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 . . . 4
- 1.4 General lemmas about real functions . . . 5
- 1.5 Rounding and fractional part . . . 6
- 1.6 General lemmas about paths . . . 8
- 1.7 Some facts about betweenness . . . 9
- 1.8 Simple loops and orientation . . . 10
- 1.9 More about circular arcs . . . 12
- 1.10 Reparametrisation of loops by shifting . . . 13

2 Some useful relations on paths **15**
- 2.1 Equivalence of paths up to reparametrisation . . . 16
- 2.2 Splitting lines and circular arcs . . . 22
- 2.3 The subpath relation . . . 24
- 2.4 Equivalence of closed paths . . . 28
- 2.5 Notation . . . 34
- 2.6 Examples . . . 34

3 Automation for paths **34**
- 3.1 Joining a list of paths together . . . 35
- 3.2 Representing a sequence of path joins as a tree . . . 39
- 3.3 Equivalence of two join trees . . . 43
- 3.4 Implementation . . . 43
- 3.5 Examples . . . 48

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"*
  ⟨*proof*⟩

lemma *dist_linepath1: "dist (linepath a b x) a = |x| * dist a b"*
⟨*proof*⟩

lemma *dist_linepath2: "dist (linepath a b x) b = |1 - x| * dist a b"*
⟨*proof*⟩

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*⟩

lemma *mono_on_compose:*
  assumes *"mono_on A f" "mono_on B g" "f ` A ⊆ B"*
  shows *"mono_on A (g ∘ f)"*
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *at_within_atLeastAtMost_eq_bot_iff_real:*
"at x within {a..b} = bot ⟷ x ∉ {a..b::real} ∨ a = b"
⟨*proof*⟩

lemma *eventually_in_pointed_at: "eventually (λx. x ∈ A - {y}) (at y within A)"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma (**in** *order_topology*) *at_within_Ico_at: "a < x ⟹ x < b ⟹ at x within {a..<b} = at x"*
⟨*proof*⟩

lemma (**in** *order_topology*) *at_within_Ioc_at: "a < x ⟹ x < b ⟹ at x within {a<..b} = at x"*
⟨*proof*⟩

lemma (**in** *order_topology*) *at_within_Ioo_at: "a < x ⟹ x < b ⟹ at x within {a<..<b} = at x"*
⟨*proof*⟩

lemma (**in** *order_topology*) *at_within_Icc_Ico:*
assumes *"a ≤ x" "x < b"*
shows *"at x within {a..b} = at x within {a..<b}"*
⟨*proof*⟩

lemma (**in** *order_topology*) *at_within_Icc_Ioc:*
assumes *"a < x" "x ≤ b"*
shows *"at x within {a..b} = at x within {a<..b}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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))"*
⟨*proof*⟩

1.5 Rounding and fractional part

lemma *frac_of_Int [simp]: "x ∈ ℤ ⟹ frac x = 0"*
⟨*proof*⟩

lemma *floor_less_not_int: "x ∉ ℤ ⟹ of_int (floor x) < x"*
⟨*proof*⟩

lemma *less_ceiling_not_int: "x ∉ ℤ ⟹ of_int (ceiling x) > x"*
⟨*proof*⟩

lemma *image_frac_atLeastLessThan:*
assumes *"y ≥ x + (1 :: 'a :: floor_ceiling)"*
shows *"frac ‘ {x..<y} = {0..<1}"*
⟨*proof*⟩

lemma *image_frac_atLeastAtMost:*
assumes *"y ≥ x + 1"*
shows *"frac ‘ {x..y} = {0..<1}"*
⟨*proof*⟩

lemma *tendsto_frac_real [tendsto_intros]:*
assumes *"(x :: real) ∉ ℤ"*
shows *"(frac ⟶ frac x) (at x within A)"*
⟨*proof*⟩

lemma *tendsto_frac_at_left_int_real:*
assumes *"(x :: real) ∈ ℤ"*
shows *"(frac ⟶ 1) (at_left x)"*
⟨*proof*⟩

lemma *filterlim_at_frac_at_left_int_real:*
assumes *"(x :: real) ∈ ℤ"*
shows *"filterlim frac (at_left 1) (at_left x)"*
⟨*proof*⟩

lemma *tendsto_frac_at_right_int_real:*
assumes *"(x :: real) ∈ ℤ"*
shows *"(frac ⟶ 0) (at_right x)"*
⟨*proof*⟩

lemma *filterlim_at_frac_at_right_int_real [tendsto_intros]:*
assumes *"(x :: real) ∈ ℤ"*
shows *"filterlim frac (at_right 0) (at_right x)"*
⟨*proof*⟩

