NNO-Internal Transit Optimization: Simplicial Semantics and a Typed Lean-to-Rust Compiler

Anonymous

1 November 2026

Abstract

We specify transit optimization in the internal language of a topos with a natural number object (NNO), and interpret it in higher toposes. Directed journey categories yield simplicial route spaces, while weighted realizations determine which changes of representation preserve optimization. A Lean implementation provides typed arithmetic, logical and higher-cell terms, and compiles a finite resource-constrained fragment to Rust with scalar, AVX2 and Rayon backends. We prove finite-search correctness, exact-realization invariance, safe subdivision and a higher truncation principle. Kernel-checked lemmas and executable tests are distinguished from mathematical arguments and unverified backend code.

Keywords: natural number object; transit routing; simplicial set; enriched category; higher topos; dependent types; staged compilation.

1. Problem, scope and contribution

Can a bus-routing problem be represented in another topological or higher-categorical space without changing its optimal journeys? Yes, provided the representation retains direction, feasible composition, resource consumption and objective values. An unweighted homotopy type does not suffice.

We study passenger journey optimization in a fixed, finite timetable: minimize nonnegative integral generalized cost subject to a boarding budget and decidable edge constraints. This is a foundation for dispatch software, not a solution to every fleet-design, vehicle-routing or network-design problem. Vehicle capacities, driver duties and interactions among passengers require additional state or joint optimization.

The contribution is a precise integration of internal arithmetic, simplicial representation and a runnable typed compiler. The ingredients include established category theory, directed topology and dynamic programming; we do not claim their invention. The new artifact is the explicit realization contract connecting typed higher/topos terms to this particular accelerated optimization fragment.

RAPTOR computes transit Pareto sets by rounds [1]; CSA exploits timetable connection order [2]. Our backend is a simple resource-expanded DAG dynamic program, not an implementation or performance replacement for either algorithm. Lawvere's enrichment [3], Grandis's weighted directed topology [4] and higher-topos semantics [5] supply the mathematical context.

2. Ambient topos, NNO and arithmetic

Let $\mathcal{E}$ be an elementary topos with a natural number object $(N, 0, s)$. Thus for every $z : 1 \to A$ and $h : A \to A$ there is a unique $f : N \to A$ with $f0 = z$ and $fs = hf$. Parameterized recursion follows in a cartesian closed category by transposing into an exponential. This universal property, rather than an assumed external machine integer type, specifies arithmetic.

Primitive recursion defines addition, multiplication, exponentiation, truncated subtraction, minimum, maximum, comparison, quotient and remainder. We choose total conventions $a/0 = 0$ and $a \bmod 0 = a$. The comparison relations on N are decidable constructively. Given a decidable predicate, bounded conjunction and disjunction are computable by NNO recursion. These operations suffice for the executable constraints below. Negative integers and rationals admit further constructions, but the current backend uses bounded natural numbers; decidable comparison of arbitrary real numbers is not asserted.

Proposition 2.1 (finite internal minimization). Suppose a list of candidates has a specified finite length in N, a decidable feasibility predicate and a natural-valued objective. There is an internally definable operation returning either no feasible candidate or a feasible candidate with least objective value.

Proof. Fold over the list while maintaining an optional best candidate. Ignore infeasible entries; initialize from the first feasible entry; replace the current best precisely when a smaller objective is encountered. Decidability makes each branch available. NNO induction proves that the accumulator is empty exactly when no processed entry is feasible, and otherwise is feasible and no more expensive than every processed feasible entry. No excluded middle for general propositions or choice principle is used. □

For an ∞-topos $\mathcal{X}$, take $N_{\mathcal{X}} = \coprod_{k\in\mathbb{N}} 1$. It is 0-truncated. The shift and zeroth summand define successor and zero; the space of recursion maps together with their coherent recursion data is contractible. One construction is the colimit of the initial chain for $1 + (-)$; coproducts preserve the relevant colimit. Equivalently, recursion assigns the successive iterates of a specified endomorphism, with a contractible space of choices. We use this higher universal property, not a collection of independent numerals in each homotopy dimension. Arithmetic remains discrete even when the objects on which it acts have higher cells.

3. A decidable finite transit fragment

Work internally in $\mathcal{E}$. Fix a positive bound $n : N$, vertices $V = \{i : N \mid i < n\}$, and a finite list E of directed, possibly parallel edges. Each edge carries source u_e, target v_e, cost $c_e : N$ and boarding count $b_e : N$. Require $u_e < v_e$ and $c_e < W$ for a positive W. A typed Boolean guard filters the list. Fix source a, destination z and budget B.

A candidate is an edge-list path from a to z with total boardings at most B. Its objective is $c(p) = \sum_{e\in p} c_e$. The empty path is admitted at a vertex and has cost and resource zero. Strictly increasing ranks imply that a path has at most $n - 1$ edges, so bounded enumeration is available. Repeated stops are permitted when they occur at distinct time-expanded states.

