Untitled

```
namespace OperatorMath

/-- syntax --/
inductive Trace : Type
| void : Trace
| delta : Trace → Trace
| integrate : Trace → Trace
| merge : Trace → Trace → Trace
deriving DecidableEq
open Trace

/-- routine encodings ; unchanged --/
@[simp] def size : Trace → Nat
| void => 0
| delta t => size t + 1
| integrate t => size t + 1
| merge a b => size a + size b + 1

@[simp] def encode : Trace → Nat
| void => 0
| delta t => 1 + encode t
| integrate t => 2 + encode t
| merge a b => 3 + encode a + encode b

@[simp] def collapse : Nat → Trace
| 0 => void
| n + 1 => delta (collapse n)

/-- elementary Boolean utilities --/
def isVoid : Trace → Bool
| void => true
| _ => false

/-- purely Boolean equality --/
def beq : Trace → Trace → Bool
| void, void => true
| delta a, delta b => beq a b
| integrate a, integrate b => beq a b
| merge a₁ a₂, merge b₁ b₂ => beq a₁ b₁ && beq a₂ b₂
| _, _ => false

@[simp] theorem beq_refl : ∀ t : Trace, beq t t = true
| void => rfl
| delta t => by simp [beq, beq_refl t]
| integrate t => by simp [beq, beq_refl t]
| merge a b => by simp [beq, beq_refl a, beq_refl b]

/-- Booleanised merging --/
def mergeSimp (na nb : Trace) : Trace :=
if isVoid na then nb
else if isVoid nb then na
else if beq na nb then na
else merge na nb

/-- normal form --/
def normalize : Trace → Trace
| void => void
| delta t => delta (normalize t)
| integrate t => integrate (normalize t)
| merge a b => mergeSimp (normalize a) (normalize b)

/-- idempotence remains a Prop proof (we use it once) --/
theorem normalize_idempotent :
∀ t : Trace, normalize (normalize t) = normalize t
| void => rfl
| delta t => by simp [normalize, normalize_idempotent t]
| integrate t => by simp [normalize, normalize_idempotent t]
| merge a b => by
let na : Trace := normalize a
let nb : Trace := normalize b
have ea : normalize na = na := by
simpa [na] using normalize_idempotent a
have eb : normalize nb = nb := by
simpa [nb] using normalize_idempotent b
by_cases h₁ : na = void
· simp [normalize, mergeSimp, isVoid, beq, na, nb, h₁, ea, eb]
· by_cases h₂ : nb = void
· simp [normalize, mergeSimp, isVoid, beq, na, nb, h₁, h₂, ea, eb]
· by_cases h₃ : beq na nb = true
· simp [normalize, mergeSimp, isVoid, beq, na, nb, h₁, h₂, h₃,
ea, eb, beq_refl] at *
· simp [normalize, mergeSimp, isVoid, beq, na, nb, h₁, h₂, h₃,
ea, eb] at *

/-- Boolean equivalence --/
def EquivB (x y : Trace) : Bool := beq (normalize x) (normalize y)

@[simp] theorem EquivB_refl (t : Trace) : EquivB t t = true := by
simp [EquivB, beq_refl]

/-- Boolean algebra --/
def negation : Trace → Trace
| void => void
| delta t => integrate (negation t)
| integrate t => delta (negation t)
| merge a b => merge (negation a) (negation b)

theorem negation_involution (t : Trace) :
negation (negation t) = t := by
induction t with
| void => simp [negation]
| delta t ih => simp [negation, ih]
| integrate t ih => simp [negation, ih]
| merge a b iha ihb => simp [negation, iha, ihb]

/-- connectives --/
def T : Trace := void
def tAnd (p q : Trace) := merge p q
def tOr (p q : Trace) := negation (merge (negation p) (negation q))
def tImp (p q : Trace) := tOr (negation p) q
def tIff (p q : Trace) := merge (tImp p q) (tImp q p)

/-- tAnd proofs in Bool --/
theorem tAnd_idem (p : Trace) :
EquivB (tAnd p p) p = true := by
dsimp [EquivB, tAnd]
have h : mergeSimp (normalize p) (normalize p) = normalize p := by
cases hp : normalize p with
| void => simp [mergeSimp, isVoid, beq, hp]
| delta t => simp [mergeSimp, isVoid, beq, hp]
| integrate t => simp [mergeSimp, isVoid, beq, hp]
| merge a b => simp [mergeSimp, isVoid, beq, hp]
simp [normalize, h, beq_refl]

/-- tAnd : ⊤ is left-unit --/
theorem tAnd_unit_left (p : Trace) :
EquivB (tAnd ⊤ p) p = true := by
-- ⊤ ∧ p → merge void p
-- that normalises to the same thing as p
dsimp [EquivB, tAnd, ⊤]
simp [normalize, -- unfold the single call to `normalize`
mergeSimp, -- …which immediately hits `mergeSimp`
isVoid, beq, -- tell `simp` how the "void" branch works
beq_refl] -- and use the reflexivity lemma
-- goal is now beq (normalize p) (normalize p) = true
-- which `simp` finishes with `beq_refl`.

theorem tAnd_unit_right (p : Trace) :
EquivB (tAnd p ⊤) p = true := by
dsimp [EquivB, tAnd, ⊤]
cases hp : normalize p with
| void => simp [normalize, mergeSimp, isVoid, beq, hp, beq_refl]
| delta t => simp [normalize, mergeSimp, isVoid, beq, hp, beq_refl]
| integrate t => simp [normalize, mergeSimp, isVoid, beq, hp, beq_refl]
| merge a b => simp [normalize, mergeSimp, isVoid, beq, hp, beq_refl]

/-- fixed-point infrastructure --/
structure FixpointWitness (F : Trace → Trace) : Type where
ψ : Trace
fixed : EquivB ψ (F ψ) = true

def mkFixed (F) (ψ) (h : EquivB ψ (F ψ) = true) :
FixpointWitness F := ⟨ψ, h⟩

/-- Gödel-style Φ --/
def Φ (_ : Trace) : Trace := integrate (delta void)

@[simp] theorem Φ_idempotent (t) :
EquivB (Φ t) (Φ (Φ t)) = true := by
dsimp [EquivB, Φ]
simp [normalize, beq_refl]

/-- choose G = Φ void --/
def G_witness : FixpointWitness Φ := by
refine mkFixed Φ (Φ void) ?_
dsimp [EquivB, Φ]
simp [normalize, beq_refl]
```

```
def G : Trace := G_witness.ψ

theorem godel_fixed_point :
EquivB G (Φ G) = true := G_witness.fixed

/-- Gödel sentence is **not** provable --/
theorem godel_not_provable :
EquivB G void = false := by
-- First expose what `G` really is
dsimp [EquivB, G] -- `G` is `G_witness.ψ`
change beq (normalize (Φ void)) void = false
-- Now it’s just beq (integrate (delta void)) void
simp [Φ, -- unfold Φ ( = integrate (delta void) )
normalize, -- one step through the normaliser
beq] -- and the Boolean equality definition

end OperatorMath
```