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*⟩

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)))"*
⟨*proof*⟩

1.6 General lemmas about paths

lemma *linepath_scaleR: "(*$_R$) c ∘ linepath a b = linepath (c *$_R$ a) (c *$_R$ b)"*
⟨*proof*⟩

lemma *linepath_mult_complex: "(*) c ∘ linepath a b = linepath (c * a) (c * b :: complex)"*
⟨*proof*⟩

lemma *linepath_translate: "(+) c ∘ linepath a b = linepath (c + a) (c + b)"*
⟨*proof*⟩

lemma *part_circlepath_translate:*
"(+) c ∘ part_circlepath x r a b = part_circlepath (c + x) r a b"
⟨*proof*⟩

lemma *circlepath_translate:*
"(+) c ∘ circlepath x r = circlepath (c + x) r"
⟨*proof*⟩

lemma *rectpath_translate:*
"(+) c ∘ rectpath a b = rectpath (c + a) (c + b)"
⟨*proof*⟩

lemma *path_image_cong: "(⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ path_image p = path_image q"*
⟨*proof*⟩

lemma *path_cong: "(⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ path p = path q"*
⟨*proof*⟩

lemma *simple_path_cong:*
shows *"(⋀x. x ∈ {0..1} ⟹ f x = g x) ⟹ simple_path f ⟷ simple_path g"*
⟨*proof*⟩

lemma *simple_path_reversepath_iff: "simple_path (reversepath g) ⟷ simple_path g"*
⟨*proof*⟩

lemma *path_image_loop:*
assumes *"pathstart p = pathfinish p"*
shows *"path_image p = p ‘ {0..<1}"*
⟨*proof*⟩

lemma *simple_pathD:*
assumes *"simple_path p" "x ∈ {0..1}" "y ∈ {0..1}" "x ≠ y" "p x = p y"*
shows *"{x, y} = {0, 1}"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *between_trans1:*
assumes *"between (a, c) b" "between (b, d) c" "b ≠ c" "a ≠ d"*
shows *"between (a, d) b"*
⟨*proof*⟩

lemma *between_trans2:*
"between (a, c) b ⟹ between (b, d) c ⟹ b ≠ c ⟹ a ≠ d ⟹ between (a, d) c"
⟨*proof*⟩

lemma *between_trans1':*
assumes *"between (a :: 'a :: euclidean_space, c) b" "between (b, d) c" "b ≠ c"*
shows *"between (a, d) b"*

⟨*proof*⟩

lemma *between_trans2':*
assumes *"between (a :: 'a :: euclidean_space, c) b" "between (b, d) c" "b ≠ c"*
shows *"between (a, d) c"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *simple_loop_cwI:*
```
  "simple_loop p ⟹ z ∉ path_image p ⟹ winding_number p z = -1 ⟹
simple_loop_cw p"
```
⟨*proof*⟩

lemma *simple_path_not_cw_and_ccw:* "¬*simple_loop_cw p* ∨ ¬*simple_loop_ccw p*"
⟨*proof*⟩

lemma *simple_loop_cw_or_ccw:*
assumes *"simple_loop p"*
shows *"simple_loop_cw p ∨ simple_loop_ccw p"*
⟨*proof*⟩

lemma *simple_loop_ccw_conv_cw:*
assumes *"simple_loop p"*
shows *"simple_loop_ccw p ⟷ ¬simple_loop_cw p"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *simple_loop_orientation_eq_0_iff [simp]:*
```
  "simple_loop_orientation p = 0 ⟷ ¬simple_loop p"
```
⟨*proof*⟩

lemma *simple_loop_ccw_reversepath_aux:*
assumes *"simple_loop_ccw p"*
shows *"simple_loop_cw (reversepath p)"*
⟨*proof*⟩

lemma *simple_loop_cw_reversepath_aux:*

assumes *"simple_loop_cw p"*
shows *"simple_loop_ccw (reversepath p)"*
⟨*proof*⟩

lemma *simple_loop_cases: "simple_loop_ccw p ∨ simple_loop_cw p ∨ ¬simple_loop p"*
⟨*proof*⟩

lemma *simple_loop_cw_reversepath [simp]: "simple_loop_cw (reversepath p) ⟷ simple_loop_ccw p"*
⟨*proof*⟩