For timetable semantics, states must distinguish event time and onboard/offboard trip information. Boarding charges one; continuing on the same vehicle charges zero. Transfers need explicit walking and minimum-connection constraints. An auxiliary destination sink, with zero-cost incoming arcs from destination arrivals, allows arrival times to differ: if every edge joins events and charges elapsed time, every path to the same event otherwise has the same elapsed cost. Initial departure time is fixed. Trip-specific rules must be resolved before lowering. GTFS specifies relevant stop-time, transfer and station-pathway data [6], but this artifact does not include a GTFS importer.

Our rank condition is an actual hypothesis. Zero-duration transfer cycles are not silently discarded. Such inputs need a separately justified normalization or another backend. History-dependent fares and accessibility rules require lifting vertices to sufficient finite states before applying the theorem.

Write $D(v, j)$ for minimum cost to v with exactly j boardings, using a formal ∞ for the empty minimum. Initialize $D(a, 0) = 0$ and all other entries to ∞. In increasing vertex order, for each edge $e : u \to v$ and $j + b_e \le B$, update

$$D(v, j + b_e) \leftarrow \min\{D(v, j + b_e), D(u, j) + c_e\}.$$

Return $\min_{j\le B} D(z, j)$ and a predecessor witness, or infeasible.

Theorem 3.1 (internal DP correctness). This recurrence returns exactly the least cost of a feasible candidate, and a path attaining it when one exists.

Proof. At vertex v, every nonempty path ending there has a last edge from some $u < v$. Its prefix has already been processed. Conversely every update appends an actual allowed edge to a feasible prefix and adds its exact resources. Induction on vertex rank therefore proves equality between each finalized table entry and the minimum over paths with those endpoints and exact boarding count. Initialization supplies precisely the empty path. Minimization over the finitely many counts proves the claim. A finite minimizing predecessor exists whenever a value is finite; following predecessors strictly decreases vertex rank and reconstructs a witness. All induction and comparisons are internal NNO operations. □

Distance computation uses $O(n + m + (B + 1)m)$ arithmetic operations and $O(n(B + 1) + m)$ storage, where m is the filtered edge count. Our simple witness reconstruction scans the edge list at each predecessor and can add $O(nm)$ work. The CLI can also recompute the scalar solution for validation. Large binary-encoded budgets cause pseudopolynomial state growth; the categorical formulation does not remove it.

4. The simplicial route space

Let C_G be the free category on the allowed finite directed graph. Objects are event states; arrows are actual directed edge-list journeys; composition concatenates lists. The nerve has

$$K_r = N(C_G)_r = \coprod_{x_0,\dots,x_r} C_G(x_0, x_1) \times \cdots \times C_G(x_{r-1}, x_r).$$

Faces compose or omit arrows, and degeneracies insert identities. Thus a 2-simplex records f, g, gf with their compositional compatibility; it is not an assertion that three nearby stops are mutually reachable. Associativity supplies the higher simplicial identities. The nerve is an ∞-category, generally not a Kan complex. Its realization $X_G = |K|$ is the requested alternative topological route space, used together with its original directed simplicial and weighted data.

The cost and resource label $w(p) = (c(p), b(p))$ is additive under composition. Consequently each triangular face satisfies $w(gf) = w(f) + w(g)$. Scalar shortest costs give a Lawvere metric with values in $([0, \infty], \geq, +, 0)$: $d(x, x) = 0$ and $d(x, z) \leq d(x, y) + d(y, z)$. Symmetry is unnecessary. The finite resource table retains more information than a single scalar distance.

Proposition 4.1 (topology alone is insufficient). Ordinary homotopy equivalence of route realizations does not imply preservation of reachability or optimum cost.

Proof. A single directed edge and its reversal both realize an interval; fixing named endpoints, the reachability answers are opposite. Keeping its orientation and changing its cost from 1 to 100 leaves the entire unweighted nerve unchanged but changes the optimum. Inverting arrows to obtain a groupoid also admits reverse journeys that the input need not allow. □

Theorem 4.2 (safe simplicial refinement). Replace an edge $u \to v$ by $u \to q \to v$, where q is new, has no other incident edges, and is not a query endpoint. Split both labels additively into nonnegative integers. Then feasible endpoint journeys correspond bijectively with identical cost and boarding count.

Proof. Expand each occurrence of the original edge into the two new edges. Every path through q must use exactly those two edges consecutively, so contraction gives the inverse. Additivity preserves labels and budget tests. The expanded graph has a compatible rank order obtained by topological sorting. A new guard must retain both refined edges, and new machine-range certificates must be checked; arbitrary guards are not automatically preserved by subdivision. □

This is a representation theorem about directed weighted subdivision. It is stronger than an unweighted homeomorphism claim and narrower than unrestricted triangulation invariance.

5. Higher cells, toposes and finite realization

Higher cells can express certified route rewrites, associators, commuting independent actions and coherent refinements. They do not automatically erase time or resource differences. For example, declare a 2-cell between two routes only when the intended semantics permits that rewrite, and require equal objective/resource labels if the cell is to be invisible to this optimizer.

