + d - real_of_int ⌊x + c -
real_of_int ⌊x + c⌋ + d⌋"
    by (simp add: frac_def)
  also have "x + c - real_of_int ⌊x + c⌋ + d = x + c + d - real_of_int
⌊x + c⌋"
    by Groebner_Basis.algebra
  also have "floor ... = ⌊x + c + d⌋ - ⌊x + c⌋"
    by (subst floor_diff_of_int) auto
  also have "x + c + d - real_of_int ⌊x + c⌋ - real_of_int (⌊x + c + d⌋
- ⌊x + c⌋) =
              frac (x + c + d)"
    by (simp add: frac_def)
  also have "p ... = shiftpath' (c + d) p x"
    by (simp add: shiftpath'_def add_ac)
  finally show "shiftpath' c (shiftpath' d p) x = shiftpath' (c + d) p
x" .
qed

lemma simple_path_shiftpath':
  assumes "simple_path p" "pathfinish p = pathstart p"
  shows   "simple_path (shiftpath' c p)"
proof -
  have "simple_path (shiftpath (frac c) p)"
    by (intro simple_path_shiftpath frac_ge_0 less_imp_le[OF frac_lt_1]
assms)
  also have "?this ⟷ simple_path (shiftpath' (frac c) p)"
    by (intro simple_path_cong) (auto simp: assms shiftpath'_eq_shiftpath
less_imp_le[OF frac_lt_1])
  also have "shiftpath' (frac c) p = shiftpath' c p"
    by (simp only: shiftpath'_frac)
  finally show ?thesis .
qed

lemma simple_path_shiftpath'_iff [simp]:
```

```
  assumes "pathfinish p = pathstart p"
  shows   "simple_path (shiftpath' c p) ⟷ simple_path p"
proof
  assume "simple_path (shiftpath' c p)"
  hence "simple_path (shiftpath' (-c) (shiftpath' c p))"
    by (rule simple_path_shiftpath') (use assms in ‹auto simp: pathfinish_shiftpath'›)
  also have "shiftpath' (-c) (shiftpath' c p) = shiftpath' 0 p"
    by (simp add: shiftpath'_shiftpath')
  also have "simple_path ... ⟷ simple_path p"
    by (intro simple_path_cong) (use assms in auto)
  finally show "simple_path p" .
qed (use assms in ‹auto intro!: simple_path_shiftpath'›)

lemma homotopic_loops_shiftpath'_left:
  assumes "path p" "path_image p ⊆ A" "pathstart p = pathfinish p"
  shows   "homotopic_loops A (shiftpath' x p) p"
proof (rule homotopic_loops_sym, rule homotopic_loops_reparametrize)
  show "continuous_on {0..1} ((+) x)"
    by (intro continuous_intros)
  show "shiftpath' x p t = p (frac (x + t))" if "t ∈ {0..1}" for t
    using that assms by (simp add: shiftpath'_def add_ac)
qed (use assms in auto)

lemma homotopic_loops_shiftpath'_right:
  assumes "path p" "path_image p ⊆ A" "pathstart p = pathfinish p"
  shows   "homotopic_loops A p (shiftpath' x p)"
  using homotopic_loops_shiftpath'_left[OF assms] by (simp add: homotopic_loops_sym)


lemma shiftpath'_full_part_circlepath:
  "shiftpath' c (part_circlepath x r a (a + 2 * of_int n * pi)) =
   part_circlepath x r (a + 2 * n * pi * c) (a + 2 * n * pi * (c + 1))"
(is "?lhs = ?rhs")
proof
  fix t
  have "shiftpath' c (part_circlepath x r a (a + 2 * of_int n * pi)) t
=
          x + rcis r a * cis (2 * pi * (c + t) * of_int n) *
          cis ((2 * pi) * (-of_int (n * ⌊c + t⌋)))"
    by (simp add: shiftpath'_def fun_eq_iff part_circlepath_altdef rcis_def
            linepath_def algebra_simps frac_def divide_conv_cnj cis_cnj

            del: cis_multiple_2pi flip: cis_mult cis_divide)
  also have "cis ((2 * pi) * (-of_int (n * ⌊c + t⌋))) = 1"
    by (rule cis_multiple_2pi) auto
  also have "x + rcis r a * cis (2 * pi * (c + t) * of_int n) * 1 =
             part_circlepath x r (a + 2 * n * pi * c) (a + 2 * n * pi
* (c + 1)) t"
    by (simp add: part_circlepath_def algebra_simps cis_conv_exp exp_add
```

```
linepath_def rcis_def)
  finally show "?lhs t = ?rhs t"
    by simp
qed

lemma shiftpath'_circlepath:
  "shiftpath' c (circlepath x r) = part_circlepath x r (c * 2 * pi) ((c
+ 1) * 2 * pi)"
  unfolding circlepath_def using shiftpath'_full_part_circlepath[of c
x r 0 1]
  by (simp add: algebra_simps)

end
```

2 Some useful relations on paths

```
theory Path_Equivalence
  imports "HOL-Complex_Analysis.Complex_Analysis" Path_Automation_Library
begin
```

2.1 Equivalence of paths up to reparametrisation

We call two paths $p, q : [0,1] \to U$ *equivalent* if p can be transformed to q by composition with an orientation-preserving homoeomorphism f – that is, there exists a continuous and strictly monotonic function f such that $q = p \circ f$. This relation is an equivalence relation.

This is a fairly standard definition in the literature[2], but it does have one downside: it does not fully capture the intuitive notion of path equivalence if the paths *stop* at some point, i.e. if $p([a,b]) = \text{const}$ for $0 \leq a < b \leq b$. Intuitively, such "constant paths" can be added or removed without changing anything. However, with respect to our notion of path equivalence, the path `linepath x x +++ p` is not equivalent to `p` in general, since the reparametrisation function we would need would be something like `λt. if t = 0 then 0 else (t + 1) / 2`, which is not continuous. This also means that the subpath relation is not antisymmetric w.r.t. path equivalence.

One possible way to fix this might be to relax strict monotonicity to non-strict monotonicity, and the continuity to something like "for every $t \in [0,1]$, q is constant on the interval $[f(t^-), f(t^+)]$", where $f(t^-)$ and $f(t^+)$ denote the left and right limit of $f(x)$ as $x \to t$, repectively.

Another way of fixing it might be the definition of Raussen and Fahrenberg [1], which defines p and q to be equivalent if $p \circ \varphi = q \circ \psi$ for continuous, (weakly) monotonic functions $\varphi, \psi : [0,1]$ with $\varphi(0) = \psi(0) = 0$ and $\varphi(1) = \psi(1) = 1$.

In any case, there is one good reason *not* to allow such equivalences, namely that they do not preserve properties such as a path being simple (i.e. not

self-intersecting) – at least in the sense that it is defined in the Isabelle/HOL library. Namely, for a path to be simple, we require it to be injective on $[0,1]$ with the possible exception that $p(0) = p(1)$ is allowed. Clearly, this is *not* preserved by appending or deleting "constant paths".

Thus, if one wanted to generalise our notion of path equivalence this way, one would ideally also generalise the notions of `arc` and `simple_path` accordingly, which will probably be a substantial bit of work. It is questionable whether this would be worth the effort.

```
locale eq_paths_locale =
  fixes p q :: "real ⇒ 'a :: topological_space" and f :: "real ⇒ real"
  assumes paths [simp, intro]: "path p" "path q"
  assumes cont [continuous_intros]: "continuous_on {0..1} f"
  assumes mono: "strict_mono_on {0..1} f"
  assumes ends [simp]: "f 0 = 0" "f 1 = 1"
  assumes equiv: "⋀t. t ∈ {0..1} ⟹ q t = p (f t)"
begin

lemmas cont' [continuous_intros] = continuous_on_compose2 [OF cont]

lemma inj: "inj_on f {0..1}"
  using strict_mono_on_imp_inj_on mono by blast

lemma inj': "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x = f y ⟷ x = y"
  using inj by (meson inj_on_contraD)

lemma less_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x < f y ⟷ x < y"
  using mono by (meson less_le_not_le linorder_linear strict_mono_onD
strict_mono_on_leD)

lemma le_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x ≤ f y ⟷ x ≤ y"
  using mono less_iff linorder_not_le by blast

lemma eq_0_iff [simp]: "x ∈ {0..1} ⟹ f x = 0 ⟷ x = 0"
  and eq_1_iff [simp]: "x ∈ {0..1} ⟹ f x = 1 ⟷ x = 1"
  using inj'[of x 0] inj'[of x 1] by simp_all

lemma f_ge_0 [simp, intro]: "x ∈ {0..1} ⟹ f x ≥ 0"
  and f_le_1 [simp, intro]: "x ∈ {0..1} ⟹ f x ≤ 1"
  using le_iff[of 0 x] le_iff[of x 1] ends by simp_all

lemma f_gt_0 [simp, intro]: "x ∈ {0<..1} ⟹ f x > 0"
  and f_less_1 [simp, intro]: "x ∈ {0..<1} ⟹ f x < 1"
  using less_iff[of 0 x] less_iff[of x 1] ends by simp_all

lemma le_0_iff [simp]: "x ∈ {0..1} ⟹ f x ≤ 0 ⟷ x = 0"
  and ge_1_iff [simp]: "x ∈ {0..1} ⟹ f x ≥ 1 ⟷ x = 1"
  using le_iff[of x 0] le_iff[of 1 x] le_iff[of 0 x] le_iff[of x 1] ends
by force+
```

```
lemma bij_betw: "bij_betw f {0..1} {0..1}"
proof -
  have "x ∈ f ‘ {0..1}" if "x ∈ {0..1}" for x
    using IVT'[of f 0 x 1] that cont by auto
  thus ?thesis
    using inj unfolding bij_betw_def by force
qed

lemma same_ends: "pathstart p = pathstart q" "pathfinish p = pathfinish
q"
  by (simp_all add: pathstart_def pathfinish_def equiv)

lemma path_image_eq: "path_image p = path_image q"
proof -
  have "path_image q = q ‘ {0..1}"
    by (simp add: path_image_def)
  also have "... = (p ∘ f) ‘ {0..1}"
    by (intro image_cong) (auto simp: equiv)
  also have "... = p ‘ (f ‘ {0..1})"
    by (simp add: image_image)
  also have "f ‘ {0..1} = {0..1}"
    using bij_betw by (meson bij_betw_def)
  also have "p ‘ ... = path_image p"
    by (simp add: path_image_def)
  finally show ?thesis ..
qed

lemma inverse: "eq_paths_locale q p (inv_into {0..1} f)"
proof
  let ?g = "inv_into {0..1} f"
  show "continuous_on {0..1} (?g)"
    using strict_mono_on_imp_continuous_on_inv_into[OF mono] bij_betw
    by (simp add: bij_betw_def)
  show *: "strict_mono_on {0..1} ?g"
    using strict_mono_on_imp_strict_mono_on_inv_into[OF mono] bij_betw
    by (simp add: bij_betw_def)
  show [simp]: "?g 0 = 0"
    using inv_into_f_f[OF inj, of 0] by simp
  show [simp]: "?g 1 = 1"
    using inv_into_f_f[OF inj, of 1] by simp
  show "p t = q (?g t)" if t: "t ∈ {0..1}" for t
  proof -
    have "?g 0 ≤ ?g t" "?g t ≤ ?g 1"
      by (rule strict_mono_on_leD[OF *]; use t in simp)+
    hence "q (?g t) = p (f (?g t))"
      by (simp add: equiv)
    also have "f (?g t) = t"
      by (rule bij_betw_inv_into_right[OF bij_betw]) (use t in auto)
```

```
    finally show ?thesis ..
  qed
qed auto

lemma reverse: "eq_paths_locale (reversepath p) (reversepath q) (λx.
1 - f (1 - x))"
proof
  show "reversepath q t = reversepath p (1 - f (1 - t))" if "t ∈ {0..1}"
for t
    using that by (auto simp: reversepath_def equiv)
qed (auto intro!: continuous_intros strict_mono_onI simp: less_iff)

lemma homotopic:
  assumes "path_image p ⊆ A"
  shows   "homotopic_paths A p q"
  by (rule homotopic_paths_reparametrize[where f = f])
     (use assms in <auto intro!: continuous_intros simp: equiv>)

lemma arc_iff: "arc p ⟷ arc q"
proof -
  have "arc q ⟷ inj_on q {0..1}"
    unfolding arc_def by simp
  also have "... ⟷ inj_on (p ∘ f) {0..1}"
    by (intro inj_on_cong) (auto simp: equiv)
  also have "... ⟷ inj_on p (f ‘ {0..1})"
    by (rule comp_inj_on_iff [OF inj, symmetric])
  also have "... ⟷ arc p"
    using bij_betw by (simp add: bij_betw_def arc_def)
  finally show ?thesis ..
qed

lemma simple_path:
  assumes "simple_path p"
  shows   "simple_path q"
proof (rule simple_pathI)
  show "x = 0 ∧ y = 1" if "0 ≤ x" "x < y" "y ≤ 1" "q x = q y" for x
y
  proof -
    have "p (f x) = p (f y)"
      using that by (simp add: equiv)
    moreover from that have "f x < f y"
      by (subst less_iff) auto
    ultimately have "{f x, f y} = {0,1}"
      using simple_pathD[OF assms, of "f x" "f y"] that by simp
    thus ?thesis using that
      by (auto simp: doubleton_eq_iff)
  qed
qed auto
```

```
lemma simple_path_iff: "simple_path p ⟷ simple_path q"
proof -
  interpret inv: eq_paths_locale q p "inv_into {0..1} f"
    by (rule inverse)
  show ?thesis
    using simple_path inv.simple_path by blast
qed

end


locale eq_paths_locale_compose =
  pq: eq_paths_locale p q f + qr : eq_paths_locale q r g for p q r f g
begin

sublocale eq_paths_locale p r "f ∘ g"
proof
  show "strict_mono_on {0..1} (f ∘ g)"
    using pq.mono qr.mono
    by (rule strict_mono_on_compose) (use qr.bij_betw in ‹simp add: bij_betw_def›)
qed (auto intro!: continuous_intros simp: pq.equiv qr.equiv)

end


lemma eq_paths_locale_refl [intro!]: "path p ⟹ eq_paths_locale p p
(λx. x)"
  by unfold_locales (auto intro!: strict_mono_onI)

lemma eq_paths_locale_refl':
  assumes "path p ∨ path q" "⋀x. x ∈ {0..1} ⟹ p x = q x"
  shows   "eq_paths_locale p q (λx. x)"
proof
  have "path p ⟷ path q"
    unfolding path_def by (intro continuous_on_cong) (use assms(2) in
auto)
  with assms show "path p" "path q"
    by auto
qed (use assms(2) in ‹auto intro!: strict_mono_onI›)


locale eq_paths_locale_join =
  p1: eq_paths_locale p1 q1 f1 + p2 : eq_paths_locale p2 q2 f2 for p1
q1 f1 p2 q2 f2 +
  assumes compatible_ends: "pathfinish p1 = pathstart p2"
begin

definition f12 :: "real ⇒ real" where
  "f12 t = (if t ≤ 1 / 2 then f1 (2 * t) / 2 else (f2 (2 * t - 1) + 1)
```

```
/ 2)"

lemma compatible_ends': "pathfinish q1 = pathstart q2"
  using p1.same_ends p2.same_ends compatible_ends by metis

sublocale p12: eq_paths_locale "p1 +++ p2" "q1 +++ q2" f12
proof
  show "strict_mono_on {0..1} f12"
  proof (rule strict_mono_onI)
    fix r s :: real assume rs: "r ∈ {0..1}" "s ∈ {0..1}" "r < s"
    consider "s ≤ 1 / 2" | "r ≤ 1 / 2" "s > 1 / 2" | "r > 1 / 2"
      using ‹r < s› by linarith
    thus "f12 r < f12 s"
    proof cases
      assume rs': "r ≤ 1 / 2" "s > 1 / 2"
      have "f12 r = f1 (2 * r) / 2"
        using rs rs' by (simp add: f12_def)
      also have "... ≤ 1 / 2"
        using rs rs' by simp
      also have "... < (f2 (2 * s - 1) + 1) / 2"
        using rs rs' by simp
      also have "... = f12 s"
        using rs rs' by (simp add: f12_def)
      finally show ?thesis .
    qed (use p1.mono p2.mono rs in ‹auto simp: strict_mono_on_def f12_def›)
  qed
next
  show "continuous_on {0..1} f12"
    unfolding f12_def by (intro continuous_on_real_If_combine continuous_intros)
auto
next
  show "(q1 +++ q2) t = (p1 +++ p2) (f12 t)" if t: "t ∈ {0..1}" for t
  proof (cases t "1 / 2 :: real" rule: linorder_cases)
    case less
    have "(p1 +++ p2) (f12 t) = q1 (2 * t)"
      using less t by (simp add: joinpaths_def f12_def p1.equiv)
    also have "... = (q1 +++ q2) t"
      using less t by (simp add: joinpaths_def)
    finally show ?thesis ..
  next
    case greater
    hence "f2 (2 * t - 1) > 0"
      using t by simp
    hence "(p1 +++ p2) (f12 t) = p2 ((2 * f2 (2 * t - 1) + 2) / 2 - 1)"
      using greater by (simp add: joinpaths_def f12_def)
    also have "(2 * f2 (2 * t - 1) + 2) / 2 - 1 = f2 (2 * t - 1)"
      by (simp add: field_simps)
    also have "p2 (f2 (2 * t - 1)) = q2 (2 * t - 1)"
      using t greater by (simp add: p2.equiv)
```

```
    also have "... = (q1 +++ q2) t"
      using greater by (simp add: joinpaths_def)
    finally show ?thesis ..
  qed (auto simp: joinpaths_def f12_def p1.equiv p2.equiv)
qed (auto simp: compatible_ends compatible_ends' f12_def)

end


locale eq_paths_locale_join_assoc =
  fixes p1 p2 p3 :: "real ⇒ 'a :: topological_space"
  assumes paths [simp, intro]: "path p1" "path p2" "path p3"
  assumes compatible_ends: "pathfinish p1 = pathstart p2" "pathfinish
p2 = pathstart p3"
begin

definition f :: "real ⇒ real" where
  "f t = (if t ≤ 1 / 2 then t / 2
          else if t ≤ 3 / 4 then t - 1 / 4
          else 2 * t - 1)"

sublocale eq_paths_locale "(p1 +++ p2) +++ p3" "p1 +++ (p2 +++ p3)" f
proof
  show "(p1 +++ (p2 +++ p3)) t = ((p1 +++ p2) +++ p3) (f t)" if t: "t
∈ {0..1}" for t
    by (auto simp: joinpaths_def pathfinish_def pathstart_def f_def)
  show "strict_mono_on {0..1} f"
    by (intro strict_mono_onI) (auto simp: f_def)
  show "continuous_on {0..1} f"
    unfolding f_def by (intro continuous_on_real_If_combine continuous_intros)
auto
qed (auto simp: f_def compatible_ends)

end
```

We now introduce the actual equivalence relation, where the reparametrisation function is hidden behind an existential quantifier.