Suggested Fixes to Basic (ignore if you have better solution)

```
/- ------------------------------------------------------------------
Compatibility bridge: Prop-level equivalence & the old `same` -
-----------------------------------------------------------------/

/-- Boolean equality that we actually compute with. -/
def beq : Trace → Trace → Bool
| void, void => true
| delta a, delta b => beq a b
| integrate a, integrate b => beq a b
| merge a₁ a₂, merge b₁ b₂ => beq a₁ b₁ && beq a₂ b₂
| _, _ => false

@[simp] theorem beq_refl (t : Trace) : beq t t = true := by
induction t <;> simp [beq, *]

/-- **Old name** kept for code that still calls `same`. -/
abbrev same := beq

@[simp] theorem same_true_iff {a b : Trace} :
same a b = true ↔ a = b := by
induction a generalizing b <;> cases b <;>
simp [same, beq] at * -- all cases close by reflexivity

/-- Prop-level equivalence used by the older files. -/
def Equiv (x y : Trace) : Prop :=
normalize x = normalize y

/-- Boolean version that new code can keep using. -/
def EquivB (x y : Trace) : Bool :=
beq (normalize x) (normalize y)

@[simp] theorem EquivB_true_of_Equiv {x y : Trace} :
Equiv x y → EquivB x y = true := by
intro h; unfold EquivB; simp [h, beq_refl]

@[simp] theorem Equiv_of_EquivB_true {x y : Trace} :
EquivB x y = true → Equiv x y := by
```

```
intro h; unfold EquivB at h
have : normalize x = normalize y := by
-- use `same_true_iff` proved above
have := (same_true_iff).1 h
exact this
exact this
```