Proposition 5.1 (discrete objectives factor through components). If a mapping space M in an ∞-topos has a cost map $M \to N_{\mathcal{X}}$, it factors uniquely up to contractible choice through its 0-truncation $\tau_{\leq 0}M$.

Proof. This is the defining adjunction for 0-truncation applied to a 0-truncated target. Thus connected higher cells have identical cost under a genuinely defined cost map. The same applies to a pair of natural-valued labels and to decidable feasibility. □

This proposition neither says that all routes are equivalent nor that higher witnesses have no verification value. It identifies exactly what a discrete objective can observe. The finite backend requires a finite realization of the relevant feasible components; it need not enumerate every syntactic representative.

Several ambient toposes are possible. $\mathrm{Sh}(X_G)$ has its sheaf NNO and can host locally varying routing data. Its higher analogue is $\mathrm{Sh}(X_G; \mathcal{S})$. For a small scenario category A, $\mathrm{Fun}(N(A)^{op}, \mathcal{S})$ is a presheaf ∞-topos with an objectwise NNO. Internal finite-search arguments apply when finite lists, decidability and the required bounds are supplied internally. These claims do not identify an arbitrary higher category with a topos.

Feasible sections specified by a subobject of a sheaf satisfy descent: matching local sections glue uniquely and membership can be checked locally. An internally defined subobject of minimizers is also a sheaf. This must not be confused with solving independent regional routing problems after deleting competing routes or cross-boundary constraints. Those altered problems can have different minimizers. Ordinary global sections need not turn local existence into a global choice of route. Compilation therefore requires a chosen computational realization, not an assumed equivalence of a general topos with sets.

One further distinction prevents a misleading presheaf argument. The representable $C_G(-, z)$ records paths into z, but path cost is generally not a natural map from this presheaf to constant N: precomposing a path by f adds $c(f)$. Weighted/enriched data, or an explicitly varying cost bundle, must be retained. The nerve and the ambient topos organize the semantics; neither supplies free optimization invariance.

Definition 5.2 (exact finite realization). A source specification has a type S of semantic candidates, a feasibility predicate F and objective $q : S \to N$. A finite routing problem P realizes it when there are: (i) a decoder taking each feasible route r of P to $s_r : S$ with $F(s_r)$ and $q(s_r) = c(r)$; (ii) for every feasible $s : S$, some feasible r with $c(r) = q(s)$. Injectivity is unnecessary.

Theorem 5.3 (minimum transport). If a least-cost feasible route of P exists, its decoded candidate is a least-cost feasible source candidate. Conversely source feasibility is equivalent to route feasibility; infeasibility and the set of feasible objective values are preserved.

Proof. Decoding gives a feasible source candidate with the same cost. For every competing source candidate, clause (ii) supplies a route with that objective, so route minimality gives the required inequality. The two clauses give both inclusions of feasible objective-value sets and both directions of feasibility. □

The theorem applies to terms in a higher-category model or the internal language of a topos when those terms come with the stated finite realization. It does not provide a decision procedure for arbitrary higher-topos terms. In particular, NNO recursion does not make unbounded quantification decidable.

6. Implemented Lean source language and compiler

The companion `Compiler.lean` is a complete executable Lean 4.24.0 program [7], reproduced in the PDF appendix. It is a hosted typed language: Lean elaborates the specification and checks its proof fields; `lean --run` specializes closed arithmetic and emits Rust source. Editing a numerical instance does not require changing the compiler functions.

Its source layer includes: a universe-polymorphic category with intrinsically typed composition; the NNO universal property and a checked set model; dimension-indexed higher-cell generators, identities and compositions; and a typed internal-language syntax with products, functions, N, Booleans, Ω, power-object terms, primitive recursion, arithmetic, comparisons, and bounded quantifiers. Unbounded logical quantification is allowed into Ω but has no elimination into executable Boolean constraints.

The current denotation of `Term` is in sets, with `omega` interpreted as `Prop` and power objects as predicates. General topos interpretation is mathematical semantics; this version does not mechanically construct every topos or implement all dependent type formers. `HigherModel` is an operation signature supplied by a model, not a kernel proof of all globular/coherence axioms. It supports arbitrary finite dimensions and composition indices; the executable higher-cell example is the terminal strict model. Models with nontrivial coherences require additional laws and realization proofs. No automatic strictification of arbitrary weak higher categories is claimed.

The general compiler interface accepts any Lean type α together with an exact realization, including types of categorical/higher-cell or internal-language terms. In abbreviated form, the implemented contract is:

```
structure Realization {H n w} (α : Type u) (p : Problem H n w) where
  feasible : α → Prop
  objective : α → Nat
  decode : Candidate p → α
  sound : ∀ r, feasible (decode r)
  costExact : ∀ r, objective (decode r) = r.val.cost
  complete : ∀ a, feasible a → ∃ r : Candidate p,
    r.val.cost = objective a
```

The implemented `minimum_transport` proves the minimality part of Theorem 5.3. The compiler consumes its `Problem` argument while retaining this realization contract for source-level reasoning. Arbitrary source terms are not lowered merely because they type-check: a suitable finite realization is an additional requirement.