lemma *simple_loop_ccw_reversepath [simp]: "simple_loop_ccw (reversepath p) ⟷ simple_loop_cw p"*
⟨*proof*⟩

lemma *simple_loop_orientation_reversepath [simp]:*
"simple_loop_orientation (reversepath p) = -simple_loop_orientation p"
⟨*proof*⟩

lemma *simple_loop_orientation_cases:*
assumes *"simple_loop p"*
shows *"simple_loop_orientation p ∈ {-1, 1}"*
⟨*proof*⟩

lemma *inside_simple_loop_iff:*
assumes *"simple_loop p"*
shows *"z ∈ inside (path_image p) ⟷ z ∉ path_image p ∧ winding_number p z ≠ 0"*
⟨*proof*⟩

lemma *outside_simple_loop_iff:*
assumes *"simple_loop p"*
shows *"z ∈ outside (path_image p) ⟷ z ∉ path_image p ∧ winding_number p z = 0"*
⟨*proof*⟩

1.9 More about circular arcs

lemma *part_circlepath_altdef:*
"part_circlepath z r a b = (λt. z + rcis r (linepath a b t))"
⟨*proof*⟩

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'"*
⟨*proof*⟩

lemma *part_circlepath_empty: "part_circlepath x r a a = linepath (x + rcis r a) (x + rcis r a)"*
⟨*proof*⟩

lemma *part_circlepath_radius_0 [simp]: "part_circlepath x 0 a b = linepath x x"*
⟨*proof*⟩

lemma *part_circlepath_scaleR:*
"($_R$*) c ∘ part_circlepath x r a b = part_circlepath (c *$_R$ *x) (c * r) a b"*
⟨*proof*⟩

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*⟩

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'"*
⟨*proof*⟩

lemma *circlepath_altdef: "circlepath x r t = x + rcis r (2 * pi * t)"*
⟨*proof*⟩

lemma *reversepath_circlepath: "reversepath (circlepath x r) = part_circlepath x r (2 * pi) 0"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *shiftpath_circlepath:*
*"shiftpath c (circlepath x r) = part_circlepath x r (c * 2 * pi) ((c + 1) * 2 * pi)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *path_image_shiftpath':*
assumes *"path p" "pathstart p = pathfinish p"*
shows *"path_image (shiftpath' c p) = path_image p"*
⟨*proof*⟩

lemma *path_shiftpath_0_iff [simp]: "path (shiftpath 0 p) ⟷ path p"*
⟨*proof*⟩

lemma *path_shiftpath'_int_iff [simp]:*
assumes *"pathstart p = pathfinish p" "c ∈ ℤ"*
shows *"path (shiftpath' c p) ⟷ path p"*
⟨*proof*⟩

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*⟩

lemma *shiftpath'_frac: "shiftpath' (frac c) p = shiftpath' c p"*
⟨*proof*⟩

lemma *path_shiftpath' [intro]:*
"pathstart p = pathfinish p ⟹ path p ⟹ path (shiftpath' c p)"
⟨*proof*⟩

lemma *pathfinish_shiftpath':*
"pathfinish (shiftpath' c p) = pathstart (shiftpath' c p)"
⟨*proof*⟩

lemma *shiftpath'_shiftpath': "shiftpath' c (shiftpath' d p) = shiftpath' (c + d) p"*
⟨*proof*⟩

lemma *simple_path_shiftpath':*
assumes *"simple_path p" "pathfinish p = pathstart p"*
shows *"simple_path (shiftpath' c p)"*
⟨*proof*⟩

lemma *simple_path_shiftpath'_iff [simp]:*
assumes *"pathfinish p = pathstart p"*
shows *"simple_path (shiftpath' c p) ⟷ simple_path p"*
⟨*proof*⟩

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*⟩

lemma *homotopic_loops_shiftpath'_right:*
assumes *"path p" "path_image p ⊆ A" "pathstart p = pathfinish p"*
shows *"homotopic_loops A p (shiftpath' x p)"*
⟨*proof*⟩

