Appendix: TDL vs Isabelle/HOL - Complete Comparative Analysis

Introduction: Why Compare with Isabelle/HOL?

Isabelle/HOL represents one of the most mature and successful approaches to formal mathematics, with decades of development and a substantial library of formalized mathematics. Unlike Lean's focus on dependent types, Isabelle/HOL is built on Higher-Order Logic (HOL) and emphasizes declarative proof styles that are closer to traditional mathematical writing.

This comparison is particularly important because:

- Mathematical Tradition: Isabelle's declarative style influenced TDL's design philosophy
- Proof Readability: Both systems prioritize human-readable proofs over tactic soup
- Maturity: Isabelle has 30+ years of development and real-world mathematical formalization
- Different Foundations: HOL vs Dependent Type Theory represents a fundamental design choice

Basic Syntax Comparison

Simple Definitions

Isabelle/HOL:

```
definition factorial :: "nat ⇒ nat" where
  "factorial n = (if n = 0 then 1 else n * factorial (n - 1))"
```

TDL:

```
definition Factorial: (Natural) -> Natural {
  interpretation recursive {
    map: (n) -> if n = 0 then 1 else n * Factorial(n - 1)
  }
}
```

TDL Advantage: Multiple interpretations, clearer type syntax, explicit recursion handling.

Structure Definitions

Isabelle/HOL:

```
record group =
  carrier :: "'a set"
  mult :: "'a ⇒ 'a ⇒ 'a"
  one :: "'a"
  inv :: "'a ⇒ 'a"

locale group =
  fixes G (structure)
  assumes assoc: "⟦a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧
                  ⍰ mult G (mult G a b) c = mult G a (mult G b c)"
  and l_one: "a ∈ carrier G ⍰ mult G (one G) a = a"
  and l_inv: "a ∈ carrier G ⍰ mult G (inv G a) a = one G"
```

TDL:

```
structure Group {
  carrier: Set<Element>,
  op: Map<(carrier, carrier), carrier>,
  identity: carrier,
  inverse: Map<carrier, carrier>,
```

```
  laws [
    associativity: forall a,b,c in carrier => op(op(a,b),c) = op(a,op(b,c)),
    identity_law: forall a in carrier => op(identity,a) = a & op(a,identity) = a,
    inverse_law: forall a in carrier => op(inverse(a),a) = identity
  ]
}
```

TDL Advantages:
- Unified syntax: No separation between data (`record`) and laws (`locale`)
- Integrated laws: Axioms are part of the structure definition
- Clear quantification: Explicit domain specification (`in carrier`)
- Readable laws: Natural mathematical notation

Modern Mathematics Support

Higher-Order Constructions

Isabelle/HOL Limitation - Category Theory:

```
(* Isabelle struggles with higher-order constructions *)
record 'obj category =
  objects :: "'obj set"
  arrows :: "'obj ⇒ 'obj ⇒ 'arr set"
  compose :: "'arr ⇒ 'arr ⇒ 'arr"
  id :: "'obj ⇒ 'arr"

(* Functors require complex type manipulations *)
locale functor =
  fixes F_obj :: "'a ⇒ 'b"
  and F_arr :: "('a category) ⇒ ('b category) ⇒ 'arr_a ⇒ 'arr_b"
  (* Complex assumptions about preservation... *)
```

TDL - Natural Category Theory:

```
structure Category {
  objects: Set<Object>,
  arrows: Map<(objects, objects), Set<Arrow>>,
  compose: Map<(Arrow, Arrow), Arrow>,
  identity: Map<objects, Arrow>,

  laws [
    associativity: forall f,g,h where composable => compose(f,compose(g,h)) =
compose(compose(f,g),h),
    identity_laws: forall f: Arrow => compose(identity(source(f)), f) = f &
compose(f, identity(target(f))) = f
  ]
}

view Functor<C1: Category, C2: Category> {
  object_map: Map<C1.objects, C2.objects>,
  arrow_map: Map<C1.arrows, C2.arrows>,

  preservation_laws [
    functoriality: forall f,g in C1 => arrow_map(C1.compose(f,g)) =
C2.compose(arrow_map(f), arrow_map(g)),
    identity_preservation: forall x in C1.objects => arrow_map(C1.identity(x)) =
C2.identity(object_map(x))
```

```
  ]
}
```