```
definition eq_paths :: "(real ⇒ 'a :: topological_space) ⇒ (real ⇒ 'a)
⇒ bool" where
  "eq_paths p q ⟷ (∃f. eq_paths_locale p q f)"

named_theorems eq_paths_intros

lemma eq_paths_imp_path [dest]:
  assumes "eq_paths p q"
  shows   "path p" "path q"
  using assms unfolding eq_paths_def eq_paths_locale_def by blast+

lemma eq_paths_refl [simp, intro!, eq_paths_intros]: "path p ⟹ eq_paths
```

```
p p"
  unfolding eq_paths_def by blast

lemma eq_paths_refl'': "path p ⟹ p = q ⟹ eq_paths p q"
  unfolding eq_paths_def by blast

lemma eq_paths_refl':
  "path p ∨ path q ⟹ (⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ eq_paths p
q"
  unfolding eq_paths_def using eq_paths_locale_refl'[of p q] by blast

lemma eq_paths_sym:
  "eq_paths p q ⟹ eq_paths q p"
  unfolding eq_paths_def using eq_paths_locale.inverse by auto

lemma eq_paths_sym_iff:
  "eq_paths p q ⟷ eq_paths q p"
  using eq_paths_sym by metis

lemma eq_paths_reverse [intro, eq_paths_intros]:
  "eq_paths p q ⟹ eq_paths (reversepath p) (reversepath q)"
  unfolding eq_paths_def using eq_paths_locale.reverse by auto

lemma eq_paths_reverse_iff:
  "eq_paths (reversepath p) (reversepath q) ⟷ eq_paths p q"
  using eq_paths_reverse reversepath_reversepath by metis

lemma eq_paths_trans [trans]:
  assumes "eq_paths p q" "eq_paths q r"
  shows   "eq_paths p r"
proof -
  from assms(1) obtain f where "eq_paths_locale p q f"
    by (auto simp: eq_paths_def)
  then interpret pq: eq_paths_locale p q f .
  from assms(2) obtain g where "eq_paths_locale q r g"
    by (auto simp: eq_paths_def)
  then interpret qr: eq_paths_locale q r g .
  interpret eq_paths_locale_compose p q r f g ..
  show ?thesis
    unfolding eq_paths_def using eq_paths_locale_axioms by blast
qed

lemma eq_paths_eq_trans [trans]:
  "p = q ⟹ eq_paths q r ⟹ eq_paths p r"
  "eq_paths p q ⟹ q = r ⟹ eq_paths p r"
  by simp_all

lemma eq_paths_shiftpath_0 [intro, eq_paths_intros]: "path p ⟹ eq_paths
(shiftpath 0 p) p"
```

```
  by (rule eq_paths_refl') (auto simp: shiftpath_def)

lemma eq_paths_shiftpath_0_iff [simp]: "eq_paths (shiftpath 0 p) q ⟷
eq_paths p q"
proof safe
  assume *: "eq_paths (shiftpath 0 p) q"
  hence "path p"
    by auto
  thus "eq_paths p q" using *
    by (meson eq_paths_shiftpath_0 eq_paths_sym_iff eq_paths_trans)
next
  assume "eq_paths p q"
  thus "eq_paths (shiftpath 0 p) q"
    by (meson eq_paths_imp_path(1) eq_paths_shiftpath_0 eq_paths_trans)
qed

lemma eq_paths_shiftpath_0_iff' [simp]: "eq_paths q (shiftpath 0 p) ⟷
eq_paths q p"
  using eq_paths_shiftpath_0_iff[of p q] by (simp add: eq_paths_sym_iff)

lemma eq_paths_shiftpath'_int [eq_paths_intros]:
  assumes "path p" "c ∈ ℤ" "pathstart p = pathfinish p"
  shows   "eq_paths (shiftpath' c p) p"
proof (rule eq_paths_refl')
  show "shiftpath' c p x = p x" if "x ∈ {0..1}" for x
  proof (cases "x = 1")
    case False
    with that have "x ∈ {0..<1}"
      by auto
    moreover from this have "frac x = x"
      by (auto simp: frac_eq)
    ultimately show ?thesis using assms
      by (auto simp: shiftpath'_def pathstart_def pathfinish_def frac_def
elim!: Ints_cases)
  qed (use assms in ‹auto simp: shiftpath'_def frac_def pathstart_def
pathfinish_def›)
qed (use assms in auto)

lemma eq_paths_shiftpath'_int_iff [simp]:
  assumes "pathstart p = pathfinish p" "c ∈ ℤ"
  shows   "eq_paths (shiftpath' c p) q ⟷ eq_paths p q"
proof safe
  assume *: "eq_paths (shiftpath' c p) q"
  hence "path p"
    using assms by auto
  thus "eq_paths p q" using * assms
    by (meson eq_paths_shiftpath'_int eq_paths_sym_iff eq_paths_trans)
next
  assume "eq_paths p q"
```

```
  thus "eq_paths (shiftpath' c p) q"
    by (meson assms eq_paths_imp_path(1) eq_paths_shiftpath'_int eq_paths_trans)
qed

lemma eq_paths_shiftpath'_int_iff' [simp]:
  assumes "pathstart p = pathfinish p" "c ∈ ℤ"
  shows "eq_paths q (shiftpath' c p) ⟷ eq_paths q p"
  using eq_paths_shiftpath'_int_iff[of p c q] assms by (simp add: eq_paths_sym_iff)

lemma eq_paths_join [eq_paths_intros]:
  assumes "eq_paths p1 q1" "eq_paths p2 q2"
  assumes *: "{pathfinish p1, pathfinish q1} ∩ {pathstart p2, pathstart
q2} ≠ {}"
  shows  "eq_paths (p1 +++ p2) (q1 +++ q2)"
proof -
  from assms(1) obtain f where "eq_paths_locale p1 q1 f"
    by (auto simp: eq_paths_def)
  then interpret p1: eq_paths_locale p1 q1 f .
  from assms(2) obtain g where "eq_paths_locale p2 q2 g"
    by (auto simp: eq_paths_def)
  then interpret p2: eq_paths_locale p2 q2 g .
  interpret eq_paths_locale_join p1 q1 f p2 q2 g
    by unfold_locales (use * in ‹auto simp: p1.same_ends p2.same_ends›)
  show ?thesis
    unfolding eq_paths_def using p12.eq_paths_locale_axioms by blast
qed

lemma eq_paths_join_assoc1 [eq_paths_intros]:
  assumes "path p1" "path p2" "path p3"
  assumes "pathfinish p1 = pathstart p2" "pathfinish p2 = pathstart p3"
  shows   "eq_paths ((p1 +++ p2) +++ p3) (p1 +++ (p2 +++ p3))"
proof -
  interpret eq_paths_locale_join_assoc p1 p2 p3
    by standard (use assms in auto)
  show ?thesis
    unfolding eq_paths_def using eq_paths_locale_axioms by blast
qed

lemma eq_paths_join_assoc2 [eq_paths_intros]:
  assumes "path p1" "path p2" "path p3"
  assumes "pathfinish p1 = pathstart p2" "pathfinish p2 = pathstart p3"
  shows   "eq_paths (p1 +++ (p2 +++ p3)) ((p1 +++ p2) +++ p3)"
  using eq_paths_join_assoc1[OF assms] by (simp add: eq_paths_sym_iff)

lemma eq_paths_imp_same_ends:
  "eq_paths p q ⟹ pathstart p = pathstart q"
  "eq_paths p q ⟹ pathfinish p = pathfinish q"
  unfolding eq_paths_def using eq_paths_locale.same_ends by blast+
```

```
lemma eq_paths_imp_path_image_eq:
  "eq_paths p q ⟹ path_image p = path_image q"
  unfolding eq_paths_def using eq_paths_locale.path_image_eq by blast

lemma eq_paths_imp_homotopic:
  assumes "eq_paths p q" "path_image p ∩ path_image q ⊆ A"
  shows   "homotopic_paths A p q"
proof -
  from assms obtain f where "eq_paths_locale p q f"
    by (auto simp: eq_paths_def)
  then interpret eq_paths_locale p q f .
  show ?thesis
    using homotopic[of A] path_image_eq assms(2) by blast
qed

lemma eq_paths_homotopic_paths_trans [trans]:
  "eq_paths p q ⟹ homotopic_paths A q r ⟹ homotopic_paths A p r"
  "homotopic_paths A p q ⟹ eq_paths q r ⟹ homotopic_paths A p r"
proof -
  show "eq_paths p q ⟹ homotopic_paths A q r ⟹ homotopic_paths A
p r"
    using eq_paths_imp_homotopic
    by (metis homotopic_paths_imp_subset homotopic_paths_trans le_infI2)
  show "homotopic_paths A p q ⟹ eq_paths q r ⟹ homotopic_paths A
p r"
    using eq_paths_imp_homotopic
    by (metis eq_paths_imp_path_image_eq homotopic_paths_imp_subset homotopic_paths_trans
inf_idem)
qed

lemma eq_paths_imp_winding_number_eq:
  assumes "eq_paths p q" "x ∉ path_image p ∩ path_image q"
  shows   "winding_number p x = winding_number q x"
  using assms by (intro winding_number_homotopic_paths eq_paths_imp_homotopic)
auto

lemma eq_paths_imp_contour_integral_eq:
  assumes "eq_paths p q" "valid_path p" "valid_path q"
  assumes "f analytic_on (path_image p ∩ path_image q)"
  shows   "contour_integral p f = contour_integral q f"
proof -
  from assms(4) obtain A where A: "open A" "f holomorphic_on A" "path_image
p ∩ path_image q ⊆ A"
    using analytic_on_holomorphic by auto
  show ?thesis
  proof (rule Cauchy_theorem_homotopic_paths)
    show "homotopic_paths A p q"
      by (intro eq_paths_imp_homotopic assms A)
  qed (use assms A in auto)
```

qed

lemma *eq_paths_imp_arc_iff:*
"eq_paths p q ⟹ arc p ⟷ arc q"
unfolding *eq_paths_def* **using** *eq_paths_locale.arc_iff* **by** *blast*

lemma *eq_paths_arc_trans [trans]:*
"eq_paths p q ⟹ arc q ⟹ arc p"
"arc p ⟹ eq_paths p q ⟹ arc q"
using *eq_paths_imp_arc_iff* **by** *metis+*

lemma *eq_paths_imp_simple_path_iff:*
"eq_paths p q ⟹ simple_path p ⟷ simple_path q"
unfolding *eq_paths_def* **using** *eq_paths_locale.simple_path_iff* **by** *blast*

lemma *eq_paths_simple_path_trans [trans]:*
"eq_paths p q ⟹ simple_path q ⟹ simple_path p"
"simple_path p ⟹ eq_paths p q ⟹ simple_path q"
using *eq_paths_imp_simple_path_iff* **by** *metis+*

2.2 Splitting lines and circular arcs

If we have a line or a circular arc, we can split that path into two subpaths of the same "type" such that the concatenation of the two subpaths is equivalent to the full path.

locale *linepaths_join =*
fixes *a b c :: "'a :: euclidean_space"*
assumes *between: "b ∈ closed_segment a c"*
begin

definition *f :: "real ⇒ real"* **where**
"f t = (let u = (if a = c then 1 / 2 else dist a b / dist a c)
*in if t ≤ 1 / 2 then 2 * u * t else -1 + 2 * t + 2 * u - 2 * t * u)"*

lemma *eq_paths_locale:*
assumes *not_degenerate: "a = c ∨ (a ≠ b ∧ b ≠ c)"*
shows *"eq_paths_locale (linepath a c) (linepath a b +++ linepath b c) f"*
proof
from *between* **obtain** *u* **where** *u: "u ∈ {0..1}" "b = (1 - u) *$_R$ a + u *$_R$ c"*
unfolding *closed_segment_def* **by** *force*

have **: "dist a b / dist a c = u"* **if** *"a ≠ c"*
proof -
have *"a - b = u *$_R$ (a - c)"*
by *(simp add: dist_norm scaleR_conv_of_real u algebra_simps)*
also have *"norm ... = u * norm (a - c)"*

```
    using u by simp
  finally show ?thesis using that
    by (simp add: field_simps dist_norm norm_minus_commute)
 qed

 show "(linepath a b +++ linepath b c) t = linepath a c (f t)" if "t
∈ {0..1}" for t
 proof (cases "a = c")
  case False
  have **: "(u * 2) *R x = u *R x + u *R x" for x :: 'a
   by (simp add: pth_8)
  show ?thesis
   unfolding f_def Let_def *[OF False]
   by (auto simp: u linepath_def joinpaths_def algebra_simps **)
 next
  case [simp]: True
  hence [simp]: "b = c"
   by (simp add: u)
  show ?thesis
   by (simp add: linepath_def joinpaths_def)
 qed
next
 define u where "u = (if a = c then 1/2 else dist a b / dist a c)"
 have "dist a b < dist a c" if "a ≠ b" "b ≠ c" "a ≠ c"
 proof -
  have "dist a c = dist a b + dist b c"
   using between between_mem_segment[of a c b] Line_Segment.between[of
a c b]
   by simp
  with that show ?thesis
   by simp
 qed
 hence u: "u > 0" "u < 1"
  using not_degenerate by (auto simp: u_def field_simps)
 show "continuous_on {0..1} f"
  unfolding f_def u_def [symmetric] Let_def
  by (intro continuous_on_real_If_combine continuous_intros) auto
 show "strict_mono_on {0..1} f"

 proof (rule strict_mono_on_atLeastAtMost_combine[where b = "1/2"])
  show "strict_mono_on {0..1 / 2} f"
  proof (rule strict_mono_onI)
   show "f r < f s" if "r ∈ {0..1/2}" "s ∈ {0..1/2}" "r < s" for r
s
    using that unfolding f_def u_def [symmetric] Let_def using u by
auto
  qed
  show "strict_mono_on {1 / 2..1} f"
  proof (rule strict_mono_onI)
```

```
      show "f r < f s" if "r ∈ {1/2..1}" "s ∈ {1/2..1}" "r < s" for r
s
      proof (cases "r = 1/2")
        case True
        have "0 < (2 * s - 1) * (1 - u)"
          using that u by (intro mult_pos_pos) auto
        also have "(2 * s - 1) * (1 - u) = f s - f (1 / 2)"
          unfolding f_def u_def [symmetric] Let_def using that
          by (auto simp: algebra_simps)
        finally show ?thesis by (simp add: True)
      next
        case False
        have "0 < 2 * (s - r) * (1 - u)"
          by (intro mult_pos_pos) (use that u in auto)
        also have "2 * (s - r) * (1 - u) = f s - f r"
          unfolding f_def u_def [symmetric] Let_def using that False
          by (simp add: algebra_simps)
        finally show ?thesis by simp
      qed
    qed
  qed
qed (auto simp: f_def)

end


locale part_circlepaths_join =
  fixes x :: complex and r a b c :: real
  assumes between: "b ∈ closed_segment a c"
begin

sublocale angle: linepaths_join a b c
  by unfold_locales (fact between)

lemma eq_paths_locale:
  assumes not_degenerate: "a = c ∨ (a ≠ b ∧ b ≠ c)"
  shows "eq_paths_locale (part_circlepath x r a c)
           (part_circlepath x r a b +++ part_circlepath x r b c) angle.f"
proof -
  interpret angle: eq_paths_locale "linepath a c" "linepath a b +++ linepath
b c" angle.f
    by (rule angle.eq_paths_locale) fact
  show ?thesis
  proof
    show "(part_circlepath x r a b +++ part_circlepath x r b c) t =
            part_circlepath x r a c (angle.f t)" if "t ∈ {0..1}" for
t
    proof -
      have "(part_circlepath x r a b +++ part_circlepath x r b c) t =
```

```
            x + rcis r ((linepath a b +++ linepath b c) t)"
        by (simp add: part_circlepath_altdef joinpaths_def)
      also have "(linepath a b +++ linepath b c) t = linepath a c (angle.f
t)"
        using that by (simp add: angle.equiv)
      also have "x + rcis r ... = part_circlepath x r a c (angle.f t)"
        by (simp add: part_circlepath_altdef)
      finally show ?thesis .
    qed
  qed (auto intro: angle.mono continuous_intros)
qed

end

lemma eq_paths_linepaths:
  fixes a b c :: "'a :: euclidean_space"
  assumes "b ∈ closed_segment a c" "a = c ∨ (a ≠ b ∧ b ≠ c)" "b =
b'"
  shows   "eq_paths (linepath a b +++ linepath b' c) (linepath a c)"
           (is "eq_paths ?g ?h")
proof -
  interpret linepaths_join a b c
    by unfold_locales fact
  interpret eq_paths_locale ?h ?g f
    unfolding ‹b = b'›[symmetric]
    by (rule eq_paths_locale) fact
  have "eq_paths ?h ?g"
    unfolding eq_paths_def using eq_paths_locale_axioms by blast
  thus ?thesis
    by (rule eq_paths_sym)
qed

lemmas eq_paths_linepaths' = eq_paths_sym [OF eq_paths_linepaths]

lemma eq_paths_joinpaths_linepath [eq_paths_intros]:
  fixes a b :: "'a :: euclidean_space"
  assumes "eq_paths p (linepath a c)"
  assumes "eq_paths q (linepath c b)"
  assumes "c ∈ closed_segment a b"
  assumes "a = b ∨ (a ≠ c ∧ c ≠ b)"
  shows   "eq_paths (p +++ q) (linepath a b)"
proof -
  have [simp]: "pathfinish p = c" "pathstart q = c"
    using eq_paths_imp_same_ends[OF assms(1)] eq_paths_imp_same_ends[OF
assms(2)]
    by auto
  have "eq_paths (p +++ q) (linepath a c +++ linepath c b)"
    by (intro eq_paths_join assms) (use assms in auto)
  also have "eq_paths ... (linepath a b)"
```

```
    by (intro eq_paths_linepaths) (use assms in auto)
  finally show ?thesis .
qed

lemma eq_paths_joinpaths_linepath' [eq_paths_intros]:
  fixes a b :: "'a :: euclidean_space"
  shows "eq_paths (linepath a c) p ⟹ eq_paths (linepath c b) q ⟹

          c ∈ closed_segment a b ⟹ a = b ∨ a ≠ c ∧ c ≠ b ⟹ eq_paths
(linepath a b) (p +++ q)"
  using eq_paths_joinpaths_linepath[of p a c q b] by (simp add: eq_paths_sym_iff)

lemma eq_paths_part_circlepaths [eq_paths_intros]:
  assumes "b ∈ closed_segment a c" "a = c ∨ (a ≠ b ∧ b ≠ c)" "b =
b'"
  shows   "eq_paths (part_circlepath x r a b +++ part_circlepath x r b'
c)
             (part_circlepath x r a c)" (is "eq_paths ?g ?h")
proof -
  interpret part_circlepaths_join x r a b c
    by unfold_locales fact
  interpret eq_paths_locale ?h ?g angle.f
    unfolding ‹b = b'› [symmetric] by (rule eq_paths_locale) fact
  have "eq_paths ?h ?g"
    unfolding eq_paths_def using eq_paths_locale_axioms by blast
  thus ?thesis
    by (rule eq_paths_sym)
qed

lemmas eq_paths_part_circlepaths' [eq_paths_intros] =
  eq_paths_sym [OF eq_paths_part_circlepaths]

lemma eq_paths_joinpaths_part_circlepath [eq_paths_intros]:
  assumes "eq_paths p (part_circlepath x r a c)"
  assumes "eq_paths q (part_circlepath x r c b)"
  assumes "c ∈ closed_segment a b"
  assumes "a = b ∨ (a ≠ c ∧ c ≠ b)"
  shows   "eq_paths (p +++ q) (part_circlepath x r a b)"
proof -
  have "eq_paths (p +++ q) (part_circlepath x r a c +++ part_circlepath
x r c b)"
    by (intro eq_paths_join assms) (use assms in auto)
  also have "eq_paths ... (part_circlepath x r a b)"
    by (intro eq_paths_part_circlepaths) (use assms in auto)
  finally show ?thesis .
qed

lemma eq_paths_joinpaths_part_circlepath' [eq_paths_intros]:
  assumes "eq_paths (part_circlepath x r a c) p"
```

```
  assumes "eq_paths (part_circlepath x r c b)q "
  assumes "c ∈ closed_segment a b"
  assumes "a = b ∨ (a ≠ c ∧ c ≠ b)"
  shows   "eq_paths (part_circlepath x r a b) (p +++ q)"
  using eq_paths_joinpaths_part_circlepath[of p x r a c q b] assms by
(simp add: eq_paths_sym)
```

2.3 The subpath relation

A path p is called a *subpath* of a path q if it can be "cut" from q with a strictly monotonic reparametrisation function just as for path equivalence before, except that now the reparametrisation function need not start at 0 and need not finish at 1.

This relation is a preorder.

