Certified Partial-Order Reduction and Symmetry Quotienting for Guarded Simplex Transfer Menus

Dana Edwards

January 14, 2026

Abstract

We formalize and implement a planning substrate for k-way allocation menus modeled as guarded unit transfers on the integer simplex. Naïve commutativity fails globally under guards; instead we use a *state-dependent* POR independence predicate and prove: (i) two-step commutation under stable-enabledness, (ii) trace-level adjacent swap invariance, (iii) a swap-equivalence relation with run-invariance, (iv) deterministic trace canonicalization with run- and horizon-invariance, and (v) symmetry equivariance for bucket transpositions supporting quotienting. We provide a runtime oracle matching the formal predicate, a deterministic canonicalizer with a sound oracle cache, a bounded-horizon simplex CEO planner, and a decision-quality benchmark gate that enforces a measurable crossover regime where POR becomes a net runtime win.

1 Artifacts, scope, and what is proved

Proof artifact (machine-checked). All theorems cited in this document are mechanically checked in Lean 4 in:

```
LeanProofs/CEO_SimplexPOR.lean
```

The proof bundle builds with:

```
cd LeanProofs && lake build
```

Runtime artifact (Rust). The executable counterparts are:

- oracle and semantics: `crates/mprd-core/src/tokenomics_v6/simplex_por_oracle.rs`
- canonicalization + oracle cache: `crates/mprd-core/src/tokenomics_v6/simplex_planner.rs`
- symmetry key: `crates/mprd-core/src/tokenomics_v6/simplex_symmetry_key.rs`
- bounded-horizon simplex CEO planner: `crates/mprd-core/src/tokenomics_v6/simplex_ceo.rs`

One explicitly non-proved lemma (declared as an axiom). The Lean file includes *one* independence lemma for the static "disjoint endpoints" criterion as an `axiom` (`stepOrStay[]comm[]of[]disjoint[]enab`
All other theorems cited below are `theorem`s in Lean.

2 Formal model

2.1 State, caps, and actions

We model simplex menu states as functions over Fin k. The exact Lean definitions are:

Listing 1: Core types (Lean)

```
abbrev State (k : Nat) := Fin k →Nat
abbrev Caps (k : Nat) := Fin k →Nat

structure Action (k : Nat) where
  src : Fin k
  dst : Fin k
  hne : src ≠dst
```

2.2 Enabledness and guarded step (failure-as-no-op)

Enabledness.

Listing 2: Enabled predicate (Lean)

```
def enabled {k : Nat} (caps : Caps k) (x : State k) (a : Action k) : Prop :=
  x a.src > 0 ∧x a.dst < caps a.dst
```

Guarded step semantics. We define a successful `step` (used under enabledness hypotheses), and the guarded `stepOrStay`:

Listing 3: Guarded stepOrStay (Lean)

```
def step {k : Nat} (x : State k) (a : Action k) : State k :=
  let x1 := update x a.src (x a.src - 1)
  update x1 a.dst (x a.dst + 1)

def stepOrStay {k : Nat} (caps : Caps k) (x : State k) (a : Action k) : State k :=
  if enabled caps x a then
    step x a
  else
    x
```

3 POR independence predicate and closed-form sufficient oracle

3.1 Stable-enabledness (dynamic POR independence)

We define the dynamic stable-enabledness predicate:

Listing 4: Stable-enabledness (Lean)

```
def stableEnabled {k : Nat} (caps : Caps k) (x : State k) (a b : Action k) : Prop :=
  enabled caps x a ∧enabled caps x b ∧
    enabled caps (stepOrStay caps x a) b ∧enabled caps (stepOrStay caps x b) a
```

3.2 Closed-form sufficient condition (stableEnabledIneq)

Listing 5: Closed-form sufficient oracle (Lean)

```
def stableEnabledIneq {k : Nat} (caps : Caps k) (x : State k) (a b : Action k) : Prop :=
  enabled caps x a ∧
  enabled caps x b ∧
  (b.src = a.src →x a.src > 1) ∧
  (b.dst = a.dst →x a.dst + 1 < caps a.dst) ∧
  (a.src = b.src →x b.src > 1) ∧
  (a.dst = b.dst →x b.dst + 1 < caps b.dst)
```