`Problem` requires `Fin n` endpoints, forward-rank proofs, typed closed cost/resource expressions, proofs of their bounds, a Boolean guard, a boarding bound and a state-space bound. Guards see source, target, cost and boardings. Addition, truncated subtraction, multiplication, division, remainder, exponentiation, min/max, all six comparisons, Boolean connectives and bounded quantifiers are executable. They are evaluated in Lean during specialization; the target executes the resulting optimization table. Higher logical propositions are not passed off as decision procedures.

The emitter prints only checked natural-number literals in a fixed Rust template. The library provides the DP and witness reconstruction. Its integer sentinel is $I = 2^{30} - 1$; certificates require $nW < I$, $B \leq 65535$ and $n(B + 1) \leq 16777216$. Thus every path cost is below I, and the target can distinguish unreachable from reachable entries. Rust revalidates the range contract, including user-supplied CLI budget overrides.

7. Rust execution and accelerator contract

The AVX2 backend uses eight unsigned 32-bit lanes via Rust's `std::arch` intrinsics [8]. For an edge of resource b, lane j updates destination lane $j + b$. Source rank is strictly lower than destination rank, so the slices are disjoint. Runtime feature detection selects AVX2 or a scalar fallback. Explicit `avx2` requests fail on unsupported processors.

Proposition 7.1 (kernel equivalence under the range contract). The scalar, AVX2 and Rayon lane updates implement the same recurrence.

Proof. Each output lane is $\min(d, \min(I, s + c))$. All finite labels and c are below I; hence even an unreachable label plus cost is below $2I < 2^{31}$ and cannot wrap a 32-bit lane. Unsigned vector addition followed by unsigned minimum therefore agrees with scalar arithmetic and sentinel saturation. Full chunks update eight independent lanes; the tail uses the scalar operation. Rayon partitions distinct mutable destination entries [9]; edges remain sequential, so there is no shared-write race between edge updates. □

This is a mathematical refinement argument, not a kernel-checked Rust proof. Unsafe intrinsic code, memory allocation, the Lean evaluator/compiler, source emission, Rust compilation and the processor remain in the execution trust boundary. There is no GPU backend in this release. A GPU implementation needs a separately checked memory, scheduling and arithmetic contract.

8. Reproducible evidence and limitations

Version 0.1.0 includes all source, a lockfile, generated Rust, positive/negative examples, tests, raw validation logs and benchmark data. Six Rust tests passed with Rayon enabled. One test compares scalar, automatic AVX2 and Rayon results on 600 deterministic small DAGs, and compares optimal values against exhaustive path enumeration. Other tests cover boarding budgets, invalid graphs/ranges, empty/unreachable routes, SIMD tails/sentinels and corrupted witnesses. Three Lean fixtures are intentionally rejected: a Boolean in arithmetic, a backward edge, and a dimension mismatch.

The example gives optimum costs $20, 12, 8$ for boarding budgets $0, 1, 2$. Both generated scalar/AVX2 execution and the Rayon executable produce a checked route witness for the compiled budget. `#print axioms` reports no axioms for minimum transport, the lowering projections, recursion equations or categorical associativity; the set-NNO construction uses the standard `Quot.sound` dependency through function extensionality. There are no admitted proofs or added axioms in the artifact. The full DP theorem, general higher-topos interpretation and Rust equivalence are proved on paper, not formalized end to end in Lean.

A synthetic distance-only benchmark uses 256 vertices, 1,014 edges, budget 1,023, three warmups and 15 measured runs, Rust 1.85.1 release mode, and four Rayon threads. Median times in this environment were:

Backend	Median microseconds
Scalar	922.148
Automatic AVX2	171.288
Rayon	52327.352

This single workload shows an approximately 5.38 scalar/AVX2 ratio and substantial Rayon overhead. It is neither a general speedup claim nor a comparison with RAPTOR/CSA or real city networks. The measured routine excludes compilation, source elaboration, witness reconstruction and reference validation. CPU/toolchain metadata and commands accompany the raw CSV. A sandbox-only same-process executable-path adapter was needed to run the unmodified Lean distribution; it did not alter the kernel or theorem sources.

The present limits are deliberate: one additive resource, nonnegative integral cost, acyclic ranked state graphs, bounded materialization and compile-time constraint specialization. No real-time telemetry ingestion, AI dispatch policy, capacity coupling or live operational safety claim belongs to this compiler artifact. Those require a subsequent application with stale-data rules, permissioned control and explicit realization validation. Unrestricted optimization of arbitrary higher-topos terms is not offered.