```
locale subpath_locale =
  fixes p q :: "real ⇒ 'a :: topological_space" and f :: "real ⇒ real"
  assumes borders: "f 0 ≥ 0" "f 1 ≤ 1"
  assumes paths [simp, intro]: "path p" "path q"
  assumes cont [continuous_intros]: "continuous_on {0..1} f"
  assumes mono: "strict_mono_on {0..1} f"
  assumes equiv: "⋀t. t ∈ {0..1} ⟹ p t = q (f t)"
begin

lemmas cont' [continuous_intros] = continuous_on_compose2 [OF cont]

lemma inj: "inj_on f {0..1}"
  using strict_mono_on_imp_inj_on mono by blast

lemma inj': "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x = f y ⟷ x = y"
  using inj by (meson inj_on_contraD)

lemma less_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x < f y ⟷ x < y"
  using mono by (meson less_le_not_le linorder_linear strict_mono_onD
strict_mono_on_leD)

lemma le_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x ≤ f y ⟷ x ≤ y"
  using mono less_iff linorder_not_le by blast

lemma eq_f0_iff [simp]: "x ∈ {0..1} ⟹ f x = f 0 ⟷ x = 0"
  and eq_f1_iff [simp]: "x ∈ {0..1} ⟹ f x = f 1 ⟷ x = 1"
  using inj'[of x 0] inj'[of x 1] by simp_all

lemma eq_0_iff: "x ∈ {0..1} ⟹ f x = 0 ⟷ x = 0 ∧ f 0 = 0"
  and eq_1_iff: "x ∈ {0..1} ⟹ f x = 1 ⟷ x = 1 ∧ f 1 = 1"
  using le_iff[of x 0] le_iff[of 1 x] borders by auto

lemma eq_0_iff' [simp]: "NO_MATCH 0 x ⟹ x ∈ {0..1} ⟹ f x = 0 ⟷
x = 0 ∧ f 0 = 0"
```

```
  and eq_1_iff' [simp]: "NO_MATCH 1 x ⟹ x ∈ {0..1} ⟹ f x = 1 ⟷
x = 1 ∧ f 1 = 1"
  by (rule eq_0_iff eq_1_iff; assumption)+

lemma ge_0 [simp]: "x ∈ {0..1} ⟹ f x ≥ 0"
  and le_1 [simp]: "x ∈ {0..1} ⟹ f x ≤ 1"
  using le_iff[of 0 x] le_iff[of x 1] borders by auto

lemma bij_betw: "bij_betw f {0..1} {f 0..f 1}"
proof -
  have "x ∈ f ` {0..1}" if "x ∈ {f 0..f 1}" for x
    using IVT'[of f 0 x 1] that cont by auto
  hence "f ` {0..1} = {f 0..f 1}"
    by (auto simp: le_iff)
  thus ?thesis
    using inj unfolding bij_betw_def by blast
qed

lemma in_range: "f x ∈ {0..1}" if "x ∈ {0..1}"
  using bij_betw that borders unfolding bij_betw_def by auto

lemma path_image_subset: "path_image p ⊆ path_image q"
proof -
  have "path_image p = p ` {0..1}"
    by (simp add: path_image_def)
  also have "... = (q ∘ f) ` {0..1}"
    by (intro image_cong) (auto simp: equiv)
  also have "... = q ` (f ` {0..1})"
    by (simp add: image_image)
  also have "f ` {0..1} = {f 0..f 1}"
    using bij_betw by (meson bij_betw_def)
  also have "q ` ... ⊆ q ` {0..1}"
    using borders by (intro image_mono) auto
  also have " ... = path_image q"
    by (simp add: path_image_def)
  finally show ?thesis .
qed

lemma reverse: "subpath_locale (reversepath p) (reversepath q) (λx. 1
- f (1 - x))"
proof
  show "reversepath p t = reversepath q (1 - f (1 - t))"
    if "t ∈ {0..1}" for t
    using that by (auto simp: reversepath_def equiv)
qed (auto intro!: continuous_intros strict_mono_onI simp: less_iff borders)

lemma arc:
  assumes "arc q"
  shows   "arc p"
```

```
  unfolding arc_def
proof (safe intro!: inj_onI)
  fix x y
  assume xy: "x ∈ {0..1}" "y ∈ {0..1}" "p x = p y"
  hence "q (f x) = q (f y)"
    by (simp add: equiv)
  hence "f x = f y"
    by (intro arcD[OF assms]) (use xy in auto)
  thus "x = y"
    using xy by (subst (asm) inj') auto
qed auto

lemma arc':
  assumes "simple_path q" "f 0 ≠ 0 ∨ f 1 ≠ 1"
  shows   "arc p"
  unfolding arc_def
proof (safe intro!: inj_onI)
  fix x y
  assume xy: "x ∈ {0..1}" "y ∈ {0..1}" "p x = p y"
  hence *: "q (f x) = q (f y)"
    by (simp add: equiv)
  have **: "f x ∈ {0..1}" "f y ∈ {0..1}"
    by (rule in_range; use xy in simp)+
  have "f x = f y"
  proof (rule ccontr)
    assume ***: "f x ≠ f y"
    hence "{f x, f y} = {0, 1}"
      using simple_pathD[OF assms(1), of "f x" "f y"] * ** by simp
    thus False
      using assms *** xy by (auto simp: doubleton_eq_iff)
  qed
  thus "x = y"
    using xy by (simp add: inj')
qed auto

lemma simple_path:
  assumes "simple_path q"
  shows   "simple_path p"
proof (rule simple_pathI)
  show "x = 0 ∧ y = 1" if "0 ≤ x" "x < y" "y ≤ 1" "p x = p y" for x
y
  proof -
    have "q (f x) = q (f y)"
      using that by (simp add: equiv)
    moreover from that have "f x < f y"
      by (subst less_iff) auto
    ultimately have "{f x, f y} = {0,1}"
      using simple_pathD[OF assms, of "f x" "f y"] that by simp
    thus ?thesis using that
```

```
    by (auto simp: doubleton_eq_iff)
  qed
qed auto

end


context eq_paths_locale
begin

sublocale subpath: subpath_locale q p f
  by standard (auto simp: cont mono equiv)

end


locale subpath_locale_compose =
  pq: subpath_locale p q f + qr : subpath_locale q r g for p q r f g
begin

sublocale subpath_locale p r "g ∘ f"
proof
  show "strict_mono_on {0..1} (g ∘ f)"
    using qr.mono pq.mono
  proof (rule strict_mono_on_compose)
    show "f ‘ {0..1} ⊆ {0..1}"
      using pq.in_range by auto
  qed
qed (auto intro!: continuous_intros simp: pq.equiv qr.equiv)

end


definition is_subpath :: "(real ⇒ 'a :: real_normed_vector) ⇒ (real ⇒
'a) ⇒ bool"
  where "is_subpath p q ⟷ (∃f. subpath_locale p q f)"

lemma subpath_locale_refl [intro!]: "path p ⟹ subpath_locale p p (λx.
x)"
  by unfold_locales (auto intro!: strict_mono_onI)

lemma is_subpath_refl [intro!]: "path p ⟹ is_subpath p p"
  unfolding is_subpath_def by blast

lemma eq_paths_imp_subpath [intro]:
  assumes "eq_paths p q"
  shows   "is_subpath p q"
proof -
  from assms obtain f where "eq_paths_locale p q f"
```

```
    by (auto simp: eq_paths_def)
  then interpret eq_paths_locale p q f .
  interpret inv: eq_paths_locale q p "inv_into {0..1} f"
    by (rule inverse)
  show ?thesis
    unfolding is_subpath_def using inv.subpath.subpath_locale_axioms by
blast
qed

lemma is_subpath_reverse [intro]:
  "is_subpath p q ⟹ is_subpath (reversepath p) (reversepath q)"
  unfolding is_subpath_def using subpath_locale.reverse by auto

lemma is_subpath_reverse_iff:
  "is_subpath (reversepath p) (reversepath q) ⟷ is_subpath p q"
  using is_subpath_reverse reversepath_reversepath by metis

lemma is_subpath_trans [trans]:
  assumes "is_subpath p q" "is_subpath q r"
  shows   "is_subpath p r"
proof -
  from assms(1) obtain f where "subpath_locale p q f"
    by (auto simp: is_subpath_def)
  then interpret pq: subpath_locale p q f .
  from assms(2) obtain g where "subpath_locale q r g"
    by (auto simp: is_subpath_def)
  then interpret qr: subpath_locale q r g .
  interpret subpath_locale_compose p q r f g ..
  show ?thesis
    unfolding is_subpath_def using subpath_locale_axioms by blast
qed

lemma is_subpath_eq_trans [trans]:
  "p = q ⟹ is_subpath q r ⟹ is_subpath p r"
  "is_subpath p q ⟹ q = r ⟹ is_subpath p r"
  by simp_all

lemma is_subpath_eq_paths_trans [trans]:
  "eq_paths p q ⟹ is_subpath q r ⟹ is_subpath p r"
  "is_subpath p q ⟹ eq_paths q r ⟹ is_subpath p r"
  using eq_paths_imp_subpath is_subpath_trans by metis+

lemma is_subpath_imp_path_image_subset:
  "is_subpath p q ⟹ path_image p ⊆ path_image q"
  unfolding is_subpath_def using subpath_locale.path_image_subset by
blast

lemma subpath_imp_arc:
  "is_subpath p q ⟹ arc q ⟹ arc p"
```

```
  unfolding is_subpath_def using subpath_locale.arc by blast

lemma subpath_imp_simple_path:
  "is_subpath p q ⟹ simple_path q ⟹ simple_path p"
  unfolding is_subpath_def using subpath_locale.simple_path by blast

lemma is_subpath_joinI1 [intro]:
  assumes [simp]: "path p" "path q" "pathfinish p = pathstart q"
  shows   "is_subpath p (p +++ q)"
  unfolding is_subpath_def
proof
  show "subpath_locale p (p +++ q) (λx. x / 2)"
  proof
    show "p t = (p +++ q) (t / 2)" if "t ∈ {0..1}" for t
      using that by (auto simp: joinpaths_def)
  qed (auto intro!: strict_mono_onI continuous_intros)
qed

lemma is_subpath_joinI2 [intro]:
  assumes [simp]: "path p" "path q" and "pathfinish p = pathstart q"
  shows   "is_subpath q (p +++ q)"
  unfolding is_subpath_def
proof
  show "subpath_locale q (p +++ q) (λx. x / 2 + 1 / 2)"
  proof
    show "q t = (p +++ q) (t / 2 + 1 / 2)" if "t ∈ {0..1}" for t
      using that assms(3)
      by (cases "t = 1") (auto simp: joinpaths_def pathstart_def pathfinish_def)
  qed (auto intro!: strict_mono_onI continuous_intros simp: assms(3))
qed

lemma eq_paths_join_subpaths:
  assumes "path p" "0 ≤ a" "a < b" "b < c" "c ≤ 1"
  shows   "eq_paths (subpath a c p) (subpath a b p +++ subpath b c p)"
  unfolding eq_paths_def
proof
  from assms have "a < c"
    by simp
  define u where "u = (b - a) / (c - a)"
  have "u > 0"
    unfolding u_def using ‹a < c› ‹a < b› by (intro divide_pos_pos) auto
  define f where "f = (λt. if t ≤ 1 / 2 then 2 * u * t else ((c - b)
* (2 * t - 1) + b - a) / (c - a))"
  show "eq_paths_locale (subpath a c p) (subpath a b p +++ subpath b
c p) f"
  proof
    show "(subpath a b p +++ subpath b c p) t = subpath a c p (f t)"
if "t ∈ {0..1}" for t
    proof (cases "t ≤ 1 / 2")
```

```
      case True
      have "(b - a) * (2 * t) + a = (c - a) * f t + a"
        using True that ‹a < c› by (simp add: field_simps u_def f_def)
      thus ?thesis
        using that True by (simp add: joinpaths_def subpath_def)
    next
      case False
      have "(c - b) * (2 * t - 1) + b = (c - a) * f t + a"
        using False that ‹a < c› by (simp add: f_def field_simps)
      thus ?thesis
        using that False by (simp add: joinpaths_def subpath_def)
    qed
    show "continuous_on {0..1} f"
      unfolding f_def using ‹a < c›
      by (intro continuous_intros continuous_on_real_If_combine)
         (auto simp: u_def field_simps)
    show "f 0 = 0" "f 1 = 1" using ‹a < c›
      by (auto simp: field_simps f_def)
    show "path (subpath a c p)" "path (subpath a b p +++ subpath b c
p)"
      using assms by auto
    show "strict_mono_on {0..1} f"
    proof (rule strict_mono_on_atLeastAtMost_combine)
      show "strict_mono_on {0..1/2} f" using assms ‹a < c› ‹u > 0›
        by (auto simp: f_def strict_mono_on_def)
      show "strict_mono_on {1/2..1} f"
      proof (rule strict_mono_onI)
        fix r s :: real assume rs: "r ∈ {1/2..1}" "s ∈ {1/2..1}" "r <
s"
        have "f r = ((c - b) * (2 * r - 1) + b - a) / (c - a)"
          using rs by (cases "r = 1/2") (auto simp: f_def u_def field_simps)
        also have "... < ((c - b) * (2 * s - 1) + b - a) / (c - a)"
          using ‹a < b› ‹b < c›
          by (intro divide_strict_right_mono mult_strict_left_mono
                diff_strict_right_mono add_strict_right_mono rs) auto
        also have "... = f s"
          using rs by (simp add: f_def)
        finally show "f r < f s" .
      qed
    qed
  qed
qed

lemma eq_paths_join_subpaths':
  assumes "path p" "0 < b" "b < 1"
  shows   "eq_paths p (subpath 0 b p +++ subpath b 1 p)"
  using eq_paths_join_subpaths[of p 0 b 1] assms by simp
```

If we have four points a, b, c, d that lie on a line in that order, then the line

connecting b and c is a subpath of the line connecting a and d.

```
locale linepath_subpath =
  fixes a b c d :: "'a :: euclidean_space"
  assumes collinear: "betweens [a, b, c, d]"
  assumes not_degenerate: "b ≠ c"
begin

lemma collinear': "between (a, d) b" "between (a, d) c"
  using collinear between_trans1' between_trans2' not_degenerate by auto

lemma not_degenerate': "a ≠ d"
  using collinear unfolding betweens.simps between_def prod.case
  by (metis IntI Int_closed_segment closed_segment_commute ends_in_segment(1)
           not_degenerate singletonD)

definition f where "f = (λx. linepath (dist a b) (dist a c) x / dist a
d)"

lemma dist_eq:
  "dist a d = dist a b + dist b c + dist c d"
  "dist a c = dist a b + dist b c" "dist b d = dist b c + dist c d"
  using collinear collinear' by (simp_all add: between)

sublocale subpath_locale "linepath b c" "linepath a d" f
proof
  have "dist a c ≤ dist a d"
    by (simp add: dist_eq)
  thus "f 0 ≥ 0" "f 1 ≤ 1" using not_degenerate'
    by (auto simp: f_def field_simps linepath_def)
  show "continuous_on {0..1} f"
    unfolding f_def by (rule continuous_intros)+ (use not_degenerate'
in auto)
  have "f x < f y" if "x ∈ {0..1}" "y ∈ {0..1}" "x < y" for x y
  proof -
    have "dist a d > 0"
      using not_degenerate' by auto
    hence "0 < (y - x) * (dist a c - dist a b) / dist a d"
      using not_degenerate that
      by (intro mult_pos_pos divide_pos_pos) (auto simp: dist_eq)
    also have "(y - x) * (dist a c - dist a b) =
               linepath (dist a b) (dist a c) y - linepath (dist a b)
(dist a c) x"
      by (simp add: linepath_def algebra_simps)
    finally show ?thesis
      using ‹dist a d > 0› by (simp add: field_simps f_def)
  qed
  thus "strict_mono_on {0..1} f"
    by (intro strict_mono_onI)
  show "linepath b c t = linepath a d (f t)" if "t ∈ {0..1}" for t
```

```
  proof -
    have "dist a d > 0"
      using not_degenerate' by simp
    have b: "b = linepath a d (dist a b / dist a d)"
      by (rule between_conv_linepath) (use collinear' in auto)
    have c: "c = linepath a d (dist a c / dist a d)"
      by (rule between_conv_linepath) (use collinear' in auto)

    have "linepath b c t - linepath a d (f t) =
          (t * dist a c / dist a d + (dist a b - t * dist a b) / dist
a d -
             (dist a b + t * dist a c - t * dist a b) / dist a d) *R
d +
          ((dist a b + t * dist a c - t * dist a b) / dist a d - t *
dist a c / dist a d -
             ((dist a b - t * dist a b) / dist a d)) *R a"
      (is "_ = ?x *R d + ?y *R a")
      by (subst b, subst c)
         (simp add: linepath_def f_def algebra_simps add_divide_distrib)
    also have "?y = 0"
      using not_degenerate' by (simp add: field_simps)
    also have "?x = 0"
      using not_degenerate' by (simp add: field_simps)
    finally show ?thesis
      by simp
  qed
qed auto

end

lemma is_subpath_linepath:
  assumes "betweens [a, b, c, d]" "b ≠ c"
  shows   "is_subpath (linepath b c) (linepath a d)"
proof -
  interpret linepath_subpath a b c d
    by unfold_locales fact+
  show ?thesis
    unfolding is_subpath_def using subpath_locale_axioms by auto
qed
```

We can similarly consider subarcs of circular arcs.