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*⟩

lemma *shiftpath'_circlepath:*
*"shiftpath' c (circlepath x r) = part_circlepath x r (c * 2 * pi) ((c + 1) * 2 * pi)"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

lemma *inj': "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x = f y ⟷ x = y"*
⟨*proof*⟩

lemma *less_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x < f y ⟷ x < y"*
⟨*proof*⟩

lemma *le_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x ≤ f y ⟷ x ≤ y"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *f_ge_0 [simp, intro]: "x ∈ {0..1} ⟹ f x ≥ 0"*
and *f_le_1 [simp, intro]: "x ∈ {0..1} ⟹ f x ≤ 1"*
⟨*proof*⟩

lemma *f_gt_0 [simp, intro]: "x ∈ {0<..1} ⟹ f x > 0"*
and *f_less_1 [simp, intro]: "x ∈ {0..<1} ⟹ f x < 1"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *bij_betw: "bij_betw f {0..1} {0..1}"*
⟨*proof*⟩

lemma *same_ends: "pathstart p = pathstart q" "pathfinish p = pathfinish q"*
⟨*proof*⟩

lemma *path_image_eq: "path_image p = path_image q"*
⟨*proof*⟩

lemma *inverse: "eq_paths_locale q p (inv_into {0..1} f)"*
⟨*proof*⟩

lemma *reverse: "eq_paths_locale (reversepath p) (reversepath q) (λx. 1 - f (1 - x))"*
⟨*proof*⟩

lemma *homotopic:*
assumes *"path_image p ⊆ A"*
shows *"homotopic_paths A p q"*
⟨*proof*⟩

lemma *arc_iff: "arc p ⟷ arc q"*
⟨*proof*⟩

lemma *simple_path:*
  assumes *"simple_path p"*
  shows *"simple_path q"*
⟨*proof*⟩

lemma *simple_path_iff: "simple_path p ⟷ simple_path q"*
⟨*proof*⟩

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*⟩

end

lemma *eq_paths_locale_refl [intro!]: "path p ⟹ eq_paths_locale p p (λx. x)"*
  ⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

sublocale *p12: eq_paths_locale "p1 +++ p2" "q1 +++ q2" f12*

⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

lemma *eq_paths_refl [simp, intro!, eq_paths_intros]: "path p ⟹ eq_paths p p"*
  ⟨*proof*⟩

lemma *eq_paths_refl'': "path p ⟹ p = q ⟹ eq_paths p q"*
  ⟨*proof*⟩

lemma *eq_paths_refl':*
  "path p ∨ path q ⟹ (⋀x. x ∈ {0..1} ⟹ p x = q x) ⟹ eq_paths p q"
  ⟨*proof*⟩

lemma *eq_paths_sym:*
  "eq_paths p q ⟹ eq_paths q p"

⟨*proof*⟩

lemma *eq_paths_sym_iff:*
"eq_paths p q ⟷ eq_paths q p"
⟨*proof*⟩

lemma *eq_paths_reverse [intro, eq_paths_intros]:*
"eq_paths p q ⟹ eq_paths (reversepath p) (reversepath q)"
⟨*proof*⟩

lemma *eq_paths_reverse_iff:*
"eq_paths (reversepath p) (reversepath q) ⟷ eq_paths p q"
⟨*proof*⟩

lemma *eq_paths_trans [trans]:*
assumes *"eq_paths p q" "eq_paths q r"*
shows *"eq_paths p r"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *eq_paths_shiftpath_0 [intro, eq_paths_intros]: "path p ⟹ eq_paths (shiftpath 0 p) p"*
⟨*proof*⟩

lemma *eq_paths_shiftpath_0_iff [simp]: "eq_paths (shiftpath 0 p) q ⟷ eq_paths p q"*
⟨*proof*⟩

lemma *eq_paths_shiftpath_0_iff' [simp]: "eq_paths q (shiftpath 0 p) ⟷ eq_paths q p"*
⟨*proof*⟩

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*⟩

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*⟩

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"*

⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *eq_paths_imp_same_ends:*
"eq_paths p q ⟹ pathstart p = pathstart q"
"eq_paths p q ⟹ pathfinish p = pathfinish q"
⟨*proof*⟩

lemma *eq_paths_imp_path_image_eq:*
"eq_paths p q ⟹ path_image p = path_image q"
⟨*proof*⟩

lemma *eq_paths_imp_homotopic:*
assumes *"eq_paths p q" "path_image p ∩ path_image q ⊆ A"*
shows *"homotopic_paths A p q"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *eq_paths_imp_arc_iff:*
"eq_paths p q ⟹ arc p ⟷ arc q"
⟨*proof*⟩

lemma *eq_paths_arc_trans [trans]:*
"eq_paths p q ⟹ arc q ⟹ arc p"
"arc p ⟹ eq_paths p q ⟹ arc q"
⟨*proof*⟩

lemma *eq_paths_imp_simple_path_iff:*
"eq_paths p q ⟹ simple_path p ⟷ simple_path q"
⟨*proof*⟩