9. Authorship, evidence and availability

Anonymous human-directed research submission. The responsible submitting human selected the problem, authorized anonymous submission and accepted the journal's interaction terms. An OpenAI assistant drafted the manuscript, implemented the software, ran the checks and prepared the artifacts. No independent referee review, field trial, formal Rust verification or deployment is claimed. Original software is distributed under the included MIT license. Proofs and evidence are open to correction. The submission bundle contains the full Lean compiler, Rust library, reproducibility instructions and this manuscript; its ready artifacts become publicly accessible only if the journal approves publication.

References

[1] D. Delling, T. Pajor and R. F. Werneck. Round-Based Public Transit Routing. ALENEX, 2012. Author-hosted publication and paper.

[2] J. Dibbelt, T. Pajor, B. Strasser and D. Wagner. Connection Scan Algorithm. arXiv:1703.05997, 2017. Manuscript.

[3] F. W. Lawvere. Metric Spaces, Generalized Logic, and Closed Categories. 1973; TAC reprint 1, 2002. Journal reprint.

[4] M. Grandis. The Fundamental Weighted Category of a Weighted Space. arXiv:math/0604506, 2006. Manuscript.

[5] J. Lurie. Higher Topos Theory. Princeton University Press, 2009; arXiv:math/0608040. Author's manuscript record.

[6] GTFS. Schedule Reference, consulted 1 November 2026. Official specification.

[7] Lean project. The Lean Language Reference, sections on the type system, IO and compilation, consulted 1 November 2026. Official manual.

[8] Rust project. core::arch, `_mm256_min_epu32` and runtime feature detection. Intrinsic documentation; feature detection.

[9] Rayon contributors. Rayon, version 1.10.0 used in the artifact. Project repository.