```
locale part_circlepath_subpath =
  fixes x :: complex and r a b c d :: real
  assumes between: "betweens [a, b, c, d]"
  assumes not_degenerate: "b ≠ c"
begin

sublocale angle: linepath_subpath a b c d
  by unfold_locales (fact between not_degenerate)+
```

```
sublocale subpath_locale "part_circlepath x r b c" "part_circlepath x
r a d" angle.f
proof
  show "part_circlepath x r b c t = part_circlepath x r a d (angle.f
t)" if "t ∈ {0..1}" for t
    using that by (simp add: part_circlepath_altdef angle.equiv)
qed (use angle.mono angle.cont in auto)

end

lemma is_subpath_part_circlepath:
  assumes "betweens [a, b, c, d]" "b ≠ c"
  shows   "is_subpath (part_circlepath x r b c) (part_circlepath x r a
d)"
proof -
  interpret part_circlepath_subpath x r a b c d
    by unfold_locales fact+
  show ?thesis
    unfolding is_subpath_def using subpath_locale_axioms by auto
qed
```

2.4 Equivalence of closed paths

For loop equivalence, we additionally allow reparametrisation by a constant shift.

```
definition eq_loops :: "(real ⇒ 'a :: topological_space) ⇒ (real ⇒ 'a)
⇒ bool" where
  "eq_loops p q ⟷
     pathstart p = pathfinish p ∧ pathstart q = pathfinish q ∧ path q
∧ (∃c. eq_paths p (shiftpath' c q))"

lemma eq_paths_imp_eq_loops:
  assumes "eq_paths p q" "pathstart p = pathfinish p ∨ pathstart q =
pathfinish q"
  shows   "eq_loops p q"
  unfolding eq_loops_def
proof safe
  show *: "pathstart p = pathfinish p" "pathstart q = pathfinish q"
    using eq_paths_imp_same_ends[OF assms(1)] assms(2) by auto
  have "path p" "path q"
    using eq_paths_imp_path[OF assms(1)] by auto
  thus "∃c. eq_paths p (shiftpath' c q)"
    using assms(1) * by (intro exI[of _ 0]) auto
  show "path q"
    by fact
qed

lemma eq_loops_refl':
```

```
  assumes "path p ∨ path q" "pathstart p = pathfinish p ∨ pathstart
q = pathfinish q"
  assumes "⋀x. x ∈ {0..1} ⟹ p x = q x"
  shows   "eq_loops p q"
  by (intro eq_paths_imp_eq_loops eq_paths_refl' assms)

lemma eq_loops_refl [simp, intro, eq_paths_intros]:
  assumes [simp]: "path p" "pathstart p = pathfinish p"
  shows   "eq_loops p p"
  by (intro eq_loops_refl') auto

lemma eq_loops_imp_loop:
  assumes "eq_loops p q"
  shows   "pathstart p = pathfinish p" "pathstart q = pathfinish q"
proof -
  show "pathstart p = pathfinish p"
    using assms by (auto simp: eq_loops_def)
  show "pathstart q = pathfinish q"
    using assms unfolding eq_loops_def by auto
qed

lemma eq_loops_shiftpath'_left:
  assumes "path p" "pathstart p = pathfinish p"
  shows   "eq_loops (shiftpath' c p) p"
  unfolding eq_loops_def using assms
  by (intro conjI exI[of _ "c"]) (auto simp: pathfinish_shiftpath')

lemma eq_loops_shiftpath'_right:
  assumes "path p" "pathstart p = pathfinish p"
  shows   "eq_loops p (shiftpath' c p)"
  unfolding eq_loops_def using assms
  by (intro conjI exI[of _ "-c"]) (auto simp: pathfinish_shiftpath' shiftpath'_shiftpath')


locale eq_paths_shiftpath_locale = eq_paths_locale +
  fixes c :: real
  assumes c: "c ∈ {0..1}"
  assumes loop: "pathstart p = pathfinish p"
begin

lemma loop': "pathstart q = pathfinish q"
  using loop by (simp_all add: same_ends)

definition g where "g = (λt. if t ≤ 1 - c then f (t + c) - f c else f
(t + c - 1) - f c + 1)"

sublocale shifted: eq_paths_locale "shiftpath (f c) p" "shiftpath c q"
g
proof
```

```
  show "shiftpath c q t = shiftpath (f c) p (g t)" if "t ∈ {0..1}" for
t
  proof (cases "t + c" "1 :: real" rule: linorder_cases)
    case less thus ?thesis using that c
      by (simp add: shiftpath_def equiv add_ac g_def)
  next
    case greater thus ?thesis using that c
      by (auto simp add: shiftpath_def equiv add_ac g_def)
  next
    case equal
    thus ?thesis using that c ends
      by (auto simp: shiftpath_def g_def equiv add.commute)
  qed
  show "strict_mono_on {0..1} g"
  proof (rule strict_mono_onI)
    fix r s :: real assume rs: "r ∈ {0..1}" "s ∈ {0..1}" "r < s"
    show "g r < g s"
    proof (cases "r ≤ 1 - c ∧ s > 1 - c")
      case False
      thus ?thesis using rs c
        by (auto simp: g_def intro!: strict_mono_onD[OF mono])
    next
      case True
      have "f (r + c) ≤ 1"
        by (rule f_le_1) (use True rs in auto)
      moreover have "f (s + c - 1) > f 0"
        by (rule strict_mono_onD[OF mono]) (use rs True c in auto)
      ultimately have "f (r + c) < f (s + c - 1) + 1"
        unfolding ends by linarith
      with True show ?thesis
        by (auto simp: g_def intro!: strict_mono_onD[OF mono])
    qed
  qed
  show "continuous_on {0..1} g"
    unfolding g_def using c
    by (auto intro!: continuous_on_real_If_combine continuous_intros)
qed (use c in ‹auto simp: loop loop' g_def intro!: path_shiftpath›)

end

lemma eq_paths_locale_cong:
  assumes "⋀x. x ∈ {0..1} ⟹ p x = p' x"
  assumes "⋀x. x ∈ {0..1} ⟹ q x = q' x"
  shows   "eq_paths_locale p q f ⟷ eq_paths_locale p' q' f"
proof -
  have *: "eq_paths_locale p' q' f"
    if "eq_paths_locale p q f" "⋀x. x ∈ {0..1} ⟹ p x = p' x" "⋀x. x
∈ {0..1} ⟹ q x = q' x"
    for p p' q q' :: "real ⇒ 'a"
```

```
  proof -
    interpret pq: eq_paths_locale p q f
      by fact
    show ?thesis
    proof
      have "path p ⟷ path p'" "path q ⟷ path q'"
        by (rule path_cong; use that(2,3) in simp; fail)+
      with pq.paths show "path p'" "path q'"
        by blast+
    next
      fix t :: real assume t: "t ∈ {0..1}"
      have "q' t = q t"
        by (rule that(3)[symmetric]) fact
      also have "q t = p (f t)"
        by (rule pq.equiv) fact
      also have "p (f t) = p' (f t)"
        by (intro that(2) pq.subpath.in_range) fact
      finally show "q' t = p' (f t)" .
    qed (fact pq.cont pq.mono pq.ends)+
  qed

  show ?thesis
    using *[of p q p' q'] *[of p' q' p q] assms by metis
qed


locale eq_paths_shiftpath'_locale = eq_paths_locale +
  fixes c :: real
  assumes loop: "pathstart p = pathfinish p"
begin

definition g :: "real ⇒ real" where
  "g = (λt. if t ≤ 1 - frac c then f (t + frac c) - f (frac c) else
              f (t + frac c - 1) - f (frac c) + 1)"

sublocale shifted: eq_paths_locale "shiftpath' (f (frac c)) p" "shiftpath'
c q" g
proof -
  interpret aux: eq_paths_shiftpath_locale p q f "frac c"
    by unfold_locales (use loop in ‹auto simp: frac_lt_1 less_imp_le›)
  have "aux.g = g"
    by (simp add: g_def aux.g_def)
  hence "eq_paths_locale (shiftpath (f (frac c)) p) (shiftpath (frac c)
q) g"
    using aux.shifted.eq_paths_locale_axioms by simp
  also have "?this ⟷ eq_paths_locale (shiftpath' (f (frac c)) p) (shiftpath'
(frac c) q) g"
    by (intro eq_paths_locale_cong)
       (auto simp: loop less_imp_le[OF frac_lt_1] shiftpath'_eq_shiftpath
```

```
aux.loop')
  also have "shiftpath' (frac c) q = shiftpath' c q"
    by (simp add: shiftpath'_frac)
  finally show "eq_paths_locale (shiftpath' (f (frac c)) p) (shiftpath'
c q) g" .
qed

end

lemma eq_paths_shiftpath_shiftpath':
  "path p ⟹ pathstart p = pathfinish p ⟹ c ∈ {0..1} ⟹
     eq_paths (shiftpath c p) (shiftpath' c p)"
  by (intro eq_paths_refl' path_shiftpath) (auto simp: shiftpath'_eq_shiftpath)

lemma eq_loops_imp_path_image_eq:
  assumes "eq_loops p q"
  shows   "path_image p = path_image q"
proof -
  from assms(1) obtain c where c: "eq_paths p (shiftpath' c q)" and
[simp]:
    "pathstart p = pathfinish p" "pathstart q = pathfinish q"
    unfolding eq_loops_def by blast
  have [simp]: "path p" "path q"
    using assms by (auto simp: eq_loops_def)
  have "path_image p = path_image (shiftpath' c q)"
    using eq_paths_imp_path_image_eq[OF c] .
  also have "... = path_image q"
    by (intro path_image_shiftpath') auto
  finally show ?thesis .
qed

lemma eq_loops_imp_simple_path_iff:
  assumes "eq_loops p q"
  shows   "simple_path p ⟷ simple_path q"
proof -
  obtain c where c: "pathstart p = pathfinish p" "pathstart q = pathfinish
q" "path q"
                    "eq_paths p (shiftpath' c q)"
    using assms unfolding eq_loops_def by blast
  thus ?thesis
    using eq_paths_imp_simple_path_iff[OF c(4)] by auto
qed

lemma eq_loops_simple_path_trans [trans]:
  "eq_loops p q ⟹ simple_path p ⟹ simple_path q"
  "simple_path p ⟹ eq_loops p q ⟹ simple_path q"
  using eq_loops_imp_simple_path_iff by metis+

lemma eq_loops_imp_simple_loop_iff:
```

```
  assumes "eq_loops p q"
  shows   "simple_loop p ⟷ simple_loop q"
  using eq_loops_imp_simple_path_iff [OF assms] eq_loops_imp_loop [OF
assms]
  by (auto simp: simple_loop_def)

lemma eq_loops_imp_homotopic:
  assumes "eq_loops p q" "path_image p ∩ path_image q ⊆ A"
  shows   "homotopic_loops A p q"
proof -
  from assms(1) obtain c where c: "eq_paths p (shiftpath' c q)" and
[simp]:
      "pathstart p = pathfinish p" "pathstart q = pathfinish q"
    by (auto simp: eq_loops_def)
  from c obtain f where "eq_paths_locale p (shiftpath' c q) f"
    by (auto simp: eq_paths_def)
  then interpret eq_paths_locale p "shiftpath' c q" f .
  have "path q"
    using assms(1) eq_loops_def by blast
  have "homotopic_loops (path_image p) p (shiftpath' c q)"
    using c path_image_eq same_ends
    by (intro homotopic_paths_imp_homotopic_loops homotopic) (auto simp:
pathfinish_shiftpath')
  also have "homotopic_loops (path_image p) (shiftpath' c q) q"
    using eq_loops_imp_path_image_eq[OF assms(1)] ‹path q›
    by (intro homotopic_loops_shiftpath'_left) auto
  finally show ?thesis
    by (rule homotopic_loops_subset) (use assms eq_loops_imp_path_image_eq[OF
assms(1)] in auto)
qed

lemma eq_loops_imp_contour_integral_eq:
  assumes "eq_loops p q" "valid_path p" "valid_path q"
  assumes "f analytic_on (path_image p ∩ path_image q)"
  shows   "contour_integral p f = contour_integral q f"
proof -
  from assms(4) obtain A where A: "open A" "f holomorphic_on A" "path_image
p ∩ path_image q ⊆ A"
    using analytic_on_holomorphic by auto
  show ?thesis
  proof (rule Cauchy_theorem_homotopic_loops)
    show "homotopic_loops A p q"
      by (intro eq_loops_imp_homotopic assms A)
  qed (use assms A in auto)
qed

lemma eq_loops_homotopic_loops_trans [trans]:
  "eq_loops p q ⟹ homotopic_loops A q r ⟹ homotopic_loops A p r"
  "homotopic_loops A p q ⟹ eq_loops q r ⟹ homotopic_loops A p r"
```

```
proof -
  show "eq_loops p q ⟹ homotopic_loops A q r ⟹ homotopic_loops A
p r"
    using eq_loops_imp_homotopic
    by (metis homotopic_loops_imp_subset homotopic_loops_trans le_infI2)
  show "homotopic_loops A p q ⟹ eq_loops q r ⟹ homotopic_loops A
p r"
    using eq_loops_imp_homotopic
    by (metis eq_loops_imp_path_image_eq homotopic_loops_imp_subset homotopic_loops_trans
inf_idem)
qed

lemma eq_loops_imp_winding_number_eq:
  assumes "eq_loops p q" "z ∉ path_image p ∩ path_image q"
  shows   "winding_number p z = winding_number q z"
proof (rule winding_number_homotopic_loops)
  show "homotopic_loops (-{z}) p q"
    by (rule eq_loops_imp_homotopic[OF assms(1)]) (use assms(2) in auto)
qed

lemma
  assumes "eq_loops p q"
  shows   eq_loops_imp_ccw_iff: "simple_loop_ccw p = simple_loop_ccw q"
    and   eq_loops_imp_cw_iff: "simple_loop_cw p = simple_loop_cw q"
  unfolding simple_loop_ccw_def simple_loop_cw_def
  using eq_loops_imp_path_image_eq[OF assms] eq_loops_imp_winding_number_eq[OF
assms]
  by (intro conj_cong eq_loops_imp_simple_loop_iff assms ex_cong1; simp)+

lemma eq_loops_imp_same_orientation:
  assumes "eq_loops p q"
  shows   "simple_loop_orientation p = simple_loop_orientation q"
  unfolding simple_loop_orientation_def
  using eq_loops_imp_ccw_iff[OF assms] eq_loops_imp_cw_iff[OF assms] by
auto

lemma eq_loops_ccw_trans [trans]:
  "eq_loops p q ⟹ simple_loop_ccw q ⟹ simple_loop_ccw p"
  "simple_loop_ccw p ⟹ eq_loops p q ⟹ simple_loop_ccw q"
  using eq_loops_imp_ccw_iff by metis+

lemma eq_loops_cw_trans [trans]:
  "eq_loops p q ⟹ simple_loop_cw q ⟹ simple_loop_cw p"
  "simple_loop_cw p ⟹ eq_loops p q ⟹ simple_loop_cw q"
  using eq_loops_imp_cw_iff by metis+

lemma eq_loops_winding_number_trans [trans]:
  "eq_loops p q ⟹ winding_number q z = a ⟹ z ∉ path_image p ∩ path_image
q ⟹
```

```
   winding_number p z = a"
  using eq_loops_imp_winding_number_eq by metis

lemma eq_loops_simple_loop_trans [trans]:
  "eq_loops p q ⟹ simple_loop p ⟹ simple_loop q"
  "simple_loop p ⟹ eq_loops p q ⟹ simple_loop q"
  using eq_loops_imp_simple_loop_iff by metis+

lemma eq_loops_trans [trans]:
  assumes "eq_loops p q" "eq_loops q r"
  shows   "eq_loops p r"
proof -
  from assms obtain c d where
    1: "eq_paths p (shiftpath' c q)" and 2: "eq_paths (shiftpath' d r)
q"
    by (auto simp: eq_loops_def eq_paths_sym_iff)

  have [simp]: "pathstart q = pathfinish q" "pathstart p = pathfinish
p"
               "pathstart r = pathfinish r" "path r"
    using assms by (auto simp: eq_loops_def)

  from 1 obtain f where "eq_paths_locale p (shiftpath' c q) f"
    by (auto simp: eq_paths_def)
  then interpret pq: eq_paths_locale p "shiftpath' c q" f .
  obtain g where "eq_paths_locale (shiftpath' d r) q g"
    using 2 by (auto simp: eq_paths_def)
  then interpret qr: eq_paths_locale "shiftpath' d r" q g .

  interpret pq': eq_paths_shiftpath'_locale "shiftpath' d r" q g c
    by unfold_locales (use <pathstart q = pathfinish q> qr.same_ends(1)
qr.same_ends(2) in metis)

  interpret pq'': eq_paths_locale "shiftpath' c q" "shiftpath' (g (frac
c)) (shiftpath' d r)"
                  "inv_into {0..1} pq'.g"
    using pq'.shifted.inverse by simp

  interpret pqr: eq_paths_locale_compose p "shiftpath' c q"
    "shiftpath' (g (frac c)) (shiftpath' d r)" f "inv_into {0..1} pq'.g"
..

