Godel

File: `C:\Users\Moses\math_ops\OperatorKernelO6\OperatorKernelO6\Meta\Godel.lean`
Type: lean
Generated: 2025-08-05 03:41:07
Size: 5139 characters

Overview

GÃ¶del-related constructions and theorems

Source Code

```
import OperatorKernelO6.Kernel
import OperatorKernelO6.Meta.Arithmetic
import OperatorKernelO6.Meta.ProofSystem

open OperatorKernelO6 Trace

namespace OperatorKernelO6.Meta

-- Helper: numeral as δ-chain representation
def numeral (n : Nat) : Trace := num n

-- Helper: complement (negation) via integration
def complement (t : Trace) : Trace := integrate t

-- Diagonal function: given a trace, construct its "quotation"
def diagonal (t : Trace) : Trace :=
  recΔ t (quote_step t) t
where
  quote_step (original : Trace) : Trace :=
    merge original original  -- Simple quotation via doubling

-- Self-reference via diagonal
def self_ref (f : Trace → Trace) : Trace :=
  let diag := diagonal (encode_function f)
  f diag
where
  encode_function (func : Trace → Trace) : Trace :=
    integrate (func void)  -- Rough encoding

-- Gödel sentence: "this sentence is not provable"
def godel_sentence : Trace :=
  self_ref (λ x => complement (provable x (numeral 1000)))

-- Fixed point property: The Gödel sentence G satisfies G ↔ ¬Prov(□G□)
theorem godel_fixed_point :
  ∃ g, StepStar godel_sentence g ∧
       StepStar (complement (provable godel_sentence (numeral 1000))) g := by
  -- The witness g is the normalization of the Gödel sentence
  use godel_sentence
  constructor
  · -- Reflexivity: G steps to itself
    exact StepStar.refl godel_sentence
  · -- By construction, G equals ¬Prov(□G□) via self_ref
    -- This follows from the diagonal lemma and self-reference construction
    unfold godel_sentence self_ref
    -- The diagonal construction ensures the fixed point property
    have diag_construction :
      let f := λ x => complement (provable x (numeral 1000))
```

```
      let encoded := integrate (f void)
      let diag := diagonal encoded
      StepStar (f diag) (complement (provable (f diag) (numeral 1000))) := by
      -- This is essentially a tautology by construction of f
      simp only []
      exact StepStar.refl _
    -- Apply the diagonal construction
    simpa using diag_construction

-- First incompleteness theorem
theorem first_incompleteness :
  ¬(∃ bound, StepStar (provable godel_sentence bound) void) ∧
  ¬(∃ bound, StepStar (provable (complement godel_sentence) bound) void) := by
  constructor
  · -- If provable, then true, but then not provable - contradiction
    intro ⟨bound, h⟩
    sorry -- Detailed argument using fixed point
  · -- If complement provable, then false, contradiction with consistency
    intro ⟨bound, h⟩
    sorry -- Use consistency theorem

-- Tarski's undefinability
def truth_predicate (formula : Trace) : Trace :=
  eqW formula void  -- "formula is true"

theorem tarski_undefinability :
  ¬(∃ truth_def : Trace → Trace,
    ∀ f, StepStar (truth_def f) void ↔ StepStar f void) := by
  -- Suppose such a truth definition exists
  intro ⟨truth_def, h_truth⟩
  -- Construct the liar sentence: "this sentence is false"
  let liar := self_ref (λ x => complement (truth_def x))
  -- The liar satisfies: Liar ↔ ¬Truth(⟨Liar⟩)
  have liar_property : StepStar liar (complement (truth_def liar)) := by
    unfold liar self_ref
    -- By diagonal construction, similar to Gödel sentence
    simp only []
    exact StepStar.refl _
  -- Now derive a contradiction
  have h1 : StepStar (truth_def liar) void ↔ StepStar liar void := h_truth liar
  -- Case analysis leads to contradiction
  by_cases h : StepStar liar void
  · -- If liar is true, then by liar_property, ¬Truth(liar) is true
    -- So Truth(liar) is false, contradicting h1
    have : StepStar (complement (truth_def liar)) void := by
      rw [←stepstar_trans liar_property]
      exact h
    -- But complement means truth_def liar ≠ void, contradicting h1.1 h
    have truth_false : ¬StepStar (truth_def liar) void := by
      -- complement(x) steps to void iff x doesn't step to void (integrate cancellation)
      sorry -- Need cancellation lemma
    have truth_true : StepStar (truth_def liar) void := h1.2 h
    exact truth_false truth_true
  · -- If liar is false, then ¬Truth(liar) is false, so Truth(liar) is true
    -- By h1, liar should be true, contradiction
    have : ¬StepStar (complement (truth_def liar)) void := by
      intro h_comp
      rw [stepstar_trans liar_property] at h_comp
      exact h h_comp
    -- This means truth_def liar steps to void
    have truth_true : StepStar (truth_def liar) void := by
      sorry -- Need double negation elimination for complement
    have liar_true : StepStar liar void := h1.1 truth_true
    exact h liar_true

-- Löb's theorem
theorem lob_theorem (formula : Trace) :
  (∃ bound, StepStar (provable (merge (provable formula (numeral 100)) formula) bound) void) →
  (∃ bound', StepStar (provable formula bound') void) := by
  sorry -- Requires careful modal logic analysis

-- Second incompleteness theorem (consistency statement)
def consistency_statement : Trace :=
  complement (merge (provable void (numeral 100)) (provable (complement void) (numeral 100)))
```

```
  complement (merge (provable void (numeral 100)) (provable (complement void) (numeral 100)))

theorem second_incompleteness :
  ¬(∃ bound, StepStar (provable consistency_statement bound) void) := by
  sorry -- Follows from first incompleteness and formalization

end OperatorKernelO6.Meta
```