Complete executable Lean compiler

The following is the complete checked compiler source from version 0.1.0. The companion archive also contains the Rust acceleration library, generated program, validation logs and negative typing fixtures.

```
import Std

/-!
Typed arithmetic/internal-language terms, dimension-polymorphic higher-cell
terms, and a finite-realization compiler. No `sorry` or user axioms.
The generic interfaces are assumptions on models, not proofs that every model
is a topos or an infinity-category. See the manuscript's verification boundary.
-/
namespace Transit

universe u v

structure Category where
  Obj : Type u
  Hom : Obj → Obj → Type v
  ident : (a : Obj) → Hom a a
  comp : {a b c : Obj} → Hom a b → Hom b c → Hom a c
  assoc : ∀ {a b c d} (f : Hom a b) (g : Hom b c) (h : Hom c d),
    comp (comp f g) h = comp f (comp g h)
  leftId : ∀ {a b} (f : Hom a b), comp (ident a) f = f
  rightId : ∀ {a b} (f : Hom a b), comp f (ident b) = f

-- The NNO universal property, parametrized by a genuine ambient category.
structure NNO (C : Category.{u,v}) where
  one : C.Obj
  terminalArrow : (a : C.Obj) → C.Hom a one
  terminalUnique : ∀ {a} (f : C.Hom a one), f = terminalArrow a
  nat : C.Obj
  zero : C.Hom one nat
  succ : C.Hom nat nat
  fold : {a : C.Obj} → C.Hom one a → C.Hom a a → C.Hom nat a
  foldZero : ∀ {a} (z : C.Hom one a) (s : C.Hom a a),
    C.comp zero (fold z s) = z
  foldSucc : ∀ {a} (z : C.Hom one a) (s : C.Hom a a),
    C.comp succ (fold z s) = C.comp (fold z s) s
  unique : ∀ {a} (z : C.Hom one a) (s : C.Hom a a)
    (f : C.Hom nat a), C.comp zero f = z →
    C.comp succ f = C.comp f s → f = fold z s

inductive CatTerm (C : Category.{u,v}) : C.Obj → C.Obj → Type (max u v) where
  | generator {a b} : C.Hom a b → CatTerm C a b
  | ident (a) : CatTerm C a a
  | comp {a b c} : CatTerm C a b → CatTerm C b c → CatTerm C a c

def CatTerm.eval {C : Category.{u,v}} {a b} : CatTerm C a b → C.Hom a b
  | .generator f => f
  | .ident a => C.ident a
  | .comp f g => C.comp f.eval g.eval

theorem categorical_associativity {C : Category.{u,v}} {a b c d}
    (f : CatTerm C a b) (g : CatTerm C b c) (h : CatTerm C c d) :
    (CatTerm.comp (CatTerm.comp f g) h).eval =
      (CatTerm.comp f (CatTerm.comp g h)).eval := C.assoc _ _ _

def setCategory : Category.{1,0} where
  Obj := Type
  Hom := fun a b => a → b
  ident := fun _ x => x
  comp := fun f g x => g (f x)
  assoc := fun _ _ _ => rfl
  leftId := fun _ => rfl
  rightId := fun _ => rfl

def setNNO : NNO setCategory where
  one := Unit
  terminalArrow := fun _ _ => ()
  terminalUnique := by intro a f; funext x; exact Subsingleton.elim _ _
  nat := Nat
  zero := fun _ => 0
  succ := Nat.succ
  fold := fun z s n => Nat.rec (z ()) (fun _ a => s a) n
  foldZero := fun _ _ => rfl
  foldSucc := fun _ _ => rfl
  unique := by
    intro a z s f hz hs
    funext n
```

```
    induction n with
    | zero => exact congrFun hz ()
    | succ n ih => exact (congrFun hs n).trans (congrArg s ih)

-- A globular signature supports compositions at every finite dimension.
-- Actual models supply globularity/coherence separately; the terminal model
-- below is a strict omega-category. No strictification is asserted here.
structure HigherModel where
  Cell : Nat → Type
  source : (n : Nat) → Cell (n+1) → Cell n
  target : (n : Nat) → Cell (n+1) → Cell n
  identity : (n : Nat) → Cell n → Cell (n+1)
  Composable : (n k : Nat) → Cell n → Cell n → Prop
  compose : (n k : Nat) → k < n → (x y : Cell n) →
    Composable n k x y → Cell n

inductive CellTerm (H : HigherModel) : (n : Nat) → H.Cell n → Type where
  | generator {n} (x : H.Cell n) : CellTerm H n x
  | identity {n x} : CellTerm H n x → CellTerm H (n+1) (H.identity n x)
  | compose {n k x y} (hk : k < n) (h : H.Composable n k x y) :
    CellTerm H n x → CellTerm H n y →
    CellTerm H n (H.compose n k hk x y h)

def terminalHigher : HigherModel where
  Cell := fun _ => Unit
  source := fun _ _ => ()
  target := fun _ _ => ()
  identity := fun _ _ => ()
  Composable := fun _ _ _ _ => True
  compose := fun _ _ _ _ _ _ => ()

inductive Ty where
  | unit | nat | bool | omega
  | prod (a b : Ty) | arrow (a b : Ty) | power (a : Ty)
  | cell (dimension : Nat)

@[reducible] def Ty.denote (H : HigherModel) : Ty → Type
  | .unit => Unit
  | .nat => Nat
  | .bool => Bool
  | .omega => Prop
  | .prod a b => a.denote H × b.denote H
  | .arrow a b => a.denote H → b.denote H
  | .power a => a.denote H → Prop
  | .cell n => H.Cell n

inductive Var : List Ty → Ty → Type where
  | zero : Var (a :: Γ) a
  | succ : Var Γ a → Var (b :: Γ) a

def Env (H : HigherModel) : List Ty → Type
  | [] => Unit
  | a :: Γ => a.denote H × Env H Γ

def Var.get {H : HigherModel} {Γ a} (x : Var Γ a) : Env H Γ → a.denote H :=
  match x with
  | .zero => fun ρ => ρ.1