  have "eq_paths p (shiftpath' (g (frac c)) (shiftpath' d r))"
    using pqr.eq_paths_locale_axioms unfolding eq_paths_def by blast
  also have "... = shiftpath' (g (frac c) + d) r"
    by (simp add: shiftpath'_shiftpath')
  finally show ?thesis
    unfolding eq_loops_def by auto
qed
```

```
lemma eq_loops_eq_trans [trans]:
  "p = q ⟹ eq_loops q r ⟹ eq_loops p r"
  "eq_loops p q ⟹ q = r ⟹ eq_loops p r"
  by simp_all

lemma eq_loops_sym:
  assumes "eq_loops p q"
  shows   "eq_loops q p"
proof -
  have [simp]: "pathstart p = pathfinish p" "pathstart q = pathfinish
q"
    using assms by (auto simp: eq_loops_def)
  from assms have  [simp]: "path p" "path q"
    by (auto simp: eq_loops_def)
  from assms obtain c where "eq_paths p (shiftpath' c q)"
    by (auto simp: eq_loops_def)
  then obtain f where "eq_paths_locale p (shiftpath' c q) f"
    by (auto simp: eq_paths_def)
  then interpret pq: eq_paths_locale p "shiftpath' c q" f .
  interpret pq': eq_paths_shiftpath'_locale p "shiftpath' c q" f "-c"
    by standard auto
  have "eq_paths (shiftpath' (f (frac (- c))) p) (shiftpath' (- c) (shiftpath'
c q))"
    unfolding eq_paths_def using pq'.shifted.eq_paths_locale_axioms by
blast
  also have "... = shiftpath' 0 q"
    by (simp add: shiftpath'_shiftpath')
  also have "eq_paths ... q"
    by simp
  finally have "eq_paths q (shiftpath' (f (frac (- c))) p)"
    by (rule eq_paths_sym)
  thus ?thesis
    unfolding eq_loops_def by auto
qed

lemma eq_loops_sym_iff: "eq_loops p q ⟷ eq_loops q p"
  using eq_loops_sym by metis

lemma eq_loops_shiftpath'_leftI:
  assumes "eq_loops p q"
  shows   "eq_loops (shiftpath' c p) q"
proof -
  have [simp]: "pathstart p = pathfinish p" "pathstart q = pathfinish
q" "path p" "path q"
    using assms by (auto simp: eq_loops_def)
  have "eq_loops (shiftpath' c p) p"
    by (intro eq_loops_shiftpath'_left) auto
  also note ‹eq_loops p q›
```

```
  finally show "eq_loops (shiftpath' c p) q" .
qed

lemma eq_loops_shiftpath'_rightI:
  assumes "eq_loops q p"
  shows   "eq_loops q (shiftpath' c p)"
  using eq_loops_shiftpath'_leftI[of p q] assms by (simp add: eq_loops_sym_iff)

lemma path_shiftpath'_iff [simp]:
  assumes "pathstart p = pathfinish p"
  shows   "path (shiftpath' c p) ⟷ path p"
proof
  assume *: "path (shiftpath' c p)"
  have "path (shiftpath' (-c) (shiftpath' c p))"
    by (rule path_shiftpath') (use assms * in ‹auto simp: pathfinish_shiftpath'›)
  hence "path (shiftpath' 0 p)"
    by (simp add: shiftpath'_shiftpath')
  also have "?this ⟷ path p"
  proof (rule path_cong)
    show "shiftpath' 0 p x = p x" if "x ∈ {0..1}" for x
      using assms that frac_eq[of x]
      by (cases "x < 1") (auto simp: pathstart_def pathfinish_def shiftpath'_def)
  qed
  finally show "path p"
    by auto
qed (use assms in auto)

lemma eq_loops_shiftpath'_left_iff [simp]:
  assumes "pathstart p = pathfinish p"
  shows   "eq_loops (shiftpath' c p) q ⟷ eq_loops p q"
proof
  assume *: "eq_loops (shiftpath' c p) q"
  have "path (shiftpath' c p)"
    using * by (auto simp: eq_loops_def)
  hence "path p" using assms
    by (metis "*" Ints_1 diff_add_cancel eq_loops_def eq_loops_shiftpath'_rightI
eq_loops_sym path_shiftpath'_int_iff shiftpath'_shiftpath')
  have "eq_loops p (shiftpath' c p)"
    using ‹path p› assms eq_loops_shiftpath'_right by blast
  also note *
  finally show "eq_loops p q" .
qed (auto intro: eq_loops_shiftpath'_leftI)

lemma eq_loops_shiftpath'_right_iff [simp]:
  assumes "pathstart p = pathfinish p"
  shows   "eq_loops q (shiftpath' c p) ⟷ eq_loops q p"
  by (subst (1 2) eq_loops_sym_iff) (use assms in simp)

lemma eq_loops_shiftpath_shiftpath':
```

```
  assumes "pathstart p = pathfinish p" "path p" "c ∈ {0..1}"
  shows   "eq_loops (shiftpath c p) (shiftpath' c p)"
  by (rule eq_loops_refl')
     (use assms in ‹auto simp: pathfinish_shiftpath' shiftpath'_eq_shiftpath›)

lemma eq_loops_shiftpath_left_iff [simp]:
  assumes "pathstart p = pathfinish p" "c ∈ {0..1}"
  shows   "eq_loops (shiftpath c p) q ⟷ eq_loops p q"
proof
  assume *: "eq_loops p q"
  hence [simp]: "path p"
    by (auto simp: eq_loops_def)
  have "eq_loops (shiftpath c p) (shiftpath' c p)"
    by (intro eq_loops_shiftpath_shiftpath') (use assms in auto)
  also from * have "eq_loops (shiftpath' c p) q"
    using assms by simp
  finally show "eq_loops (shiftpath c p) q" .
next
  assume "eq_loops (shiftpath c p) q"
  hence "path (shiftpath c p)"
    by (auto simp: eq_loops_def)
  hence [simp]: "path p"
    using assms by (metis path_cong path_shiftpath'_iff shiftpath'_eq_shiftpath)
  have "eq_loops p (shiftpath' c p)"
    using assms by simp
  also have "eq_loops (shiftpath' c p) (shiftpath c p)"
    by (rule eq_loops_sym, rule eq_loops_shiftpath_shiftpath') (use assms
in auto)
  also have "eq_loops (shiftpath c p) q"
    by fact
  finally show "eq_loops p q" .
qed

lemma eq_loops_shiftpath_right_iff [simp]:
  assumes "pathstart p = pathfinish p" "c ∈ {0..1}"
  shows   "eq_loops q (shiftpath c p) ⟷ eq_loops q p"
  by (subst (1 2) eq_loops_sym_iff) (use assms in simp)

lemma eq_paths_shiftpath_join_onehalf:
  assumes "path p" "path q" "pathfinish p = pathstart q" "pathfinish
q = pathstart p"
  shows   "eq_paths (shiftpath (1/2) (p +++ q)) (q +++ p)"
proof (rule eq_paths_refl')
  show "shiftpath (1 / 2) (p +++ q) x = (q +++ p) x" if "x ∈ {0..1}"
for x
  proof (cases "x ∈ {0, 1 / 2, 1}")
    case True
    thus ?thesis
      using assms that by (auto simp: pathstart_def pathfinish_def shiftpath_def
```

```
joinpaths_def)
  qed (use that in ‹auto simp: shiftpath_def joinpaths_def›)
qed (use assms in auto)

lemma eq_loops_eq_paths_trans [trans]:
  "eq_loops p q ⟹ eq_paths q r ⟹ eq_loops p r"
  "eq_paths p q ⟹ eq_loops q r ⟹ eq_loops p r"
  by (meson eq_loops_def eq_loops_trans eq_paths_imp_eq_loops)+

lemma eq_loops_joinpaths:
  assumes "eq_paths p p'" "eq_paths q q'"
  assumes "pathfinish p = pathstart q" "pathfinish q = pathstart p"
  shows   "eq_loops (p +++ q) (p' +++ q')"
  by (intro eq_paths_imp_eq_loops eq_paths_intros) (use assms in auto)

lemma eq_loops_joinpaths_commute:
  assumes "path p" "path q" "pathfinish p = pathstart q" "pathfinish
q = pathstart p"
  shows   "eq_loops (p +++ q) (q +++ p)"
proof -
  have "eq_loops (p +++ q) (shiftpath (1/2) (p +++ q))"
    using assms by simp
  also have "eq_paths ... (q +++ p)"
    by (intro eq_paths_shiftpath_join_onehalf) (use assms in auto)
  finally show ?thesis .
qed

lemma eq_loops_part_circlepath:
  assumes "b = a + 2 * of_int n * pi" "b' = a' + 2 * of_int n * pi" "n
≠ 0"
  shows   "eq_loops (part_circlepath x r a b) (part_circlepath x r a'
b')"
proof -
  define c where "c = (a'-a)/(2*n*pi)"
  have "eq_loops (part_circlepath x r a b) (shiftpath' c (part_circlepath
x r a b))"
    by (rule eq_loops_shiftpath'_right)
       (auto simp: assms exp_eq_polar cis_multiple_2pi' simp flip: cis_mult)
  also have "shiftpath' c (part_circlepath x r a b) =
               part_circlepath x r (a + 2 * real_of_int n * pi * c) (a
+ 2 * real_of_int n * pi * (c + 1))"
      using shiftpath'_full_part_circlepath[of c x r a n] by (auto simp:
assms)
  also have "a + 2 * of_int n * pi * c = a'"
    using ‹n ≠ 0› by (auto simp: c_def field_simps)
  also have "a + 2 * real_of_int n * pi * (c + 1) = b'"
    using ‹n ≠ 0› by (auto simp: c_def assms(1,2) field_simps)
  finally show ?thesis .
qed
```

```
lemma eq_loops_full_part_circlepath:
  assumes "b = a + 2 * pi"
  shows   "eq_loops (part_circlepath x r a b) (circlepath x r)"
  unfolding circlepath_def
  by (rule eq_loops_part_circlepath[where n = 1]) (simp_all add: assms)

lemma eqloops_reversepath_cong:
  assumes "eq_loops p q"
  shows   "eq_loops (reversepath p) (reversepath q)"
proof -
  obtain c where pq:
     "pathstart p = pathfinish p" "pathstart q = pathfinish q" "path q"
"eq_paths p (shiftpath' c q)"
    using assms unfolding eq_loops_def by blast

  have "eq_paths (reversepath p) (shiftpath' (1-frac c) (reversepath q))"
  proof -
    have "eq_paths (reversepath p) (reversepath (shiftpath' c q))"
      by (intro eq_paths_reverse pq)
    also have "shiftpath' c q = shiftpath' (frac c) q"
      by (simp add: shiftpath'_frac)
    also have *: "eq_paths (reversepath (shiftpath' (frac c) q)) (reversepath
(shiftpath (frac c) q))"
      by (rule eq_paths_sym, intro eq_paths_reverse eq_paths_shiftpath_shiftpath')
         (use pq less_imp_le[OF frac_lt_1[of c]] in auto)
    also have "eq_paths ... (shiftpath (1 - frac c) (reversepath q))"
    proof (rule eq_paths_refl')
      show "reversepath (shiftpath (frac c) q) x = shiftpath (1 - frac
c) (reversepath q) x"
        if "x ∈ {0..1}" for x
        using that pq by (subst shiftpath_reversepath_loop) auto
    qed (use * in blast)
    also have "eq_paths ... (shiftpath' (1 - frac c) (reversepath q))"
      by (intro eq_paths_shiftpath_shiftpath') (use pq less_imp_le[OF
frac_lt_1[of c]] in auto)
    finally show ?thesis .
  qed
  thus ?thesis
    using pq unfolding eq_loops_def by auto
qed
```

2.5 Notation

Lastly, we introduce some convenient notation for these relations.

```
bundle path_rel_notation
begin

notation eq_paths (infix "≡ₚ" 60)
```

notation *eq_loops* (**infix** "$\equiv_\circ$" 60)
notation *is_subpath* (**infix** "$\leq_p$" 60)

end

unbundle *path_rel_notation*

2.6 Examples

lemma "linepath 0 1 +++ linepath 1 (3::complex) $\equiv_p$ linepath 0 3"
  by (intro eq_paths_intros) (auto simp: closed_segment_same_Im closed_segment_eq_real_ivl)

lemma "linepath 0 1 +++ linepath 1 (3::complex) $\equiv_p$ linepath 0 3"
  by (intro eq_paths_intros) (auto simp: closed_segment_same_Im closed_segment_eq_real_ivl)

end

3 Automation for paths

theory *Path_Automation*
  imports "HOL-Library.Sublist" Path_Equivalence
begin

In this section, we provide some machinery to make certain common arguments about paths easier. In particular:

- Proving the equivalence of some combination of lines and circular arcs modulo associativity
- Proving the equivalence of loops modulo associativity and "rotation"
- Proving subpath relationships
- Decomposing a contour integral over a composite path into the contour integrals of its constituent paths

Equivalence arguments that involve splitting, e.g. `linepath 0 1 +++ linepath 1 (2::'a)` $\equiv_p$ `linepath 0 (2::'a)` are not supported.

3.1 Joining a list of paths together

The following operation takes a non-empty list of paths and joines them together left-to-right, i.e. it is an `n`-ary version of `(+++)`. Associativity is to the right.

A list of paths is considered well-formed if it is non-empty, each path is indeed a well-formed path, and each successive pair of paths has compatible ends.

```
fun joinpaths_list :: "(real ⇒ 'a :: real_normed_vector) list ⇒ real
⇒ 'a" where
  "joinpaths_list [] = linepath 0 0"
| "joinpaths_list [p] = p"
| "joinpaths_list (p # ps) = p +++ joinpaths_list ps"

lemma joinpaths_list_Cons [simp]: "ps ≠ [] ⟹ joinpaths_list (p # ps)
= p +++ joinpaths_list ps"
  by (cases ps) auto

fun wf_pathlist :: "(real ⇒ 'a :: real_normed_vector) list ⇒ bool" where
  "wf_pathlist [] ⟷ False"
| "wf_pathlist [p] ⟷ path p"
| "wf_pathlist (p # q # ps) ⟷ path p ∧ path q ∧ pathfinish p = pathstart
q ∧ wf_pathlist (q # ps)"

fun weak_wf_pathlist :: "(real ⇒ 'a :: real_normed_vector) list ⇒ bool"
where
  "weak_wf_pathlist [] ⟷ False"
| "weak_wf_pathlist [p] ⟷ True"
| "weak_wf_pathlist (p # q # ps) ⟷ pathfinish p = pathstart q ∧ weak_wf_pathlist
(q # ps)"

fun arc_joinpaths_list_aux :: "(real ⇒ 'a :: real_normed_vector) list
⇒ bool" where
  "arc_joinpaths_list_aux [] ⟷ False"
| "arc_joinpaths_list_aux [p] ⟷ True"
| "arc_joinpaths_list_aux (p # q # ps) ⟷
     path_image p ∩ path_image q ⊆ {pathfinish p} ∧
     (∀r∈set ps. path_image p ∩ path_image r = {}) ∧
     arc_joinpaths_list_aux (q # ps)"

definition arc_joinpaths_list :: "(real ⇒ 'a :: real_normed_vector) list
⇒ bool" where
  "arc_joinpaths_list ps ⟷ arc_joinpaths_list_aux ps ∧ (∀p∈set ps.
arc p)"

fun simple_joinpaths_list :: "(real ⇒ 'a :: real_normed_vector) list
⇒ bool" where
  "simple_joinpaths_list [] ⟷ False"
| "simple_joinpaths_list [p] ⟷ simple_path p"
| "simple_joinpaths_list [p, q] ⟷
     path_image p ∩ path_image q ⊆ {pathfinish p} ∪ ({pathstart p} ∩
{pathfinish q}) ∧ arc p ∧ arc q"
| "simple_joinpaths_list (p # q # ps) ⟷
     path_image p ∩ path_image q ⊆ {pathfinish p} ∧
```

```
    (∀r∈set (butlast ps). path_image p ∩ path_image r = {}) ∧
    path_image p ∩ path_image (last ps) ⊆ {pathstart p} ∩ {pathfinish
(last ps)} ∧
    arc_joinpaths_list_aux (q # ps) ∧ arc p ∧ (∀r∈set (q#ps). arc r)"

lemma simple_joinpaths_list_Cons [simp]:
  assumes "ps ≠ []"
  shows "simple_joinpaths_list (p # q # ps) ⟷
          path_image p ∩ path_image q ⊆ {pathfinish p} ∧
    (∀r∈set (butlast ps). path_image p ∩ path_image r = {}) ∧
    path_image p ∩ path_image (last ps) ⊆ {pathstart p} ∩ {pathfinish
(last ps)} ∧
    arc_joinpaths_list_aux (q # ps) ∧ arc p ∧ (∀q∈set (q#ps). arc q)"
  using assms by (cases ps rule: simple_joinpaths_list.cases) simp_all


lemma wf_pathlist_Cons:
  "wf_pathlist (p # ps) ⟷ path p ∧ (ps = [] ∨ pathfinish p = pathstart
(hd ps) ∧ wf_pathlist ps)"
  by (induction ps arbitrary: p) auto

lemma weak_wf_pathlist_Cons:
  "weak_wf_pathlist (p # ps) ⟷ (ps = [] ∨ pathfinish p = pathstart
(hd ps) ∧ weak_wf_pathlist ps)"
  by (induction ps arbitrary: p) auto

fun valid_path_pathlist where
  "valid_path_pathlist [] ⟷ False"
| "valid_path_pathlist [p] ⟷ valid_path p"
| "valid_path_pathlist (p # ps) ⟷ valid_path p ∧ valid_path_pathlist
ps"

lemma valid_path_pathlist_Cons:
  "valid_path_pathlist (p # ps) ⟷ valid_path p ∧ (ps = [] ∨ valid_path_pathlist
ps)"
  by (cases ps) auto

lemma valid_path_pathlist_altdef: "valid_path_pathlist xs ⟷ xs ≠
[] ∧ list_all valid_path xs"
  by (induction xs) (auto simp: valid_path_pathlist_Cons)

lemma valid_path_weak_wf_pathlist_imp_wf:
  "valid_path_pathlist ps ⟹ weak_wf_pathlist ps ⟹ wf_pathlist ps"
  by (induction ps)
     (auto dest: valid_path_imp_path simp: valid_path_pathlist_Cons
                 weak_wf_pathlist_Cons wf_pathlist_Cons)

lemma wf_pathlist_append:
  assumes "ps ≠ []" "qs ≠ []"
```

```
  shows   "wf_pathlist (ps @ qs) ⟷
             wf_pathlist ps ∧ wf_pathlist qs ∧ pathfinish (last ps) =
pathstart (hd qs)"
  using assms
  by (induction ps arbitrary: qs rule: wf_pathlist.induct) (auto simp:
wf_pathlist_Cons)

lemma wf_pathlist_append':
  "wf_pathlist (ps @ qs) ⟷ (ps = [] ∧ wf_pathlist qs) ∨ (qs = [] ∧
wf_pathlist ps) ∨
     (wf_pathlist ps ∧ wf_pathlist qs ∧ pathfinish (last ps) = pathstart
(hd qs))"
  using wf_pathlist_append[of ps qs] by (cases "ps = []"; cases "qs =
[]") auto

lemma weak_wf_pathlist_append:
  assumes "ps ≠ []" "qs ≠ []"
  shows   "weak_wf_pathlist (ps @ qs) ⟷
             weak_wf_pathlist ps ∧ weak_wf_pathlist qs ∧ pathfinish (last
ps) = pathstart (hd qs)"
  using assms
  by (induction ps arbitrary: qs rule: weak_wf_pathlist.induct) (auto
simp: weak_wf_pathlist_Cons)

lemma weak_wf_pathlist_append':
  "weak_wf_pathlist (ps @ qs) ⟷ (ps = [] ∧ weak_wf_pathlist qs) ∨ (qs
= [] ∧ weak_wf_pathlist ps) ∨
     (weak_wf_pathlist ps ∧ weak_wf_pathlist qs ∧ pathfinish (last ps)
= pathstart (hd qs))"
  using weak_wf_pathlist_append[of ps qs] by (cases "ps = []"; cases "qs
= []") auto

lemma pathstart_joinpaths_list [simp]:
  "xs ≠ [] ⟹ pathstart (joinpaths_list xs) = pathstart (hd xs)"
  by (induction xs rule: joinpaths_list.induct) auto

lemma pathfinish_joinpaths_list [simp]:
  "xs ≠ [] ⟹ pathfinish (joinpaths_list xs) = pathfinish (last xs)"
  by (induction xs rule: joinpaths_list.induct) auto

lemma path_joinpaths_list [simp, intro]: "wf_pathlist xs ⟹ path (joinpaths_list
xs)"
  by (induction xs rule: joinpaths_list.induct) auto

lemma valid_path_joinpaths_list [intro]:
  "valid_path_pathlist xs ⟹ weak_wf_pathlist xs ⟹ valid_path (joinpaths_list
xs)"
  by (induction xs rule: joinpaths_list.induct) (auto intro!: valid_path_join)
```