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"
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

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)"*
  ⟨*proof*⟩

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}"*
  ⟨*proof*⟩

lemma *inj': "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x = f y ⟷ x = y"*
  ⟨*proof*⟩

lemma *less_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x < f y ⟷ x < y"*
  ⟨*proof*⟩

lemma *le_iff: "x ∈ {0..1} ⟹ y ∈ {0..1} ⟹ f x ≤ f y ⟷ x ≤ y"*
  ⟨*proof*⟩

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"*

⟨*proof*⟩

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"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

lemma *ge_0 [simp]: "x ∈ {0..1} ⟹ f x ≥ 0"*
  and *le_1 [simp]: "x ∈ {0..1} ⟹ f x ≤ 1"*
  ⟨*proof*⟩

lemma *bij_betw: "bij_betw f {0..1} {f 0..f 1}"*
⟨*proof*⟩

lemma *in_range: "f x ∈ {0..1}"* **if** *"x ∈ {0..1}"*
  ⟨*proof*⟩

lemma *path_image_subset: "path_image p ⊆ path_image q"*
⟨*proof*⟩

lemma *reverse: "subpath_locale (reversepath p) (reversepath q) (λx. 1 - f (1 - x))"*
⟨*proof*⟩

lemma *arc:*
  assumes *"arc q"*
  shows *"arc p"*
  ⟨*proof*⟩

lemma *arc':*
  assumes *"simple_path q" "f 0 ≠ 0 ∨ f 1 ≠ 1"*
  shows *"arc p"*
  ⟨*proof*⟩

lemma *simple_path:*
  assumes *"simple_path q"*
  shows *"simple_path p"*
⟨*proof*⟩

end

context *eq_paths_locale*
begin

sublocale *subpath: subpath_locale q p f*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *is_subpath_refl [intro!]: "path p ⟹ is_subpath p p"*
⟨*proof*⟩

lemma *eq_paths_imp_subpath [intro]:*
assumes *"eq_paths p q"*
shows *"is_subpath p q"*
⟨*proof*⟩

lemma *is_subpath_reverse [intro]:*
"is_subpath p q ⟹ is_subpath (reversepath p) (reversepath q)"
⟨*proof*⟩

lemma *is_subpath_reverse_iff:*
"is_subpath (reversepath p) (reversepath q) ⟷ is_subpath p q"
⟨*proof*⟩

lemma *is_subpath_trans [trans]:*
assumes *"is_subpath p q" "is_subpath q r"*
shows *"is_subpath p r"*
⟨*proof*⟩

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"

⟨*proof*⟩

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"
⟨*proof*⟩

lemma *is_subpath_imp_path_image_subset:*
"is_subpath p q ⟹ path_image p ⊆ path_image q"
⟨*proof*⟩

lemma *subpath_imp_arc:*
"is_subpath p q ⟹ arc q ⟹ arc p"
⟨*proof*⟩

lemma *subpath_imp_simple_path:*
"is_subpath p q ⟹ simple_path q ⟹ simple_path p"
⟨*proof*⟩

lemma *is_subpath_joinI1 [intro]:*
assumes *[simp]: "path p" "path q" "pathfinish p = pathstart q"*
shows *"is_subpath p (p +++ q)"*
⟨*proof*⟩

lemma *is_subpath_joinI2 [intro]:*
assumes *[simp]: "path p" "path q"* **and** *"pathfinish p = pathstart q"*
shows *"is_subpath q (p +++ q)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *eq_paths_join_subpaths':*
assumes *"path p" "0 < b" "b < 1"*
shows *"eq_paths p (subpath 0 b p +++ subpath b 1 p)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *not_degenerate': "a ≠ d"*
⟨*proof*⟩