FixedPoint

```
import OperatorMath.Basic

namespace OperatorMath
open Trace

-- ======================================================================
-- 🎯 COMPREHENSIVE FIXED POINT THEORY MODULE
-- This consolidates all fixed point functionality:
-- 1. Basic fixed point witnesses and construction
-- 2. Idempotent function fixed points
-- 3. Diagonalization for self-reference
-- 4. Alternative proof approaches
-- ======================================================================

-- 📏 DIAGONALIZATION FUNCTIONS (from FixedPointDiag)
-- Core diagonalization for self-reference constructions
noncomputable def diag (F : Nat → Trace) : Trace :=
F (encode (F 0))

theorem diag_spec (F : Nat → Trace) :
diag F = F (encode (F 0)) := by
unfold diag
rfl

-- 🎯 FIXED POINT THEOREMS - Main approach (using mkFixed from Basic.lean)
def fixedpoint_of_idempotent {F : Trace → Trace}
(h : ∀ t, normalize (F t) = normalize (F (F t))) :
FixpointWitness F :=
⟨F void, by simpa using h void⟩

theorem exists_fixedpoint_of_idempotent {F : Trace → Trace}
(h : ∀ t, normalize (F t) = normalize (F (F t))) :
∃ ψ, normalize ψ = normalize (F ψ) := by
have w := fixedpoint_of_idempotent (F := F) h
exact ⟨w.ψ, w.fixed⟩

-- 🔄 ALTERNATIVE FIXED POINT CONSTRUCTION (from FixedPointIDem)
-- Alternative direct construction approach for comparison
def fixedpoint_of_idempotent_alt {F : Trace → Trace}
(h : ∀ t, normalize (F t) = normalize (F (F t))) :
FixpointWitness F :=
{ ψ := F void, fixed := by simpa using h void }

theorem exists_fixedpoint_of_idempotent_alt {F : Trace → Trace}
(h : ∀ t, normalize (F t) = normalize (F (F t))) :
∃ ψ, normalize ψ = normalize (F ψ) :=
let w := fixedpoint_of_idempotent_alt (F:=F) h
⟨w.ψ, w.fixed⟩

-- 🔍 FIXED POINT ANALYSIS
-- Verify both approaches give equivalent results
theorem fixedpoint_approaches_equiv (F : Trace → Trace)
(h : ∀ t, normalize (F t) = normalize (F (F t))) :
let w1 := fixedpoint_of_idempotent h
let w2 := fixedpoint_of_idempotent_alt h
normalize w1.ψ = normalize w2.ψ := by
simp [fixedpoint_of_idempotent, fixedpoint_of_idempotent_alt]

-- 🎭 ADVANCED FIXED POINT PROPERTIES
-- General fixed point existence for certain classes of functions
theorem normalize_preserving_has_fixedpoint (F : Trace → Trace)
(h_idem : ∀ t, normalize (F t) = normalize (F (F t))) :
∃ ψ, F ψ = ψ ∨ normalize (F ψ) = normalize ψ := by
have w := fixedpoint_of_idempotent h_idem
exact ⟨w.ψ, Or.inr w.fixed.symm⟩

-- 🔗 DIAGONALIZATION FIXED POINTS
-- Fixed points via diagonalization (useful for Gödel constructions)
theorem diag_gives_self_reference (F : Nat → Trace) :
diag F = F (encode (F 0)) :=
diag_spec F

-- 🌟 FIXED POINT NORMALIZATION PROPERTIES
-- Key property: fixed points respect normalization
theorem fixedpoint_normalized (F : Trace → Trace)
(h_idem : ∀ t, normalize (F t) = normalize (F (F t))) :
let w := fixedpoint_of_idempotent h_idem
normalize w.ψ = normalize (F w.ψ) :=
(fixedpoint_of_idempotent h_idem).fixed

-- Alternative characterization using normalization
theorem exists_normalized_fixedpoint (F : Trace → Trace)
(h_idem : ∀ t, normalize (F t) = normalize (F (F t))) :
∃ ψ, normalize ψ = normalize (F ψ) :=
exists_fixedpoint_of_idempotent h_idem

end OperatorMath
-- 6 of these erros
All Messages (6)
FixedPoint.lean:28:14
type mismatch, term
h void
after simplification has type
normalize (F void) = normalize (F (F void)) : Prop
but is expected to have type
EquivB (F void) (F (F void)) = true : Prop
FixedPoint.lean:34:14
Application type mismatch: In the application
Exists.intro w.ψ w.fixed
the argument
w.fixed
has type
EquivB w.ψ (F w.ψ) = true : Prop
but is expected to have type
```

```
normalize w.ψ = normalize (F w.ψ) : Prop
```

Suggested Fixes to FixedPoint (ignore if you have a better solution)

```
import OperatorMath.Basic

open OperatorMath

/- A witness now records **both** views so any file can use whichever
flavour it prefers. -/
structure FixpointWitness (F : Trace → Trace) : Type where
ψ : Trace
fixed_B : EquivB ψ (F ψ) = true
fixed_P : Equiv ψ (F ψ) -- derived, no effort

/-- Build a witness from the Boolean equation -/
def mkFixedB {F ψ}
(h : EquivB ψ (F ψ) = true) : FixpointWitness F :=
by
refine ⟨ψ, h, ?_⟩
-- Boolean ⇒ Prop via the bridge lemma we added in Basic
have : EquivB ψ (F ψ) = true := h
exact (Equiv_of_EquivB_true this)

/-- **Constant** Gödel-style operator Φ -/
def Φ (_ : Trace) : Trace := integrate (delta void)

@[simp] theorem Φ_idempotent (t) :
EquivB (Φ t) (Φ (Φ t)) = true := by
simp [Φ, normalize, beq_refl]

/-- Concrete fixed point **G = Φ void** -/
def G_witness : FixpointWitness Φ :=
mkFixedB (by
simp [Φ, EquivB, beq_refl, normalize])

def G : Trace := G_witness.ψ

theorem godel_fixed_point_B : EquivB G (Φ G) = true :=
G_witness.fixed_B

theorem godel_fixed_point_P : Equiv G (Φ G) :=
G_witness.fixed_P
```