```
lemma path_image_joinpaths_list:
  assumes "wf_pathlist ps"
  shows   "path_image (joinpaths_list ps) = (⋃p∈set ps. path_image p)"
  using assms by (induction ps rule: wf_pathlist.induct) (auto simp: path_image_join)

lemma joinpaths_list_append:
  assumes "wf_pathlist xs" "wf_pathlist ys" "pathfinish (last xs) = pathstart
(hd ys)"
  shows   "joinpaths_list (xs @ ys) ≡p joinpaths_list xs +++ joinpaths_list
ys"
proof -
  from assms(1) have "xs ≠ []"
    by auto
  from assms show ?thesis
  proof (induction xs arbitrary: ys rule: joinpaths_list.induct)
    case (2 p ys)
    have "ys ≠ []"
      using 2 by auto
    then obtain y ys' where [simp]: "ys = y # ys'"
      by (cases ys) auto
    show ?case using 2 by auto
  next
    case (3 p1 p2 ps qs)
    obtain q qs' where [simp]: "qs = q # qs'"
      using 3 by (cases qs) auto
    have "joinpaths_list ((p1 # p2 # ps) @ qs) =
          p1 +++ joinpaths_list ((p2 # ps) @ qs)"
      by simp
    also have "... ≡p p1 +++ joinpaths_list (p2 # ps) +++ joinpaths_list
qs"
      using 3 by (intro eq_paths_join eq_paths_refl 3) auto
    also have "... ≡p (p1 +++ joinpaths_list (p2 # ps)) +++ joinpaths_list
qs"
      by (intro eq_paths_join_assoc2) (use 3 in auto)
    finally show ?case
      by simp
  qed auto
qed

lemma arc_joinpaths_list_weak_wf_imp_wf:
  assumes "weak_wf_pathlist xs" "arc_joinpaths_list xs"
  shows   "wf_pathlist xs"
  using assms
  by (induction xs rule: wf_pathlist.induct) (auto intro: arc_imp_path
simp: arc_joinpaths_list_def)

lemma arc_joinpaths_aux:
  assumes "wf_pathlist xs" "arc_joinpaths_list_aux xs" "∀x∈set xs. arc
x"
```

```
  shows   "arc (joinpaths_list xs)"
  using assms
proof (induction xs rule: wf_pathlist.induct)
  case (3 p q ps)
  thus ?case
    by (fastforce intro!: arc_join simp: path_image_joinpaths_list)
qed auto

lemma arc_joinpaths_list [intro?]:
  assumes "weak_wf_pathlist xs" "arc_joinpaths_list xs"
  shows   "arc (joinpaths_list xs)"
  using assms arc_joinpaths_aux[of xs] arc_joinpaths_list_weak_wf_imp_wf[of
xs]
  by (auto simp: arc_joinpaths_list_def)

lemma simple_joinpaths_list_weak_wf_imp_wf:
  assumes "weak_wf_pathlist xs" "simple_joinpaths_list xs"
  shows   "wf_pathlist xs"
  using arc_joinpaths_list_weak_wf_imp_wf[of "tl xs"] assms
  by (cases xs rule: simple_joinpaths_list.cases)
     (auto dest: simple_path_imp_path arc_imp_path simp: arc_joinpaths_list_def)

lemma simple_path_joinpaths_list [intro?]:
  assumes "weak_wf_pathlist xs" "simple_joinpaths_list xs"
  shows   "simple_path (joinpaths_list xs)"
proof (cases xs rule: simple_joinpaths_list.cases)
  case (3 p q)
  thus ?thesis using assms
    by (force split: if_splits intro!: simple_path_joinI)
next
  case (4 p q r rs)
  define rs' where "rs' = r # rs"
  have [simp]: "rs' ≠ []"
    by (auto simp: rs'_def)
  have [simp]: "xs = p # q # rs'"
    by (simp add: 4 rs'_def)
  note [simp] = wf_pathlist_Cons

  have "simple_path (p +++ joinpaths_list (q # rs'))"
  proof (rule simple_path_joinI)
    show "arc p"
      using assms by auto
  next
    show "arc (joinpaths_list (q # rs'))" using assms
      by (intro arc_joinpaths_list) (auto split: if_splits simp: arc_joinpaths_list_def)
  next
    have *: "set rs' = insert (last rs') (set (butlast rs'))"
      by (subst append_butlast_last_id [symmetric]) (auto simp del: append_butlast_last_id)
    have "wf_pathlist (q # rs')"
```

```
      using assms arc_joinpaths_list_weak_wf_imp_wf[of "q # rs'"]
      by (auto simp: arc_joinpaths_list_def)
    thus "path_image p ∩ path_image (joinpaths_list (q # rs'))
            ⊆ insert (pathstart (joinpaths_list (q # rs')))
           (if pathstart p = pathfinish (joinpaths_list (q # rs')) then
{pathstart p} else {})"
      using assms by (subst path_image_joinpaths_list) (auto simp: *)
  qed (use assms in auto)
  thus ?thesis
    by (simp add: rs'_def)
qed (use assms in auto)

lemma wf_pathlist_sublist:
  assumes "wf_pathlist ys" "sublist xs ys" "xs ≠ []"
  shows   "wf_pathlist xs"
proof -
  from assms(2) obtain as bs where *: "ys = as @ xs @ bs"
    by (auto simp: sublist_def)
  have **: "wf_pathlist xs" if "wf_pathlist (xs @ bs)"
    using that ‹xs ≠ []› by (induction xs rule: wf_pathlist.induct) (auto
simp: wf_pathlist_Cons)
  show ?thesis
    using assms(1) ‹xs ≠ []› unfolding *
    by (induction as) (auto simp: ** wf_pathlist_Cons)
qed


lemma is_subpath_joinpaths_list_append_right:
  assumes "wf_pathlist (xs @ ys)" "xs ≠ []"
  shows   "is_subpath (joinpaths_list xs) (joinpaths_list (xs @ ys))"
proof (cases "ys = []")
  case False
  hence "is_subpath (joinpaths_list xs) (joinpaths_list xs +++ joinpaths_list
ys)"
    using assms by (intro is_subpath_joinI1 path_joinpaths_list) (auto
simp: wf_pathlist_append)
  also have "eq_paths ... (joinpaths_list (xs @ ys))"
    using False assms by (intro eq_paths_sym[OF joinpaths_list_append])
                         (auto simp: wf_pathlist_append)
  finally show ?thesis .
qed (use assms in auto)

lemma is_subpath_joinpaths_list_append_left:
  assumes "wf_pathlist (xs @ ys)" "ys ≠ []"
  shows   "is_subpath (joinpaths_list ys) (joinpaths_list (xs @ ys))"
proof (cases "xs = []")
  case False
  hence "is_subpath (joinpaths_list ys) (joinpaths_list xs +++ joinpaths_list
ys)"
```

```
    using assms by (intro is_subpath_joinI2 path_joinpaths_list) (auto
simp: wf_pathlist_append)
  also have "eq_paths ... (joinpaths_list (xs @ ys))"
    using False assms by (intro eq_paths_sym[OF joinpaths_list_append])
                         (auto simp: wf_pathlist_append)
  finally show ?thesis .
qed (use assms in auto)

lemma is_subpath_joinpaths_list:
  assumes "wf_pathlist ys" "sublist xs ys" "xs ≠ []"
  shows   "is_subpath (joinpaths_list xs) (joinpaths_list ys)"
proof -
  from assms(2) obtain as bs where *: "ys = as @ xs @ bs"
    by (auto simp: sublist_def)
  have "is_subpath (joinpaths_list xs) (joinpaths_list (xs @ bs))"
    using assms by (intro is_subpath_joinpaths_list_append_right)
                   (auto simp: wf_pathlist_append' *)
  also have "is_subpath ... (joinpaths_list (as @ xs @ bs))"
    using assms by (intro is_subpath_joinpaths_list_append_left)
                   (auto simp: wf_pathlist_append' *)
  finally show ?thesis
    by (simp add: *)
qed

lemma eq_loops_joinpaths_list_append:
  assumes "wf_pathlist (xs @ ys)" "pathfinish (last (xs @ ys)) = pathstart
(hd (xs @ ys))"
  shows   "eq_loops (joinpaths_list (xs @ ys)) (joinpaths_list (ys @ xs))"
proof (cases "xs = [] ∨ ys = []")
  case True
  have "xs ≠ [] ∨ ys ≠ []"
    using assms by auto
  with True show ?thesis
    using assms by auto
next
  case False
  have "eq_paths (joinpaths_list (xs @ ys)) (joinpaths_list xs +++ joinpaths_list
ys)"
    using assms False by (intro joinpaths_list_append) (auto simp: wf_pathlist_append)
  also have "eq_loops ... (joinpaths_list ys +++ joinpaths_list xs)"
    using assms False by (intro eq_loops_joinpaths_commute) (auto simp:
wf_pathlist_append)
  also have "eq_paths ... (joinpaths_list (ys @ xs))"
    using assms False
    by (intro eq_paths_sym[OF joinpaths_list_append]) (auto simp: wf_pathlist_append)
  finally show ?thesis .
qed

lemma eq_loops_rotate:
```

```
  assumes "wf_pathlist xs" "pathfinish (last xs) = pathstart (hd xs)"
  shows   "eq_loops (joinpaths_list xs) (joinpaths_list (rotate n xs))"
proof -
  define m where "m = n mod length xs"
  have "eq_loops (joinpaths_list (take m xs @ drop m xs))
                 (joinpaths_list (drop m xs @ take m xs))"
    using assms by (intro eq_loops_joinpaths_list_append) auto
  thus ?thesis
    by (simp add: m_def rotate_drop_take)
qed

lemma winding_number_joinpaths_list:
  assumes "wf_pathlist ps" "⋀p. p ∈ set ps ⟹ x ∉ path_image p"
  shows   "winding_number (joinpaths_list ps) x = (∑p←ps. winding_number
p x)"
  using assms
proof (induction ps rule: wf_pathlist.induct)
  case (3 p q ps)
  have "winding_number (joinpaths_list (p # q # ps)) x =
        winding_number (p +++ joinpaths_list (q # ps)) x"
    by simp
  also have "... = winding_number p x + winding_number (joinpaths_list
(q # ps)) x"
    using "3.prems" by (intro winding_number_join) (auto simp: path_image_joinpaths_list)
  also have "winding_number (joinpaths_list (q # ps)) x = (∑r←q#ps.
winding_number r x)"
    by (intro "3.IH") (use "3.prems" in auto)
  finally show ?case
    by simp
qed auto

lemma contour_integral_joinpaths_list:
  assumes "weak_wf_pathlist ps" "valid_path_pathlist ps"
          "f contour_integrable_on (joinpaths_list ps)"
  shows   "contour_integral (joinpaths_list ps) f = (∑p←ps. contour_integral
p f)"
  using assms
proof (induction ps rule: wf_pathlist.induct)
  case (3 p q ps)
  have wf: "wf_pathlist (p # q # ps)"
    using "3.prems" valid_path_weak_wf_pathlist_imp_wf by blast
  have int: "f contour_integrable_on (p +++ joinpaths_list (q # ps))"
    using "3.prems" by simp
  have int1: "f contour_integrable_on p"
    using contour_integrable_joinD1[OF int] "3.prems" by auto
  have int2: "f contour_integrable_on joinpaths_list (q # ps)"
    using contour_integrable_joinD2[OF int] "3.prems" by auto

  have "contour_integral (joinpaths_list (p # q # ps)) f =
```

```
        contour_integral (p +++ joinpaths_list (q # ps)) f"
    by simp
  also have "... = contour_integral p f + contour_integral (joinpaths_list
(q # ps)) f"
    using "3.prems" int1 int2 by (intro contour_integral_join) auto
  also have "contour_integral (joinpaths_list (q # ps)) f = (∑r←q#ps.
contour_integral r f)"
    by (intro "3.IH") (use "3.prems" int2 in auto)
  finally show ?case
    by simp
qed auto
```

3.2 Representing a sequence of path joins as a tree

To deal with the problem that path joining is not associative, we define an expression tree to represent all the possible different bracketings of joining `n` paths together.

There is also a "flattening" operation to convert the tree to a list of paths, since our eventual goal is to show that the order does not matter (up to path equivalence).

Well-formedness is again defined similarly to the list case.

```
datatype 'a joinpaths_tree =
  Path "real ⇒ 'a" | Reverse "'a joinpaths_tree" | Join "'a joinpaths_tree"
"'a joinpaths_tree"

primrec paths_joinpaths_tree :: "'a joinpaths_tree ⇒ (real ⇒ 'a) set"
where
  "paths_joinpaths_tree (Path p) = {p}"
| "paths_joinpaths_tree (Reverse p) = paths_joinpaths_tree p"
| "paths_joinpaths_tree (Join l r) = paths_joinpaths_tree l ∪ paths_joinpaths_tree
r"

fun start_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree ⇒
'a"
and finish_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ 'a" where
  "start_joinpaths_tree (Path p) = pathstart p"
| "start_joinpaths_tree (Reverse p) = finish_joinpaths_tree p"
| "start_joinpaths_tree (Join l r) = start_joinpaths_tree l"
| "finish_joinpaths_tree (Path p) = pathfinish p"
| "finish_joinpaths_tree (Reverse p) = start_joinpaths_tree p"
| "finish_joinpaths_tree (Join l r) = finish_joinpaths_tree r"

primrec eval_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ real ⇒ 'a" where
  "eval_joinpaths_tree (Path p) = p"
| "eval_joinpaths_tree (Reverse t) = reversepath (eval_joinpaths_tree
t)"
```

```
| "eval_joinpaths_tree (Join l r) = eval_joinpaths_tree l +++ eval_joinpaths_tree
r"

primrec flatten_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ (real ⇒ 'a) list" where
  "flatten_joinpaths_tree (Path p) = [p]"
| "flatten_joinpaths_tree (Reverse t) = rev (map reversepath (flatten_joinpaths_tree
t))"
| "flatten_joinpaths_tree (Join l r) = flatten_joinpaths_tree l @ flatten_joinpaths_tree
r"

primrec wf_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ bool" where
  "wf_joinpaths_tree (Path p) ⟷ path p"
| "wf_joinpaths_tree (Reverse t) ⟷ wf_joinpaths_tree t"
| "wf_joinpaths_tree (Join l r) ⟷
     wf_joinpaths_tree l ∧ wf_joinpaths_tree r ∧ finish_joinpaths_tree
l = start_joinpaths_tree r"

primrec weak_wf_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ bool" where
  "weak_wf_joinpaths_tree (Path p) ⟷ True"
| "weak_wf_joinpaths_tree (Reverse t) ⟷ weak_wf_joinpaths_tree t"
| "weak_wf_joinpaths_tree (Join l r) ⟷
     weak_wf_joinpaths_tree l ∧ weak_wf_joinpaths_tree r ∧ finish_joinpaths_tree
l = start_joinpaths_tree r"

primrec valid_path_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ bool" where
  "valid_path_joinpaths_tree (Path p) ⟷ valid_path p"
| "valid_path_joinpaths_tree (Reverse p) ⟷ valid_path_joinpaths_tree
p"
| "valid_path_joinpaths_tree (Join l r) ⟷
     valid_path_joinpaths_tree l ∧ valid_path_joinpaths_tree r ∧ finish_joinpaths_tree
l = start_joinpaths_tree r"

primrec arc_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ bool" where
  "arc_joinpaths_tree (Path p) ⟷ arc p"
| "arc_joinpaths_tree (Reverse p) ⟷ arc_joinpaths_tree p"
| "arc_joinpaths_tree (Join l r) ⟷
     (∀l'∈paths_joinpaths_tree l. ∀r'∈paths_joinpaths_tree r.
          path_image l' ∩ path_image r' ⊆ {finish_joinpaths_tree l})
∧
     arc_joinpaths_tree l ∧ arc_joinpaths_tree r"

primrec simple_joinpaths_tree :: "'a :: real_normed_vector joinpaths_tree
⇒ bool" where
  "simple_joinpaths_tree (Path p) ⟷ simple_path p"
```