TDL Advantages:
- Native higher-order support: Categories and functors are first-class concepts
- Automatic type inference: No complex type annotations needed
- Mathematical clarity: Reads like category theory textbook definitions

Tactic System Comparison

Proof Construction Philosophy

Isabelle's Approach - Isar Declarative Style:

```
theorem fundamental_group_abelian_iff_commutator_trivial:
  assumes "topological_space X" and "path_connected X"
  shows "abelian_group (fundamental_group X x₀) ↔
         (∀α β. homotopic (compose α (compose β (inverse α))) (inverse β))"
proof
  assume abelian: "abelian_group (fundamental_group X x₀)"
  show "∀α β. homotopic (compose α (compose β (inverse α))) (inverse β)"
  proof (intro allI)
    fix α β
    have "compose α (compose β (inverse α)) = compose (compose α β) (inverse α)"
      by (simp add: path_compose_assoc)
    also have "... = compose (compose β α) (inverse α)"
      using abelian by (simp add: abelian_group.commute)
    also have "... = compose β (compose α (inverse α))"
      by (simp add: path_compose_assoc)
    also have "... = compose β (path_refl x₀)"
      by (simp add: path_inverse_compose)
    also have "... = β"
      by (simp add: path_compose_refl)
    finally show "homotopic (compose α (compose β (inverse α))) (inverse β)"
      using group_inverse_unique by simp
  qed
next
  (* Reverse direction... many more lines *)
qed
```

TDL's Approach - Mathematical Reasoning:

```
theorem "Fundamental Group Commutativity Characterization"
  context [
    X: forall TopologicalSpace where PathConnected<X>,
    x₀: forall X.points
  ]
  shows {
    IsAbelian(FundamentalGroup(X, x₀)) ↔
    ∀α,β: Loop(X,x₀) => α·β·α⁻¹ ≃ β⁻¹
  }
  proof {
    direction forward: {
      assume abelian: IsAbelian(FundamentalGroup(X, x₀))
      let α,β: forall Loop(X,x₀)

      calc α·β·α⁻¹
      = (α·β)·α⁻¹        by associativity
```

```
      = (β·α)·α⁻¹        by abelian_property
      = β·(α·α⁻¹)        by associativity
      = β·e              by inverse_law
      = β                by identity_law

      therefore α·β·α⁻¹ ≃ β by homotopy_from_equality
      conclude α·β·α⁻¹ ≃ β⁻¹ by group_inverse_unique
    }

    direction backward: {
      assume commutator_trivial: ∀α,β => α·β·α⁻¹ ≃ β⁻¹
      show ∀α,β => α·β ≃ β·α by {
        let α,β: forall Loop(X,x₀)
        have α·β·α⁻¹ ≃ β⁻¹ by commutator_trivial
        apply right_multiplication(α) to both_sides
        get α·β ≃ β⁻¹·α by path_homotopy_algebra
        apply group_inverse_unique
        conclude α·β ≃ β·α
      }
    }
  }
```

TDL Advantages:

- Bidirectional proof structure: Clear `direction forward/backward` for equivalences
- Mathematical notation: Uses standard symbols (≃ for homotopy, · for composition)
- High-level reasoning: Appeals to standard algebraic manipulations
- Calc chains: Direct mathematical computation style
- Proof architecture: Clear logical flow without Isar bureaucracy