  | .succ y => fun ρ => y.get ρ.2

inductive NatOp where
  | add | sub | mul | div | mod | pow | min | max

def NatOp.eval : NatOp → Nat → Nat → Nat
  | .add => Nat.add
  | .sub => Nat.sub
  | .mul => Nat.mul
  | .div => Nat.div
  | .mod => Nat.mod
  | .pow => Nat.pow
  | .min => Nat.min
  | .max => Nat.max

inductive Cmp where
  | eq | ne | lt | le | gt | ge

def Cmp.eval : Cmp → Nat → Nat → Bool
  | .eq, a, b => decide (a = b)
  | .ne, a, b => decide (a ≠ b)
  | .lt, a, b => decide (a < b)
  | .le, a, b => decide (a ≤ b)
  | .gt, a, b => decide (a > b)
  | .ge, a, b => decide (a ≥ b)

-- Simply typed internal language with an NNO, power objects and higher-cell
```

```
-- sorts. Unbounded logical quantifiers do not eliminate into executable Bool.
inductive Term (H : HigherModel) : List Ty → Ty → Type where
  | var : Var Γ a → Term H Γ a
  | unit : Term H Γ .unit
  | num : Nat → Term H Γ .nat
  | bit : Bool → Term H Γ .bool
  | arithmetic : NatOp → Term H Γ .nat → Term H Γ .nat → Term H Γ .nat
  | compare : Cmp → Term H Γ .nat → Term H Γ .nat → Term H Γ .bool
  | and : Term H Γ .bool → Term H Γ .bool → Term H Γ .bool
  | or : Term H Γ .bool → Term H Γ .bool → Term H Γ .bool
  | not : Term H Γ .bool → Term H Γ .bool
  | ite : Term H Γ .bool → Term H Γ a → Term H Γ a → Term H Γ a
  | pair : Term H Γ a → Term H Γ b → Term H Γ (.prod a b)
  | fst : Term H Γ (.prod a b) → Term H Γ a
  | snd : Term H Γ (.prod a b) → Term H Γ b
  | lam : Term H (a :: Γ) b → Term H Γ (.arrow a b)
  | app : Term H Γ (.arrow a b) → Term H Γ a → Term H Γ b
  | natRec : Term H Γ a → Term H Γ (.arrow .nat (.arrow a a)) →
    Term H Γ .nat → Term H Γ a
  | allBelow : Term H Γ .nat → Term H (.nat :: Γ) .bool → Term H Γ .bool
  | anyBelow : Term H Γ .nat → Term H (.nat :: Γ) .bool → Term H Γ .bool
  | truth : Term H Γ .bool → Term H Γ .omega
  | implies : Term H Γ .omega → Term H Γ .omega → Term H Γ .omega
  | forallNat : Term H (.nat :: Γ) .omega → Term H Γ .omega
  | comprehension : Term H (a :: Γ) .omega → Term H Γ (.power a)
  | member : Term H Γ a → Term H Γ (.power a) → Term H Γ .omega
  | cell {n x} : CellTerm H n x → Term H Γ (.cell n)

def Term.eval {H Γ a} (t : Term H Γ a) (ρ : Env H Γ) : a.denote H :=
  match t with
  | .var x => x.get ρ
  | .unit => ()
  | .num n => n
  | .bit b => b
  | .arithmetic op x y => op.eval (x.eval ρ) (y.eval ρ)
  | .compare op x y => op.eval (x.eval ρ) (y.eval ρ)
  | .and x y => x.eval ρ && y.eval ρ
  | .or x y => x.eval ρ || y.eval ρ
  | .not x => !(x.eval ρ)
  | .ite c x y => bif c.eval ρ then x.eval ρ else y.eval ρ
  | .pair x y => (x.eval ρ, y.eval ρ)
  | .fst x => (x.eval ρ).1
  | .snd x => (x.eval ρ).2
  | .lam x => fun y => x.eval (y, ρ)
  | .app f x => f.eval ρ (x.eval ρ)
  | .natRec z step n => Nat.rec (z.eval ρ) (fun i x => step.eval ρ i x) (n.eval ρ)
  | .allBelow n p => (List.range (n.eval ρ)).all (fun i => p.eval (i, ρ))
  | .anyBelow n p => (List.range (n.eval ρ)).any (fun i => p.eval (i, ρ))
  | .truth b => b.eval ρ = true
  | .implies x y => x.eval ρ → y.eval ρ
  | .forallNat p => ∀ n : Nat, p.eval (n, ρ)
  | .comprehension p => fun x => p.eval (x, ρ)
  | .member x s => s.eval ρ (x.eval ρ)
  | .cell (x := x) _ => x

theorem eval_add {H Γ} (x y : Term H Γ .nat) (ρ : Env H Γ) :
    (Term.arithmetic .add x y).eval ρ = x.eval ρ + y.eval ρ := rfl

theorem eval_rec_zero {H Γ a} (z : Term H Γ a)
    (s : Term H Γ (.arrow .nat (.arrow a a))) (ρ : Env H Γ) :
    (Term.natRec z s (.num 0)).eval ρ = z.eval ρ := rfl

theorem eval_rec_succ {H Γ a} (z : Term H Γ a)
    (s : Term H Γ (.arrow .nat (.arrow a a))) (n : Nat) (ρ : Env H Γ) :
    (Term.natRec z s (.num (n+1))).eval ρ =
      s.eval ρ n ((Term.natRec z s (.num n)).eval ρ) := rfl

def infinity : Nat := 1073741823

structure Edge (H : HigherModel) (n width : Nat) where
  source : Fin n
  target : Fin n
  forward : source.val < target.val
  cost : Term H [] .nat
  costBound : cost.eval () < width
  boards : Term H [] .nat
  boardsBound : boards.eval () ≤ 65535

structure RawEdge where
  source : Nat
  target : Nat
  cost : Nat
  boards : Nat
  deriving Repr, BEq, DecidableEq
```

```

def Edge.lower {H n w} (e : Edge H n w) : RawEdge :=
  ⟨e.source.val, e.target.val, e.cost.eval (), e.boards.eval ()⟩

theorem lower_forward {H n w} (e : Edge H n w) :
    e.lower.source < e.lower.target := e.forward

theorem lower_cost {H n w} (e : Edge H n w) :
    e.lower.cost = e.cost.eval () := rfl

structure Problem (H : HigherModel) (n width : Nat) where
  positive : 0 < n
  widthPositive : 0 < width
  arithmeticSafe : n * width < infinity
  edges : List (Edge H n width)
  source : Fin n
  target : Fin n
  budget : Term H [] .nat
  budgetBound : budget.eval () ≤ 65535
  spaceSafe : n * (budget.eval () + 1) ≤ 16777216
  -- Variables, in order: source, target, cost, number of boardings.