```
| "simple_joinpaths_tree (Reverse t) ⟷ simple_joinpaths_tree t"
| "simple_joinpaths_tree (Join l r) ⟷
     (∀l'∈paths_joinpaths_tree l. ∀r'∈paths_joinpaths_tree r.
        path_image l' ∩ path_image r' ⊆
           {finish_joinpaths_tree l} ∪ ({start_joinpaths_tree l} ∩ {finish_joinpaths_tree
r})) ∧
     arc_joinpaths_tree l ∧ arc_joinpaths_tree r"

lemma flatten_joinpaths_tree_nonempty [simp]: "flatten_joinpaths_tree
t ≠ []"
  by (induction t) auto



lemma pathstart_eval_joinpaths_tree [simp]: "pathstart (eval_joinpaths_tree
t) = start_joinpaths_tree t"
  and pathfinish_eval_joinpaths_tree [simp]: "pathfinish (eval_joinpaths_tree
t) = finish_joinpaths_tree t"
  by (induction t) auto

lemma pathstart_last_flatten_joinpaths_tree [simp]:
        "pathstart (hd (flatten_joinpaths_tree t)) = start_joinpaths_tree
t" (is ?th1)
  and pathfinish_last_flatten_joinpaths_tree [simp]:
        "pathfinish (last (flatten_joinpaths_tree t)) = finish_joinpaths_tree
t" (is ?th2)
  by (induction t and t rule: start_joinpaths_tree_finish_joinpaths_tree.induct)
     (auto simp: hd_rev last_rev hd_map last_map)

lemma wf_pathlist_map_rev [simp]: "wf_pathlist (map reversepath xs) ⟷
wf_pathlist (rev xs)"
  by (induction xs) (auto simp: wf_pathlist_Cons hd_map wf_pathlist_append'
last_rev)

lemma weak_wf_pathlist_map_rev' [simp]: "weak_wf_pathlist (map reversepath
xs) ⟷ weak_wf_pathlist (rev xs)"
  by (induction xs) (auto simp: weak_wf_pathlist_Cons weak_wf_pathlist_append'
last_rev rev_map hd_map)

lemma weak_wf_pathlist_map_rev [simp]: "weak_wf_pathlist (rev (map reversepath
xs)) ⟷ weak_wf_pathlist xs"
  by (induction xs) (auto simp: weak_wf_pathlist_Cons weak_wf_pathlist_append'
last_rev rev_map hd_map last_map)

lemma wf_pathlist_map_rev' [simp]: "wf_pathlist (rev (map reversepath
xs)) ⟷ wf_pathlist xs"
  by (induction xs) (auto simp: wf_pathlist_Cons hd_map wf_pathlist_append'
last_rev)
```

```
lemma wf_pathlist_flatten_pathree [simp]: "wf_pathlist (flatten_joinpaths_tree
t) ⟷ wf_joinpaths_tree t"
  by (induction t) (auto simp: wf_pathlist_append rev_map)

lemma weak_wf_pathlist_flatten_pathree [simp]:
  "weak_wf_pathlist (flatten_joinpaths_tree t) ⟷ weak_wf_joinpaths_tree
t"
  by (induction t) (auto simp: weak_wf_pathlist_append)

lemma reversepath_joinpaths_list:
  assumes "wf_pathlist xs"
  shows   "reversepath (joinpaths_list xs) ≡p joinpaths_list (rev (map
reversepath xs))"
  using assms
proof (induction xs rule: wf_pathlist.induct)
  case (3 p q ps)
  have "reversepath (joinpaths_list (p # q # ps)) =
        reversepath (joinpaths_list (q # ps)) +++ reversepath p"
    using 3 by (simp_all add: reversepath_joinpaths)
  also have "... ≡p joinpaths_list (rev (map reversepath (q # ps))) +++
reversepath p"
    using 3 by (intro eq_paths_join) auto
  also have "... = joinpaths_list (rev (map reversepath (q # ps))) +++
joinpaths_list [reversepath p]"
    by simp
  also have "... ≡p joinpaths_list (rev (map reversepath (q # ps)) @ [reversepath
p])"
    using 3 by (intro eq_paths_sym[OF joinpaths_list_append])
               (auto simp: wf_pathlist_append' last_rev hd_map wf_pathlist_Cons)
  finally show ?case
    by simp
qed auto

lemma joinpaths_flatten_joinpaths_tree:
  assumes "wf_joinpaths_tree t"
  shows   "eval_joinpaths_tree t ≡p joinpaths_list (flatten_joinpaths_tree
t)"
  using assms
proof (induction t)
  case (Path p)
  thus ?case by simp
next
  case (Reverse t)
  have "eval_joinpaths_tree (Reverse t) ≡p
        reversepath (joinpaths_list (flatten_joinpaths_tree t))"
    unfolding eval_joinpaths_tree.simps using Reverse.prems
    by (intro eq_paths_reverse Reverse.IH) auto
  also have "... ≡p joinpaths_list (rev (map reversepath (flatten_joinpaths_tree
t)))"
```

```
    by (intro reversepath_joinpaths_list) (use Reverse in auto)
  finally show ?case
    by simp
next
  case (Join l r)
  have "eval_joinpaths_tree l +++ eval_joinpaths_tree r ≡ₚ
          joinpaths_list (flatten_joinpaths_tree l) +++ joinpaths_list
(flatten_joinpaths_tree r)"
    using Join by (intro eq_paths_join) auto
  also have "... ≡ₚ joinpaths_list (flatten_joinpaths_tree l @ flatten_joinpaths_tree
r)"
    by (rule eq_paths_sym[OF joinpaths_list_append]) (use Join in auto)
  finally show ?case
    by simp
qed

lemma valid_path_joinpaths_tree:
  fixes t :: "'a :: real_normed_field joinpaths_tree"
  shows "valid_path_joinpaths_tree t ⟹ valid_path (eval_joinpaths_tree
t)"
  by (induction t) auto

lemma path_image_eval_joinpaths_tree:
  "wf_joinpaths_tree t ⟹
     path_image (eval_joinpaths_tree t) = (⋃p∈paths_joinpaths_tree t.
path_image p)"
  by (induction t) (auto simp: path_image_join)

lemma arc_joinpaths_tree [intro?]:
  "wf_joinpaths_tree t ⟹ arc_joinpaths_tree t ⟹ arc (eval_joinpaths_tree
t)"
  by (induction t) (auto simp: arc_join_eq path_image_eval_joinpaths_tree
intro!: arc_reversepath)

lemma simple_joinpaths_tree [intro?]:
  "wf_joinpaths_tree t ⟹ simple_joinpaths_tree t ⟹ simple_path (eval_joinpaths_tree
t)"
  by (induction t)
    (fastforce intro!: simple_path_joinI arc_joinpaths_tree split: if_splits
               simp: path_image_eval_joinpaths_tree simple_path_reversepath_iff)+
```

3.3 Equivalence of two join trees

Two trees are considered equivalent if they flatten to the same list of paths. Equivalence implies that one tree is well-formed if and only if the other one is as well, and in that case that their evaluations are equivalent paths.

```
definition equiv_joinpaths_tree ::
  "('a :: real_normed_vector joinpaths_tree) ⇒ 'a joinpaths_tree ⇒ bool"
where
```

```
  "equiv_joinpaths_tree t1 t2 ⟷ flatten_joinpaths_tree t1 = flatten_joinpaths_tree
t2"

lemma equiv_joinpaths_tree_imp_wf_iff:
  "equiv_joinpaths_tree t1 t2 ⟹ wf_joinpaths_tree t1 ⟷ wf_joinpaths_tree
t2"
  by (metis equiv_joinpaths_tree_def wf_pathlist_flatten_pathree)

lemma equiv_joinpaths_tree_imp_eval_eq:
  "equiv_joinpaths_tree t1 t2 ⟹ wf_joinpaths_tree t1 ⟹
     eval_joinpaths_tree t1 ≡p eval_joinpaths_tree t2"
  by (metis eq_paths_sym eq_paths_trans equiv_joinpaths_tree_def
            equiv_joinpaths_tree_imp_wf_iff joinpaths_flatten_joinpaths_tree)
```

3.4 Implementation

```
named_theorems path_automation_simps
named_theorems path_automation_intros
```

The following allows us to reify an expression containing join operations into a tree. One might be able to incorporate path reversal as well.

```
definition REIFY_JOINPATHS_TAG where "REIFY_JOINPATHS_TAG x = x"

lemma REIFY_JOINPATHS_TAG:
  "REIFY_JOINPATHS_TAG (x :: real ⇒ 'a :: real_normed_vector) = y ⟹
x = y"
  by (simp add: REIFY_JOINPATHS_TAG_def)

named_theorems reify_joinpath_tree

lemma reify_joinpaths_tree [reify_joinpath_tree]:
  "REIFY_JOINPATHS_TAG (reversepath p) = reversepath (REIFY_JOINPATHS_TAG
p)"
  "REIFY_JOINPATHS_TAG (p +++ q) = REIFY_JOINPATHS_TAG p +++ REIFY_JOINPATHS_TAG
q"
  "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree (Path p)"
  "eval_joinpaths_tree l +++ eval_joinpaths_tree r = eval_joinpaths_tree
(Join l r)"
  "reversepath (eval_joinpaths_tree t) = eval_joinpaths_tree (Reverse
t)"
  by (simp_all add: REIFY_JOINPATHS_TAG_def)


lemma path_via_joinpaths_tree [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "wf_joinpaths_tree t"
  shows   "path p"
  using assms joinpaths_flatten_joinpaths_tree[of t] by (auto simp: REIFY_JOINPATHS_TAG_def
```

```
lemma valid_path_via_joinpaths_tree [path_automation_intros]:
  fixes p :: "real ⇒ 'a :: real_normed_field"
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "valid_path_joinpaths_tree t"
  shows   "valid_path p"
  using assms valid_path_joinpaths_tree[of t] by (auto simp: REIFY_JOINPATHS_TAG_def)

lemma arc_via_joinpaths_tree [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "arc_joinpaths_list (flatten_joinpaths_tree t) ∧ weak_wf_joinpaths_tree
t"
  shows   "arc p"
proof -
  have wf: "wf_joinpaths_tree t"
    using arc_joinpaths_list_weak_wf_imp_wf[of "flatten_joinpaths_tree
t"] assms
    by auto
  have "arc (joinpaths_list (flatten_joinpaths_tree t))"
    using assms by (intro arc_joinpaths_list) auto
  moreover have "eval_joinpaths_tree t ≡p joinpaths_list (flatten_joinpaths_tree
t)"
    using wf by (intro joinpaths_flatten_joinpaths_tree) auto
  ultimately show ?thesis
    using assms eq_paths_imp_arc_iff unfolding REIFY_JOINPATHS_TAG_def
by metis
qed

lemma simple_path_via_joinpaths_tree [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "simple_joinpaths_list (flatten_joinpaths_tree t) ∧ weak_wf_joinpaths_tree
t"
  shows   "simple_path p"
proof -
  have wf: "wf_joinpaths_tree t"
    using simple_joinpaths_list_weak_wf_imp_wf[of "flatten_joinpaths_tree
t"] assms
    by auto
  have "simple_path (joinpaths_list (flatten_joinpaths_tree t))"
    using assms by (intro simple_path_joinpaths_list) auto
  moreover have "eval_joinpaths_tree t ≡p joinpaths_list (flatten_joinpaths_tree
t)"
    using wf by (intro joinpaths_flatten_joinpaths_tree) auto
  ultimately show ?thesis
    using assms eq_paths_imp_simple_path_iff unfolding REIFY_JOINPATHS_TAG_def
by metis
qed

lemma eq_paths_via_reify_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t1"
```

```
  assumes "REIFY_JOINPATHS_TAG q = eval_joinpaths_tree t2"
  assumes "wf_joinpaths_tree t1 ∧ wf_joinpaths_tree t2 ∧
           flatten_joinpaths_tree t1 = flatten_joinpaths_tree t2"
  shows   "eq_paths p q"
  using assms unfolding REIFY_JOINPATHS_TAG_def
  by (simp add: equiv_joinpaths_tree_def equiv_joinpaths_tree_imp_eval_eq)

definition is_rotation_of :: "'a list ⇒ 'a list ⇒ bool" where
  "is_rotation_of xs ys ⟷ (∃n. xs = rotate n ys)"

fun is_rotation_of_aux :: "'a list ⇒ 'a list ⇒ nat ⇒ bool" where
  "is_rotation_of_aux xs ys 0 ⟷ False"
| "is_rotation_of_aux xs [] _ ⟷ xs = []"
| "is_rotation_of_aux xs (y # ys) (Suc n) ⟷
    xs = y # ys ∨ is_rotation_of_aux xs (ys @ [y]) n"

lemma is_rotation_of_aux_correct: "is_rotation_of_aux xs ys n ⟷ (∃k<n.
xs = rotate k ys)"
proof (induction xs ys n rule: is_rotation_of_aux.induct)
  case (3 xs y ys n)
  show ?case
  proof
    assume "is_rotation_of_aux xs (y # ys) (Suc n)"
    hence "xs = y # ys ∨ is_rotation_of_aux xs (ys @ [y]) n"
      by auto
    thus "∃k<Suc n. xs = rotate k (y # ys)"
    proof
      assume "xs = y # ys"
      thus ?thesis
        by (intro exI[of _ 0]) auto
    next
      assume "is_rotation_of_aux xs (ys @ [y]) n"
      with 3 obtain k where "k < n" "xs = rotate k (ys @ [y])"
        by blast
      thus "∃k<Suc n. xs = rotate k (y # ys)"
        by (intro exI[of _ "Suc k"]) (auto simp: rotate1_rotate_swap)
    qed
  next
    assume "∃k<Suc n. xs = rotate k (y # ys)"
    then obtain k where k: "k < Suc n" "xs = rotate k (y # ys)"
      by blast
    show "is_rotation_of_aux xs (y # ys) (Suc n)"
    proof (cases k)
      case 0
      with k show ?thesis by simp
    next
      case (Suc k')
      with k have "k' < n" "xs = rotate k' (ys @ [y])"
        by (simp_all add: rotate1_rotate_swap)
```

```
      with 3 have "is_rotation_of_aux xs (ys @ [y]) n"
        by blast
      thus ?thesis by simp
    qed
  qed
qed auto

lemma is_rotation_of_code [code]:
  "is_rotation_of xs ys ⟷ length xs = length ys ∧ (xs = [] ∨ is_rotation_of_aux
xs ys (length xs))"
proof (intro iffI conjI)
  assume "is_rotation_of xs ys"
  then obtain n where n: "xs = rotate n ys"
    by (auto simp: is_rotation_of_def)
  also have "rotate n ys = rotate (n mod length ys) ys"
    by (simp add: rotate_drop_take)
  also have "length ys = length xs"
    by (simp add: n)
  finally have "xs = rotate (n mod length xs) ys"
    by simp
  moreover have "n mod length xs < length xs" if "xs ≠ []"
    using that by auto
  ultimately show "xs = [] ∨ is_rotation_of_aux xs ys (length xs)"
    unfolding is_rotation_of_aux_correct by blast
qed (auto simp: is_rotation_of_def is_rotation_of_aux_correct)

lemma eq_loops_via_reify_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t1"
  assumes "REIFY_JOINPATHS_TAG q = eval_joinpaths_tree t2"
  assumes "wf_joinpaths_tree t1 ∧ wf_joinpaths_tree t2 ∧
           finish_joinpaths_tree t2 = start_joinpaths_tree t2 ∧
           is_rotation_of (flatten_joinpaths_tree t1) (flatten_joinpaths_tree
t2)"
  shows   "eq_loops p q"
proof -
  from assms obtain n where n: "flatten_joinpaths_tree t1 = rotate n
(flatten_joinpaths_tree t2)"
    unfolding is_rotation_of_def by blast
  have "eq_paths (eval_joinpaths_tree t2) (joinpaths_list (flatten_joinpaths_tree
t2))"
    using assms eq_paths_sym_iff joinpaths_flatten_joinpaths_tree by blast
  also have "eq_loops ... (joinpaths_list (flatten_joinpaths_tree t1))"
    unfolding n by (intro eq_loops_rotate) (use assms in auto)
  also have "eq_paths ... (eval_joinpaths_tree t1)"
    using assms eq_paths_sym_iff joinpaths_flatten_joinpaths_tree by blast
  finally show ?thesis
    using assms by (simp add: eq_loops_sym_iff REIFY_JOINPATHS_TAG_def)
qed
```

```
lemma is_subpath_via_reify_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t1"
  assumes "REIFY_JOINPATHS_TAG q = eval_joinpaths_tree t2"
  assumes "wf_joinpaths_tree t1 ∧ wf_joinpaths_tree t2 ∧
           sublist (flatten_joinpaths_tree t1) (flatten_joinpaths_tree
t2)"
  shows   "is_subpath p q"
  using assms unfolding REIFY_JOINPATHS_TAG_def
  by (meson eq_paths_sym flatten_joinpaths_tree_nonempty is_subpath_eq_paths_trans
      is_subpath_joinpaths_list joinpaths_flatten_joinpaths_tree wf_pathlist_flatten_pathr

lemma sum_list_singleton: "sum_list [x] = x"
  by simp

lemma sum_list_Cons_rev: "sum_list (x # y # xs) = sum_list (y # xs) +
(x :: 'a :: comm_monoid_add)"
  by (simp add: add_ac)

lemma winding_number_via_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "(∑ q←rev (flatten_joinpaths_tree t). winding_number q x)
= T ∧
           (∀p∈set (flatten_joinpaths_tree t). x ∉ path_image p) ∧
           weak_wf_joinpaths_tree t ∧ valid_path_pathlist (flatten_joinpaths_tree
t)"
  shows   "winding_number p x = T"
proof -
  have wf: "wf_joinpaths_tree t"
    using assms valid_path_weak_wf_pathlist_imp_wf weak_wf_pathlist_flatten_pathree
          wf_pathlist_flatten_pathree by blast
  have "p ≡ₚ joinpaths_list (flatten_joinpaths_tree t)"
    using assms wf unfolding REIFY_JOINPATHS_TAG_def
    by (metis joinpaths_flatten_joinpaths_tree)
  hence "winding_number p x = winding_number (joinpaths_list (flatten_joinpaths_tree
t)) x"
    using assms wf by (intro eq_paths_imp_winding_number_eq) (auto simp:
path_image_joinpaths_list)
  also have "... = (∑ p←flatten_joinpaths_tree t. winding_number p x)"
    using wf assms by (subst winding_number_joinpaths_list) auto
  finally show ?thesis using assms by (simp flip: rev_map)
qed

lemma valid_path_pathlist_flatten_imp_valid_path_eval_joinpaths_tree:
  assumes "weak_wf_pathlist (flatten_joinpaths_tree t)"
  assumes "valid_path_pathlist (flatten_joinpaths_tree t)"
  shows   "valid_path (eval_joinpaths_tree t)"
  using assms
  by (induction t)
     (auto intro!: valid_path_join simp: valid_path_pathlist_altdef
```