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"
⟨*proof*⟩

sublocale *subpath_locale "linepath b c" "linepath a d" f*
⟨*proof*⟩

end

lemma *is_subpath_linepath:*
assumes *"betweens [a, b, c, d]" "b ≠ c"*
shows *"is_subpath (linepath b c) (linepath a d)"*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *subpath_locale "part_circlepath x r b c" "part_circlepath x r a d" angle.f*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *eq_loops_refl [simp, intro, eq_paths_intros]:*
assumes *[simp]: "path p" "pathstart p = pathfinish p"*
shows *"eq_loops p p"*
⟨*proof*⟩

lemma *eq_loops_imp_loop:*
assumes *"eq_loops p q"*
shows *"pathstart p = pathfinish p" "pathstart q = pathfinish q"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_left:*
assumes *"path p" "pathstart p = pathfinish p"*
shows *"eq_loops (shiftpath' c p) p"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_right:*
assumes *"path p" "pathstart p = pathfinish p"*
shows *"eq_loops p (shiftpath' c p)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

end

lemma *eq_paths_shiftpath_shiftpath':*
  "path p ⟹ pathstart p = pathfinish p ⟹ c ∈ {0..1} ⟹
     eq_paths (shiftpath c p) (shiftpath' c p)"
  ⟨*proof*⟩

lemma *eq_loops_imp_path_image_eq:*
  assumes *"eq_loops p q"*
  shows *"path_image p = path_image q"*
⟨*proof*⟩

lemma *eq_loops_imp_simple_path_iff:*
  assumes *"eq_loops p q"*
  shows *"simple_path p ⟷ simple_path q"*
⟨*proof*⟩

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"

⟨*proof*⟩

lemma *eq_loops_imp_simple_loop_iff:*
assumes *"eq_loops p q"*
shows *"simple_loop p ⟷ simple_loop q"*
⟨*proof*⟩

lemma *eq_loops_imp_homotopic:*
assumes *"eq_loops p q" "path_image p ∩ path_image q ⊆ A"*
shows *"homotopic_loops A p q"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *eq_loops_imp_same_orientation:*
assumes *"eq_loops p q"*
shows *"simple_loop_orientation p = simple_loop_orientation q"*
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *eq_loops_trans [trans]:*
assumes *"eq_loops p q" "eq_loops q r"*
shows *"eq_loops p r"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *eq_loops_sym:*
assumes *"eq_loops p q"*
shows *"eq_loops q p"*
⟨*proof*⟩

lemma *eq_loops_sym_iff: "eq_loops p q ⟷ eq_loops q p"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_leftI:*
assumes *"eq_loops p q"*
shows *"eq_loops (shiftpath' c p) q"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_rightI:*
assumes *"eq_loops q p"*
shows *"eq_loops q (shiftpath' c p)"*
⟨*proof*⟩

lemma *path_shiftpath'_iff [simp]:*
assumes *"pathstart p = pathfinish p"*
shows *"path (shiftpath' c p) ⟷ path p"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_left_iff [simp]:*
assumes *"pathstart p = pathfinish p"*
shows *"eq_loops (shiftpath' c p) q ⟷ eq_loops p q"*
⟨*proof*⟩

lemma *eq_loops_shiftpath'_right_iff [simp]:*
assumes *"pathstart p = pathfinish p"*
shows *"eq_loops q (shiftpath' c p) ⟷ eq_loops q p"*
⟨*proof*⟩

lemma *eq_loops_shiftpath_shiftpath':*
assumes *"pathstart p = pathfinish p" "path p" "c ∈ {0..1}"*
shows *"eq_loops (shiftpath c p) (shiftpath' c p)"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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"
⟨*proof*⟩

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')"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *eq_loops_full_part_circlepath:*
  assumes *"b = a + 2 * pi"*
  shows *"eq_loops (part_circlepath x r a b) (circlepath x r)"*
  ⟨*proof*⟩

lemma *eqloops_reversepath_cong:*
  assumes *"eq_loops p q"*
  shows *"eq_loops (reversepath p) (reversepath q)"*
⟨*proof*⟩

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** *"≡$_○$" 60*)
notation *is_subpath* (**infix** *"≤$_p$" 60*)

end

unbundle *path_rel_notation*

2.6 Examples

lemma *"linepath 0 1 +++ linepath 1 (3::complex) ≡$_p$ linepath 0 3"*
  ⟨*proof*⟩

lemma *"linepath 0 1 +++ linepath 1 (3::complex) ≡$_p$ linepath 0 3"*
  ⟨*proof*⟩

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"
  ⟨proof⟩

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)"
```

⟨*proof*⟩

lemma *wf_pathlist_Cons:*

```
  "wf_pathlist (p # ps) ⟷ path p ∧ (ps = [] ∨ pathfinish p = pathstart (hd ps) ∧ wf_pathlist ps)"
```

⟨*proof*⟩

lemma *weak_wf_pathlist_Cons:*