  guard : Term H [.nat, .nat, .nat, .nat] .bool

def Problem.lower {H n w} (p : Problem H n w) : List RawEdge :=
  (p.edges.map Edge.lower).filter fun e =>
    p.guard.eval (e.source, (e.target, (e.cost, (e.boards, ()))))

inductive Route (es : List RawEdge) : Nat → Nat → Type where
  | nil (v : Nat) : Route es v v
  | cons (e : RawEdge) (included : e ∈ es) {v : Nat} :
    Route es e.target v → Route es e.source v

def Route.cost {es u v} : Route es u v → Nat
  | .nil _ => 0
  | .cons e _ tail => e.cost + tail.cost

def Route.boards {es u v} : Route es u v → Nat
  | .nil _ => 0
  | .cons e _ tail => e.boards + tail.boards

abbrev Candidate {H n w} (p : Problem H n w) :=
  {r : Route p.lower p.source.val p.target.val // r.boards ≤ p.budget.eval ()}

-- This interface accepts any Lean type of higher/topos terms, not just a
-- graph datatype. Its proof obligations make the finite extraction explicit.
structure Realization {H n w} (α : Type u) (p : Problem H n w) where
  feasible : α → Prop
  objective : α → Nat
  decode : Candidate p → α
  sound : ∀ r, feasible (decode r)
  costExact : ∀ r, objective (decode r) = r.val.cost
  complete : ∀ a, feasible a → ∃ r : Candidate p, r.val.cost = objective a

theorem minimum_transport {H n w α} {p : Problem H n w}
    (R : Realization α p) (r : Candidate p)
    (minimal : ∀ s : Candidate p, r.val.cost ≤ s.val.cost) :
    R.feasible (R.decode r) ∧
      ∀ a, R.feasible a → R.objective (R.decode r) ≤ R.objective a := by
  constructor
  · exact R.sound r
  · intro a ha
    obtain ⟨s, hs⟩ := R.complete a ha
    rw [R.costExact r, ← hs]
    exact minimal s

def emitEdge (e : RawEdge) : String :=
  "Edge { source: " ++ toString e.source ++ ", target: " ++ toString e.target ++
  ", cost: " ++ toString e.cost ++ ", boards: " ++ toString e.boards ++ " }"

def compile {H n w} (p : Problem H n w) : String :=
  let es := String.intercalate ",\n    " (p.lower.map emitEdge)
  "// Generated by Compiler.lean; do not edit.\n" ++
  "use routing_accel::{Edge, Instance, run_cli};\n" ++
  "fn main() {\n  let instance = Instance { " ++
  s!"vertices: {n}, width: {w}, source: {p.source.val}, target: {p.target.val}, budget: {p.budget.eval ()}, edges: vec![\n
    ↪  " ++
  es ++ "\n  ] };\n  run_cli(&instance);\n}\n"

def compileRealized {H n w α} {p : Problem H n w}
    (_ : Realization α p) : String := compile p

-- An actual typed higher-cell term in dimension 3; dimension is unrestricted.
def cell3 : CellTerm terminalHigher 3 () :=
  .identity (.identity (.identity (.generator (n := 0) ())))
```

```
def higherTerm : Term terminalHigher [] (.cell 3) := .cell cell3

-- A power-object term representing the internal subset {n | n < 8} of N.
def smallNaturals : Term terminalHigher [] (.power .nat) :=
  .comprehension (.truth (.compare .lt (.var .zero) (.num 8)))

def boundedExample : Term terminalHigher [] .bool :=
  .allBelow (.num 8) (.compare .lt (.var .zero) (.num 8))

example : boundedExample.eval () = true := by decide
example : smallNaturals.eval () 7 := by
  change (decide (7 < 8) : Bool) = true
  decide
example : ¬ smallNaturals.eval () 8 := by
  change ¬ ((decide (8 < 8) : Bool) = true)
  decide

def exampleEdge (u v c b : Nat) (hu : u < 6) (hv : v < 6)
    (hf : u < v) (hc : c < 32) (hb : b ≤ 65535) :
    Edge terminalHigher 6 32 :=
  ⟨⟨u, hu⟩, ⟨v, hv⟩, hf, .num c, hc, .num b, hb⟩

def exampleProblem : Problem terminalHigher 6 32 where
  positive := by decide
  widthPositive := by decide
  arithmeticSafe := by decide
  edges := [exampleEdge 0 1 0 1 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 1 2 5 0 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 2 3 3 1 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 3 5 0 0 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 0 4 12 1 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 4 5 0 0 (by decide) (by decide) (by decide) (by decide) (by decide),
    exampleEdge 0 5 20 0 (by decide) (by decide) (by decide) (by decide) (by decide)]
  source := ⟨0, by decide⟩
  target := ⟨5, by decide⟩
  budget := .arithmetic .add (.num 1) (.num 1)
  budgetBound := by decide
  spaceSafe := by decide
  guard := .and (.compare .lt (.var .zero) (.var (.succ .zero)))
    (.compare .lt (.var (.succ (.succ .zero))) (.num 32))

def exampleRealization : Realization (Candidate exampleProblem) exampleProblem where
  feasible := fun _ => True
  objective := fun r => r.val.cost
  decode := id
  sound := fun _ => True.intro
  costExact := fun _ => rfl
  complete := fun a _ => ⟨a, rfl⟩

end Transit

def main (args : List String) : IO Unit := do
  let file := args.headD "rust/generated.rs"
  IO.FS.writeFile file (Transit.compileRealized Transit.exampleRealization)
  IO.println s!"Generated {file} from a typed NNO constraint specification."
```