Listing 6: Oracle implies stable-enabledness (Lean)

```
theorem stableEnabled_of_stableEnabledIneq
    {k : Nat} (caps : Caps k) (x : State k) (a b : Action k) :
    stableEnabledIneq caps x a b →stableEnabled caps x a b := by
  ...
```

4 Two-step commutation and trace-level swap

Listing 7: Two-step commutation (Lean)

```
theorem stepOrStay_comm_of_stableEnabled
    {k : Nat} (caps : Caps k) (x : State k) (a b : Action k) :
    stableEnabled caps x a b →
      stepOrStay caps (stepOrStay caps x a) b = stepOrStay caps (stepOrStay caps x b) a :=
           by
  ...
```

Listing 8: Oracle implies commutation (Lean)

```
theorem stepOrStay_comm_of_stableEnabledIneq
    {k : Nat} (caps : Caps k) (x : State k) (a b : Action k) :
    stableEnabledIneq caps x a b →
      stepOrStay caps (stepOrStay caps x a) b = stepOrStay caps (stepOrStay caps x b) a :=
           by
  ...
```

Listing 9: Trace run and adjacent-swap lemma (Lean)

```
def run {k : Nat} (caps : Caps k) : List (Action k) →State k →State k
  | [], x => x
  | a :: as, x => run caps as (stepOrStay caps x a)

theorem run_swap_adjacent_of_stableEnabledIneq
    {k : Nat} (caps : Caps k) (pre suf : List (Action k)) (x : State k) (a b : Action k) :

    stableEnabledIneq caps (run caps pre x) a b →
      run caps (pre ++ (a :: b :: suf)) x = run caps (pre ++ (b :: a :: suf)) x := by
  ...
```

5 Swap-equivalence (SwapEq) and run invariance

Listing 10: SwapEq and run invariance (Lean)

```
inductive SwapEq {k : Nat} (caps : Caps k) (x0 : State k) : List (Action k) →List (
    Action k) →Prop where
  | refl (xs : List (Action k)) : SwapEq caps x0 xs xs
  | step {xs ys : List (Action k)} : SwapStep caps x0 xs ys →SwapEq caps x0 xs ys
  | trans {xs ys zs : List (Action k)} : SwapEq caps x0 xs ys →SwapEq caps x0 ys zs →
      SwapEq caps x0 xs zs
  | symm {xs ys : List (Action k)} : SwapEq caps x0 xs ys →SwapEq caps x0 ys xs

theorem run_invariant_of_SwapEq
    {k : Nat} (caps : Caps k) (x0 : State k) :
    ∀{xs ys : List (Action k)}, SwapEq (k := k) caps x0 xs ys →run caps xs x0 = run caps
        ys x0 := by
  ...
```

6 Canonicalization and bounded-horizon completeness

Listing 11: Canonicalization preserves run and length (Lean)

```
def canonicalize {k : Nat} (caps : Caps k) (x0 : State k) (xs : List (Action k)) : List (
    Action k) :=
  canonIter (k := k) caps x0 (xs.length * xs.length) xs

theorem run_canonicalize_eq
    {k : Nat} (caps : Caps k) (x0 : State k) (xs : List (Action k)) :
    run caps (canonicalize (k := k) caps x0 xs) x0 = run caps xs x0 := by
  ...

theorem length_canonicalize
    {k : Nat} (caps : Caps k) (x0 : State k) (xs : List (Action k)) :
    (canonicalize (k := k) caps x0 xs).length = xs.length := by
  ...
```

Listing 12: Reachability completeness under canonicalize (Lean)

```
def ReachableWithin {k : Nat} (caps : Caps k) (x0 : State k) (h : Nat) (x : State k) :
    Prop :=
  ∃xs : List (Action k), xs.length ≤h ∧run caps xs x0 = x

theorem reachableWithin_via_canonicalize
    {k : Nat} (caps : Caps k) (x0 : State k) (h : Nat) (x : State k) :
    ReachableWithin (k := k) caps x0 h x ↔
      ∃xs : List (Action k), xs.length ≤h ∧run caps (canonicalize (k := k) caps x0 xs) x0
          = x := by
  ...
```

7 Benchmark gate

```
bash tools/ceo/check_ceo_simplex_rail.sh
```

The rail blocks regressions by requiring, on sweep rows with `eval_iters` ≥ 200, POR win-rate ≥ 0.75 and median runtime ratio ≤ 1.0, implemented by `tools/ceo/check_ceo_simplex_sweep_strict.py`.