Foundational Advantages

Type System Expressivity

Isabelle's HOL Limitations:

```
(* Isabelle cannot naturally express dependent types *)
(* Vector spaces must use workarounds: *)
typedef 'a vector_space = "{(V,add,smul). vector_space_axioms V add smul}"

(* Path types require complex encoding: *)
definition path :: "'a ⇒ 'a ⇒ ('a path_type)"
  where "path a b = {f. continuous_on {0..1} f ∧ f 0 = a ∧ f 1 = b}"

(* Higher inductive types impossible: *)
(* Cannot define S¹ as quotient of [0,1] with endpoints identified *)
```

TDL's Native Dependent Types:

```
structure VectorSpace<F: Field> {
  carrier: Set<Vector>,
  add: Map<(carrier, carrier), carrier>,
  scalar_mult: Map<(F.carrier, carrier), carrier>,
  zero: carrier,

  laws [
    vector_addition_abelian: Group(carrier, add, zero),
    scalar_distributivity: forall k: F.carrier, v,w: carrier =>
      k • (v + w) = (k • v) + (k • w),
```

```
    field_distributivity: forall k,l: F.carrier, v: carrier =>
      (k + l) • v = (k • v) + (l • v)
  ]
}

// Path types are natural:
constructor Path<X: TopologicalSpace>(a,b: X.points) -> PathType<X> {
  proof {
    let result: PathType<X> = {
      map: ContinuousMap([0,1], X),
      start_condition: map(0) = a,
      end_condition: map(1) = b
    }
    return result
  }
}

// Higher inductive types work naturally:
structure Circle {
  base: Point,
  loop: PathType<Circle>(base, base),

  quotient_law: forall p: PathType<RealInterval([0,1]), base> =>
    p(0) = p(1) ? identify_endpoints(p)
}
```

Modern Mathematics Integration

What Isabelle Cannot Do:

- Univalence: No computational content for path equality
- Higher Categories: Complex encoding required, loses mathematical intuition
- Synthetic Homotopy Theory: Impossible without higher inductive types
- Cubical Types: No native support for higher-dimensional equality

What TDL Enables:

```
// Univalence axiom with computational content:
axiom Univalence<A,B: Type> {
  Equivalent<A,B> ≃ PathType<Universe>(A,B)
}

// Natural higher categories:
structure InfinityCategory {
  objects: Set<Object>,
  morphisms: forall n: Natural => Map<objects^(n+1), Set<nMorphism>>,
  composition: InfiniteComposition,

  laws [
    segal_condition: forall n => SegalMaps(morphisms[n]) are_equivalences,
    completeness: InnerHornFilling(morphisms)
  ]
}

// Synthetic homotopy theory:
theorem "Fundamental Group of Circle"
  shows { FundamentalGroup(Circle, base) ≅ Integers }
  proof {
```

```
    // Use path space and loop space directly
    let loop_space = PathType<Circle>(base, base)
    show loop_space ≃ CircleMap by univalence
    conclude FundamentalGroup(Circle, base) ≅ Z by winding_number_theorem
  }
```

Summary: Why TDL Surpasses Isabelle

Aspect	Isabelle/HOL	TDL
Foundations	Simple HOL	Dependent types + univalence
Modern Math	Complex workarounds	Native support
Proof Style	Isar declarative	Mathematical reasoning
Type System	Monomorphic	Polymorphic + dependent
Category Theory	Difficult encoding	First-class support
Homotopy Theory	Impossible/awkward	Synthetic and natural
Mathematical Notation	Limited	Rich + customizable
Learning Curve	Steep (proof infrastructure)	Natural (mathematical thinking)

The Verdict: While Isabelle/HOL has served the formal methods community well for decades, TDL represents the next generation of mathematical formalization. TDL combines Isabelle's declarative philosophy with modern type theory, dependent types, and native support for 21st-century mathematics.

For working mathematicians, TDL offers:
- Mathematical authenticity: Proofs read like mathematics, not proof scripts
- Modern foundations: Native support for homotopy theory, higher categories, univalence
- Computational content: Definitions can be executed, not just verified
- Collaborative development: Git-like versioning for mathematical proofs

TDL is not just an improvement over Isabelle—it's a fundamental advance that makes formal mathematics accessible to the broader mathematical community.