NormalForm

```
import OperatorMath.Basic
import OperatorMath.Confluence

namespace OperatorMath
open Trace

def IsNormal (t : Trace) : Prop := normalize t = t

private theorem normalize_step_eq {a b : Trace} (h : Step a b) :
normalize a = normalize b := by
```

```
cases h <;> simp [normalize, mergeSimp, isVoid, same]
case merge_void_right =>
intro h
cases hp : normalize b with
| void => rfl
| delta t =>
have : ¬(isVoid (delta t)) := by simp [isVoid]
have : isVoid (delta t) := by rw [← hp]; exact h
contradiction
| integrate t =>
have : ¬(isVoid (integrate t)) := by simp [isVoid]
have : isVoid (integrate t) := by rw [← hp]; exact h
contradiction
| merge a b =>
have : ¬(isVoid (merge a b)) := by simp [isVoid]
have : isVoid (merge a b) := by rw [← hp]; exact h
contradiction

private theorem normalize_stepStar_eq {a b : Trace} (h : StepStar a b) :
normalize a = normalize b := by
induction h with
| refl => rfl
| trans _ _ _ h₁ _ ih =>
exact (normalize_step_eq h₁).trans ih

theorem reaches_normal_eq_normalize
{t u : Trace} (h : StepStar t u) (nu : IsNormal u) :
u = normalize t := by
have h' := normalize_stepStar_eq h
exact (h'.trans nu).symm

theorem unique_normal_form
{t u v : Trace}
(hu : StepStar t u) (hv : StepStar t v)
(nu : IsNormal u) (nv : IsNormal v) :
u = v := by
have hu' := reaches_normal_eq_normalize hu nu
have hv' := reaches_normal_eq_normalize hv nv
exact hu'.trans hv'.symm

end OperatorMath
```

```
All Messages (3)
NormalForm.lean:11:50
unknown identifier 'same'
NormalForm.lean:11:50
unknown identifier 'same'
NormalForm.lean:11:50
unknown identifier 'same'
```

LogicLayers

```
import OperatorMath.Basic
import OperatorMath.Negation

namespace OperatorMath
```

```
open Trace

-- def Equiv (a b : Trace) : Prop := normalize a = normalize b

-- theorem Equiv.refl (t) : Equiv t t := rfl
-- theorem Equiv.symm {a b} : Equiv a b → Equiv b a := Eq.symm
-- theorem Equiv.trans {a b c} : Equiv a b → Equiv b c → Equiv a c :=
-- Eq.trans

theorem Equiv.delta {a b} (h : Equiv a b) : Equiv (Trace.delta a) (Trace.delta b) :=
calc normalize (Trace.delta a)
= Trace.delta (normalize a) := rfl
_ = Trace.delta (normalize b) := congrArg Trace.delta h
_ = normalize (Trace.delta b) := rfl

theorem Equiv.integrate {a b} (h : Equiv a b) : Equiv (Trace.integrate a) (Trace.integrate b) :=
calc normalize (Trace.integrate a)
= Trace.integrate (normalize a) := rfl
_ = Trace.integrate (normalize b) := congrArg Trace.integrate h
_ = normalize (Trace.integrate b) := rfl

theorem Equiv.merge_left {a a' b} (h : Equiv a a') :
Equiv (merge a b) (merge a' b) := by
unfold Equiv at h ⊢; simp only [normalize, h]

theorem Equiv.merge_right {a b b'} (h : Equiv b b') :
Equiv (merge a b) (merge a b') := by
unfold Equiv at h ⊢; simp only [normalize, h]

theorem Equiv.merge {a a' b b'} (ha : Equiv a a') (hb : Equiv b b') :
Equiv (merge a b) (merge a' b') :=
(Equiv.merge_left ha).trans (Equiv.merge_right hb)

-- Our trace-based conditional equivalence
theorem conditional_equiv (c : Prop) [Decidable c] (p q : Trace) :
(c → Equiv p q) → Equiv (if c then p else q) (if c then p else q) := by
intro h
rfl

end OperatorMath
```

Suggested Fixes to FixedPoint (ignore if you have a better solution)

Suggested Changes