```
                weak_wf_pathlist_append list.pred_map o_def)

lemma path_image_eval_joinpaths_tree':
  assumes "wf_joinpaths_tree t"
  shows   "path_image (eval_joinpaths_tree t) = (⋃p∈set (flatten_joinpaths_tree
t). path_image p)"
  using assms by (induction t) (simp_all add: path_image_join)


lemma contour_integral_via_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "(∑q←rev (flatten_joinpaths_tree t). contour_integral q f)
= T ∧
           f analytic_on (path_image p) ∧
           weak_wf_joinpaths_tree t ∧ valid_path_pathlist (flatten_joinpaths_tree
t)"
  shows   "contour_integral p f = T"
proof -
  have valid: "valid_path (eval_joinpaths_tree t)"
    by (intro valid_path_pathlist_flatten_imp_valid_path_eval_joinpaths_tree)
(use assms in auto)
  have wf: "wf_joinpaths_tree t"
    using assms valid_path_weak_wf_pathlist_imp_wf weak_wf_pathlist_flatten_pathree
          wf_pathlist_flatten_pathree by blast
  have int: "f contour_integrable_on joinpaths_list (flatten_joinpaths_tree
t)"
    using assms wf path_image_eval_joinpaths_tree'[OF wf]
    by (intro analytic_imp_contour_integrable valid_path_joinpaths_list)
       (auto simp: path_image_joinpaths_list REIFY_JOINPATHS_TAG_def)
  have eq: "p ≡p joinpaths_list (flatten_joinpaths_tree t)"
    using assms wf unfolding REIFY_JOINPATHS_TAG_def
    by (metis joinpaths_flatten_joinpaths_tree)
  moreover have "f analytic_on path_image (eval_joinpaths_tree t) ∩ ⋃
(path_image ‘ set (flatten_joinpaths_tree t))"
  proof (rule analytic_on_subset[of f "path_image p"])
    have "path_image (eval_joinpaths_tree t) = path_image p"
      using assms by (simp add: path_image_joinpaths_list REIFY_JOINPATHS_TAG_def)
    thus "path_image (eval_joinpaths_tree t) ∩ ⋃ (path_image ‘ set (flatten_joinpaths_tree
t))
          ⊆ path_image p" by simp
  qed (use assms in auto)
  ultimately have "contour_integral p f = contour_integral (joinpaths_list
(flatten_joinpaths_tree t)) f"
    using assms wf valid
    by (intro eq_paths_imp_contour_integral_eq)
       (auto simp: path_image_joinpaths_list REIFY_JOINPATHS_TAG_def)
  also have "... = (∑p←flatten_joinpaths_tree t. contour_integral p
f)"
    using wf assms int by (subst contour_integral_joinpaths_list) auto
```

```
  finally show ?thesis
    using assms by (simp flip: rev_map)
qed
```

The following is an alternative way to split contour integrals that uses holomorphicity w.r.t. a user-defined region rather than analyticity on the path. This may sometimes be more convenient.

```
lemma contour_integral_via_joinpaths_holo:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "(∑q←rev (flatten_joinpaths_tree t). contour_integral q f)
= T ∧
           f holomorphic_on A ∧ open A ∧ path_image p ⊆ A ∧
           weak_wf_joinpaths_tree t ∧ valid_path_pathlist (flatten_joinpaths_tree
t)"
  shows   "contour_integral p f = T"
proof (rule contour_integral_via_joinpaths[of _ t], goal_cases)
  case 2
  from assms have "f holomorphic_on A" "open A" "path_image p ⊆ A"
    by blast+
  hence "f analytic_on path_image p"
    using analytic_on_holomorphic by blast
  with assms show ?case
    by blast
qed fact+


fun unions_list :: "('a ⇒ 'b set) ⇒ 'a list ⇒ 'b set" where
  "unions_list f [] = {}"
| "unions_list f [x] = f x"
| "unions_list f (x # xs) = f x ∪ unions_list f xs"

lemma unions_list_eq: "unions_list f xs = (⋃x∈set xs. f x)"
  by (induction f xs rule: unions_list.induct) auto

lemma path_image_via_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "wf_joinpaths_tree t ∧ unions_list path_image (flatten_joinpaths_tree
t) = T"
  shows   "path_image p = T"
proof -
  have "path_image p = path_image (eval_joinpaths_tree t)"
    using assms by (simp add: REIFY_JOINPATHS_TAG_def)
  also have "... = path_image (joinpaths_list (flatten_joinpaths_tree
t))"
    by (intro eq_paths_imp_path_image_eq joinpaths_flatten_joinpaths_tree)
(use assms in auto)
  also have "... = (⋃x∈set (flatten_joinpaths_tree t). path_image x)"
    by (subst path_image_joinpaths_list) (use assms in auto)
  also have "... = unions_list path_image (flatten_joinpaths_tree t)"
```

```
    by (rule unions_list_eq [symmetric])
  finally show ?thesis
    using assms by auto
qed

lemma path_image_subset_via_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "wf_joinpaths_tree t ∧ list_all (λp. path_image p ⊆ T) (flatten_joinpaths_tree
t)"
  shows   "path_image p ⊆ T"
proof -
  have "path_image p = path_image (eval_joinpaths_tree t)"
    using assms by (simp add: REIFY_JOINPATHS_TAG_def)
  also have "... = path_image (joinpaths_list (flatten_joinpaths_tree
t))"
    by (intro eq_paths_imp_path_image_eq joinpaths_flatten_joinpaths_tree)
(use assms in auto)
  also have "... = (⋃x∈set (flatten_joinpaths_tree t). path_image x)"
    by (subst path_image_joinpaths_list) (use assms in auto)
  finally show ?thesis
    using assms by (auto simp: list.pred_set)
qed

lemma not_in_path_image_via_joinpaths [path_automation_intros]:
  assumes "REIFY_JOINPATHS_TAG p = eval_joinpaths_tree t"
  assumes "wf_joinpaths_tree t ∧ list_all (λp. x ∉ path_image p) (flatten_joinpaths_tree
t)"
  shows   "x ∉ path_image p"
  using path_image_subset_via_joinpaths[of p t "-{x}"] assms by auto

lemma list_all_singleton_iff: "list_all P [x] ⟷ P x"
  by auto

lemmas [path_automation_simps] =
  flatten_joinpaths_tree.simps simple_joinpaths_list.simps weak_wf_joinpaths_tree.simps
  append.simps list.sel last.simps butlast.simps list.simps if_False if_True
refl
  arc_joinpaths_list_aux.simps arc_joinpaths_list_def ball_simps HOL.simp_thms
  start_joinpaths_tree.simps finish_joinpaths_tree.simps wf_joinpaths_tree.simps
  valid_path_joinpaths_tree.simps sublist_code prefix_code is_rotation_of_code
  is_rotation_of_aux.simps list.size add_Suc_right plus_nat.add_Suc add_0_right
add_0_left
  pathstart_linepath pathfinish_linepath pathstart_part_circlepath' pathfinish_part_circlepa
  pathstart_circlepath pathfinish_circlepath pathstart_rectpath pathfinish_rectpath
  path_linepath path_part_circlepath path_circlepath path_rectpath
  valid_path_linepath valid_path_part_circlepath valid_path_circlepath
valid_path_rectpath
  simple_path_part_circlepath simple_path_circlepath sum_list_Cons_rev
sum_list_singleton list.map
```

```
  valid_path_pathlist.simps rev.simps reversepath_linepath reversepath_part_circlepath
  reversepath_circlepath unions_list.simps path_image_linepath path_image_circlepath
  list.pred_inject(1) list_all_singleton_iff list.pred_inject(2)[of P
x "y # xs" for P x y xs]

lemma arc_linepath_iff [path_automation_simps]: "arc (linepath a b) ⟷
a ≠ b"
proof
  assume "arc (linepath a b)"
  thus "a ≠ b"
    by (smt (verit, best) arcD atLeastAtMost_iff linepath_0' linepath_1')
qed auto

lemma simple_path_linepath_iff [path_automation_simps]: "simple_path
(linepath a b) ⟷ a ≠ b"
proof
  assume "simple_path (linepath a b)"
  thus "a ≠ b"
    by (metis linepath_1' simple_path_subpath_eq subpath_refl)
qed auto

lemma arc_part_circlepath_iff [path_automation_simps]:
  "arc (part_circlepath x r a b) ⟷ r ≠ 0 ∧ a ≠ b ∧ |a - b| < 2 * pi"
proof (intro iffI conjI)
  assume *: "arc (part_circlepath x r a b)"
  show "r ≠ 0"
    using * by (auto simp: arc_linepath_iff)
  show "a ≠ b"
    using * by (auto simp: part_circlepath_empty arc_linepath_iff)
  show "|a - b| < 2 * pi"
  proof (rule ccontr)
    assume **: "¬|a - b| < 2 * pi"
    hence "a ≠ b"
      by auto
    have "part_circlepath x r a b (2 * pi / |a - b|) =
            x + rcis r ((1 - 2 * pi / |a - b|) * a + 2 * pi * b / |a -
b|)"
      by (simp add: part_circlepath_altdef linepath_def)
    also have "(1 - 2 * pi / |a - b|) * a + 2 * pi * b / |a - b| = a + sgn
(b - a) * 2 * pi"
      using ** ‹a ≠ b› by (auto simp: divide_simps) (auto simp: field_simps
abs_if split: if_splits)?
    also have "x + rcis r (a + sgn (b - a) * 2 * pi) = part_circlepath
x r a b 0"
      by (simp add: part_circlepath_altdef linepath_def rcis_def sgn_if
flip: cis_mult cis_cnj)
    finally have "part_circlepath x r a b (2 * pi / |a - b|) = part_circlepath
x r a b 0" .
    moreover have "0 ∈ {0..(1::real)}"
```

```
    by simp
  moreover have "2 * pi / |a - b| ∈ {0..1}"
    using ** ‹a ≠ b› by (auto simp: field_simps)
  ultimately show False
    using arcD[OF *, of 0 "2 * pi / |a - b|"] ‹a ≠ b› by fastforce
 qed
qed (auto intro!: arc_part_circlepath)

lemma arc_circlepath_iff [path_automation_simps]: "arc (circlepath x
r) ⟷ False"
 unfolding circlepath_def arc_part_circlepath_iff by auto



named_theorems path_automation_unfolds

ML ‹
signature PATH_REIFY = sig
  val do_path_reify_tac : Proof.context -> int -> tactic
  val path_reify_tac : Proof.context -> int -> tactic
  val tac : Proof.context -> int -> tactic
end


structure Path_Reify : PATH_REIFY = struct

val intros = named_theorems ‹path_automation_intros›
val simps = named_theorems ‹path_automation_simps›
val reifies = named_theorems ‹reify_joinpath_tree›
val unfolds = named_theorems ‹path_automation_unfolds›

fun do_path_reify_tac ctxt i =
  let
    val thms = Named_Theorems.get ctxt reifies
  in
    REPEAT (EqSubst.eqsubst_tac ctxt [0] thms i) THEN resolve_tac ctxt
@{thms HOL.refl} i
  end

local

fun tac {context = ctxt, concl, ...} =
    case Thm.term_of concl of
      const ‹Trueprop› $ (Const (const_name ‹HOL.eq›, _) $
         (Const (const_name ‹REIFY_JOINPATHS_TAG›, _) $ _) $ _) =>
        HEADGOAL (do_path_reify_tac ctxt)
    | _ => all_tac

in
```

```
val path_reify_tac = Subgoal.FOCUS_PARAMS tac

end


local

fun tac' {context = ctxt, ...} =
  let
    val intros = Named_Theorems.get ctxt intros
    val simps = Named_Theorems.get ctxt simps
    val unfolds = Named_Theorems.get ctxt unfolds
    val ctxt' = ctxt |> put_simpset HOL_basic_ss |> Simplifier.add_simps
simps
  in
    Local_Defs.unfold_tac ctxt unfolds
    THEN HEADGOAL (
      resolve_tac ctxt intros
      THEN_ALL_NEW DETERM o path_reify_tac ctxt
      THEN_ALL_NEW DETERM o Simplifier.simp_tac ctxt'
      THEN_ALL_NEW (TRY o REPEAT_ALL_NEW (DETERM o resolve_tac ctxt @{thms
conjI}))
    )
    THEN distinct_subgoals_tac
    THEN Local_Defs.fold_tac ctxt unfolds
  end

val sections =
  Method.sections
    [
     Args.add -- Args.colon >> K (Method.modifier (Named_Theorems.add
intros) here),
     Args.del -- Args.colon >> K (Method.modifier (Named_Theorems.del
intros) here),
     Args.$$$ "simp" -- Args.add -- Args.colon >> K (Method.modifier (Named_Theorems.add
simps) here),
     Args.$$$ "simp" -- Args.colon >> K (Method.modifier (Named_Theorems.add
simps) here),
     Args.$$$ "simp" -- Args.del -- Args.colon >> K (Method.modifier (Named_Theorems.del
simps) here),
     Args.$$$ "defs" -- Args.add -- Args.colon >> K (Method.modifier (Named_Theorems.add
unfolds) here),
     Args.$$$ "defs" -- Args.colon >> K (Method.modifier (Named_Theorems.add
unfolds) here),
     Args.$$$ "defs" -- Args.del -- Args.colon >> K (Method.modifier (Named_Theorems.del
unfolds) here)
    ]
```

```
in

val tac = Subgoal.FOCUS_PARAMS tac'
val method = sections >> K (SIMPLE_METHOD' o tac)

val _ =
  Theory.setup
   (Method.setup binding‹path_reify› method "reification of composite
paths into a path tree" #>
    Method.setup binding‹path› method
      "automation for various common path problems, e.g. path equivalence,
splitting integrals");

end

end
›
```

3.5 Examples

We now look at some concrete examples of how the method can be used.

experiment
begin

Showing well-formedness:

```
lemma "path (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1 + i)
0)"
  by path

lemma "valid_path (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1
+ i) 0)"
  by path
```

Showing that a path is simple:

```
lemma "arc (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1 + i) i)"
  apply path
        apply (auto simp: closed_segment_def complex_eq_iff)
  done

lemma "simple_path (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath
(1 + i) 0)"
  apply path
        apply (auto simp: closed_segment_def complex_eq_iff)
  done
```

Computing the image of a composite path:

```
lemma "path_image (linepath 0 (1::complex) +++ linepath 1 i) =
        closed_segment 0 1 ∪ closed_segment 1 i"
```

by *path*

Showing equivalence of paths modulo associativity and reversal:

lemma *"((linepath 0 1 +++ linepath 1 2) +++ linepath 2 3) +++ linepath 3 4* $\equiv_p$
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
by *path*

lemma *"linepath 0 1 +++ reversepath (linepath 3 2 +++ linepath 2 1) +++ linepath 3 4* $\equiv_p$
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
by *path*

Subpath relationships can also be shown in the same fashion.

lemma *"linepath 1 2 +++ linepath 2 3* $\leq_p$
linepath 0 1 +++ linepath 1 2 +++ linepath 2 3 +++ linepath 3 4"
by *path*

lemma *"linepath 0 1 +++ reversepath (linepath 3 2 +++ linepath 2 1)* $\leq_p$
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
by *path*

For loops, one can, in addition to reversal and associativity, also show equivalence modulo "rotation". Consider e.g. a counter-clockwise rectangular path and consider paths to be equal modular associativity. Then there are four different ways to write that path, corresponding to which corner we start in.

The automation can prove automatically that all of these four paths are equivalent to one another (basically by brute-forcing all 4 possibilities).

lemma *"linepath 0 1 +++ linepath 1 (1 +* i*) +++ linepath (1 +* i*)* i *+++ linepath* i *0* $\equiv_\circlearrowleft$
linepath 1 (1 + i*) +++ linepath (1 +* i*)* i *+++ linepath* i *0 +++ linepath 0 1"*
by *path*

For the next few examples, we define a path consisting of three perpendicular lines.

definition *g* **where** *"g = (linepath 0 1 +++ linepath 1 (1 +* i*) +++ linepath (1 +* i*)* i*)"*

Contour integrals on such composite paths can be split into integrals on the constituent paths. Since the path is often a large, unwieldy expression that is hidden behind a definition, one can give that definition theorem to the *path* method with the *defs* keyword. Its definition is then unfolded automatically and re-folded in any of the arising proof obligations that contain the full path again, such as the analyticity condition here.

```
lemma "contour_integral g (λx. x) = -1/2"
proof (path defs: g_def)
  show "contour_integral (linepath 0 1) (λx. x) +
        contour_integral (linepath 1 (1 + i)) (λx. x) +
        contour_integral (linepath (1 + i) i) (λx. x) = - 1 / 2"
    by (simp add: field_simps)
next
  show "(λx. x) analytic_on path_image g"
    by auto
qed
```

Alternatively, one can also show holomorphicity on some open superset of the path's image instead of analyticity on exactly the path's image.

```
lemma "contour_integral g (λx. x) = -1/2"
proof (path defs: g_def del: contour_integral_via_joinpaths add: contour_integral_via_joinp
  show "contour_integral (linepath 0 1) (λx. x) +
        contour_integral (linepath 1 (1 + i)) (λx. x) +
        contour_integral (linepath (1 + i) i) (λx. x) = - 1 / 2"
    by (simp add: field_simps)
next
  show "(λx. x) holomorphic_on UNIV"
    by auto
next
  show "open (UNIV :: complex set)"
    by simp
next
```

Conditions such as a path being a subset of some other set can also be simplified using the *path* method:

```
  show "path_image g ⊆ UNIV"
    apply (path defs: g_def)
      apply auto
    done
qed
```

Winding numbers can be split into the winding numbers of the constituent paths in the same way as integrals. However, for concrete paths it is probably better to use Wenda Li's automation rather than this.

```
lemma "winding_number g (1 / 3 + 1 / 3 * i) = undefined"
  apply (path defs: g_def)
     apply (simp_all add: closed_segment_same_Re closed_segment_same_Im)
  oops

end

end
```

References

[1] M. Raussen and U. Fahrenberg. Reparametrizations of continuous paths, 2007. URL: https://arxiv.org/abs/0706.3560, doi:10.48550/arXiv.0706.3560.

[2] A. A. Tuzhilin. Lectures on Hausdorff and Gromov–Hausdorff distance geometry, 2020. URL: https://arxiv.org/abs/2012.00756, doi:10.48550/arXiv.2012.00756.