```
  "weak_wf_pathlist (p # ps) ⟷ (ps = [] ∨ pathfinish p = pathstart (hd ps) ∧ weak_wf_pathlist ps)"
```

⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *valid_path_pathlist_altdef: "valid_path_pathlist xs ⟷ xs ≠ [] ∧ list_all valid_path xs"*
⟨*proof*⟩

lemma *valid_path_weak_wf_pathlist_imp_wf:*
"valid_path_pathlist ps ⟹ weak_wf_pathlist ps ⟹ wf_pathlist ps"
⟨*proof*⟩

lemma *wf_pathlist_append:*
assumes *"ps ≠ []" "qs ≠ []"*
shows *"wf_pathlist (ps @ qs) ⟷*
wf_pathlist ps ∧ wf_pathlist qs ∧ pathfinish (last ps) = pathstart (hd qs)"
⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *pathstart_joinpaths_list [simp]:*
"xs ≠ [] ⟹ pathstart (joinpaths_list xs) = pathstart (hd xs)"
⟨*proof*⟩

lemma *pathfinish_joinpaths_list [simp]:*
"xs ≠ [] ⟹ pathfinish (joinpaths_list xs) = pathfinish (last xs)"
⟨*proof*⟩

lemma *path_joinpaths_list [simp, intro]: "wf_pathlist xs ⟹ path (joinpaths_list xs)"*
⟨*proof*⟩

lemma *valid_path_joinpaths_list [intro]:*
"valid_path_pathlist xs ⟹ weak_wf_pathlist xs ⟹ valid_path (joinpaths_list xs)"
⟨*proof*⟩

lemma *path_image_joinpaths_list:*
assumes *"wf_pathlist ps"*
shows *"path_image (joinpaths_list ps) = (⋃p∈set ps. path_image p)"*
⟨*proof*⟩

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*⟩

lemma *arc_joinpaths_list_weak_wf_imp_wf:*
assumes *"weak_wf_pathlist xs" "arc_joinpaths_list xs"*
shows *"wf_pathlist xs"*
⟨*proof*⟩

lemma *arc_joinpaths_aux:*
assumes *"wf_pathlist xs" "arc_joinpaths_list_aux xs" "∀x∈set xs. arc x"*
shows *"arc (joinpaths_list xs)"*
⟨*proof*⟩

lemma *arc_joinpaths_list [intro?]:*
assumes *"weak_wf_pathlist xs" "arc_joinpaths_list xs"*
shows *"arc (joinpaths_list xs)"*
⟨*proof*⟩

lemma *simple_joinpaths_list_weak_wf_imp_wf:*
assumes *"weak_wf_pathlist xs" "simple_joinpaths_list xs"*
shows *"wf_pathlist xs"*
⟨*proof*⟩

lemma *simple_path_joinpaths_list [intro?]:*
assumes *"weak_wf_pathlist xs" "simple_joinpaths_list xs"*
shows *"simple_path (joinpaths_list xs)"*
⟨*proof*⟩

lemma *wf_pathlist_sublist:*
assumes *"wf_pathlist ys" "sublist xs ys" "xs ≠ []"*

shows *"wf_pathlist xs"*
⟨*proof*⟩

lemma *is_subpath_joinpaths_list_append_right:*
assumes *"wf_pathlist (xs @ ys)" "xs ≠ []"*
shows *"is_subpath (joinpaths_list xs) (joinpaths_list (xs @ ys))"*
⟨*proof*⟩

lemma *is_subpath_joinpaths_list_append_left:*
assumes *"wf_pathlist (xs @ ys)" "ys ≠ []"*
shows *"is_subpath (joinpaths_list ys) (joinpaths_list (xs @ ys))"*
⟨*proof*⟩

lemma *is_subpath_joinpaths_list:*
assumes *"wf_pathlist ys" "sublist xs ys" "xs ≠ []"*
shows *"is_subpath (joinpaths_list xs) (joinpaths_list ys)"*
⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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 ≠ []"
  ⟨proof⟩



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"
```

⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *wf_pathlist_map_rev [simp]: "wf_pathlist (map reversepath xs) ⟷ wf_pathlist (rev xs)"*
⟨*proof*⟩

lemma *weak_wf_pathlist_map_rev' [simp]: "weak_wf_pathlist (map reversepath xs) ⟷ weak_wf_pathlist (rev xs)"*
⟨*proof*⟩

lemma *weak_wf_pathlist_map_rev [simp]: "weak_wf_pathlist (rev (map reversepath xs)) ⟷ weak_wf_pathlist xs"*
⟨*proof*⟩

lemma *wf_pathlist_map_rev' [simp]: "wf_pathlist (rev (map reversepath xs)) ⟷ wf_pathlist xs"*
⟨*proof*⟩

lemma *wf_pathlist_flatten_pathree [simp]: "wf_pathlist (flatten_joinpaths_tree t) ⟷ wf_joinpaths_tree t"*
⟨*proof*⟩

lemma *weak_wf_pathlist_flatten_pathree [simp]:*
"weak_wf_pathlist (flatten_joinpaths_tree t) ⟷ weak_wf_joinpaths_tree t"
⟨*proof*⟩

lemma *reversepath_joinpaths_list:*
assumes *"wf_pathlist xs"*
shows *"reversepath (joinpaths_list xs) ≡$_p$ joinpaths_list (rev (map reversepath xs))"*
⟨*proof*⟩

lemma *joinpaths_flatten_joinpaths_tree:*
assumes *"wf_joinpaths_tree t"*
shows *"eval_joinpaths_tree t ≡$_p$ joinpaths_list (flatten_joinpaths_tree t)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *path_image_eval_joinpaths_tree:*
"wf_joinpaths_tree t ⟹
path_image (eval_joinpaths_tree t) = (⋃p∈paths_joinpaths_tree t. path_image p)"
⟨*proof*⟩

lemma *arc_joinpaths_tree [intro?]:*
"wf_joinpaths_tree t ⟹ arc_joinpaths_tree t ⟹ arc (eval_joinpaths_tree t)"
⟨*proof*⟩

lemma *simple_joinpaths_tree [intro?]:*
"wf_joinpaths_tree t ⟹ simple_joinpaths_tree t ⟹ simple_path (eval_joinpaths_tree t)"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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)"
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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"*
  ⟨*proof*⟩

lemma *sum_list_singleton: "sum_list [x] = x"*
  ⟨*proof*⟩

lemma *sum_list_Cons_rev: "sum_list (x # y # xs) = sum_list (y # xs) +*

```
(x :: 'a :: comm_monoid_add)"
```
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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*⟩

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)"*
  ⟨*proof*⟩

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*⟩

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*⟩

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"*
  ⟨*proof*⟩

lemma *list_all_singleton_iff: "list_all P [x] ⟷ P x"*
  ⟨*proof*⟩

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*⟩

lemma *simple_path_linepath_iff [path_automation_simps]: "simple_path (linepath a b) ⟷ a ≠ b"*
⟨*proof*⟩

lemma *arc_part_circlepath_iff [path_automation_simps]:*
*"arc (part_circlepath x r a b) ⟷ r ≠ 0 ∧ a ≠ b ∧ |a - b| < 2 * pi"*
⟨*proof*⟩

lemma *arc_circlepath_iff [path_automation_simps]: "arc (circlepath x r) ⟷ False"*
⟨*proof*⟩

named_theorems *path_automation_unfolds*

⟨*ML*⟩

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)"*
⟨*proof*⟩

lemma *"valid_path (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1 + i) 0)"*
⟨*proof*⟩

Showing that a path is simple:

lemma *"arc (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1 + i) i)"*
⟨*proof*⟩

lemma *"simple_path (linepath 0 1 +++ linepath 1 (1 + i) +++ linepath (1 + i) 0)"*

⟨*proof*⟩

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*"*
⟨*proof*⟩

Showing equivalence of paths modulo associativity and reversal:

lemma *"((linepath 0 1 +++ linepath 1 2) +++ linepath 2 3) +++ linepath 3 4 ≡$_p$*
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
⟨*proof*⟩

lemma *"linepath 0 1 +++ reversepath (linepath 3 2 +++ linepath 2 1) +++ linepath 3 4 ≡$_p$*
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
⟨*proof*⟩

Subpath relationships can also be shown in the same fashion.

lemma *"linepath 1 2 +++ linepath 2 3 ≤$_p$*
linepath 0 1 +++ linepath 1 2 +++ linepath 2 3 +++ linepath 3 4"
⟨*proof*⟩

lemma *"linepath 0 1 +++ reversepath (linepath 3 2 +++ linepath 2 1) ≤$_p$*
linepath 0 1 +++ (linepath 1 2 +++ (linepath 2 3 +++ linepath 3 4))"
⟨*proof*⟩

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 ≡$_\circ$*
linepath 1 (1 + i*) +++ linepath (1 +* i*)* i *+++ linepath* i *0 +++ linepath 0 1"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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.