1 Introduction

This is a small example of a ledger specification for Midnight. It only assumes that we know how to hash some types and provides a step function for the ledger together with a proof of a non-trivial property.

2 Epochs & Consensus

We define the number of slots in an epoch and a conversion function from slots to epochs here, as well as a structure to refer to certain points in the chain.

```
record Point : Set where
  field slot      : Maybe ℕ
        blockHash : Hash

slotsInEpoch : ℕ
slotsInEpoch = 50

epochOf : Maybe ℕ → Maybe ℕ
epochOf nothing = nothing
epochOf (just s) = just (s / slotsInEpoch)
```

Figure 1: Point- and epoch-related definitions

3 Transactions & Blocks

Transactions can increment or decrement a counter that is stored in the ledger, and blocks consist of a list of transactions and a header, which contains some meta-information about the block. We also use the accessor functions of the headers transitively, to access fields in the header given the body.

```
data Tx : Set where
  inc : Tx
  dec : Tx

txDelta : Tx → ℤ
txDelta inc = 1ℤ
txDelta dec = -1ℤ
```

Figure 2: Transactions

```
record Header : Set where
  field slotNo    : ℕ
        blockNo   : ℕ
        blockHash : Hash
        prev      : ℕ
        nodeId    : ℕ

record Block : Set where
  field header : Header
        body   : List Tx

  open Header header public

blockPoint : Block → Point
blockPoint b = record { slot = just slotNo ; blockHash = blockHash }
  where open Block b

computeBlockHash : Block → Hash
computeBlockHash b = hash (slotNo , blockNo , prev , body)
  where open Block b

addBlockHash : Block → Block
addBlockHash b = record b { header = record header { blockHash = computeBlockHash b } }
  where open Block b
```

Figure 3: Blocks and functions related to them

4 Ledger

The ledger state consists of a pointer to the previous block as well as a counter and two snapshots. Ticking a ledger state means to roll over the snapshots.

```
record LedgerState : Set where
  field tip                 : Point
        count               : ℤ
        snapshot1 snapshot2 : ℤ

tickLedgerState : ℕ → LedgerState → LedgerState
tickLedgerState newSlot st = if isNewEpoch
  then record st { snapshot1 = count st ; snapshot2 = snapshot1 st }
  else st
  where isNewEpoch = epochOf (Point.slot (tip st)) <ᵇ epochOf (just newSlot)
```

Figure 4: Ledger state and its tick function

The ledger transition system updates the tip to point to the most recent applied block, and increments/decrements the counter according to the transactions. We also want to check some non-trivial conditions, so we require the change to the counter to be non-zero and that the hash in the header is correct.

```
data _⊢_⇀⦇_,LEDGER⦈_ : ⊤ → LedgerState → Block → LedgerState → Set where
  LEDGER-inductive : ∀ {Γ} {s} {b} →
    let open Block b
        acc = Σˡ[ x ← body ] txDelta x
        s' = tickLedgerState slotNo s
    in • acc ≢ 0ℤ • computeBlockHash b ≡ blockHash
    ────────────────────────────────
    Γ ⊢ s ⇀⦇ b ,LEDGER⦈
      record s' { tip = blockPoint b ; count = count s + acc }
```

Figure 5: The LEDGER transition system

We also have automation that generates a step function for this relation. As an example, we also provide a function that does the same computation explicitly.

```
LEDGER-step : ⊤ → LedgerState → Block → Maybe LedgerState
LEDGER-step = compute

applyBlockTo : Block → LedgerState → Maybe LedgerState
applyBlockTo b st = let acc = Σˡ[ x ← Block.body b ] txDelta x in
  ifᵈ acc ≢ 0ℤ ∧ computeBlockHash b ≡ Block.blockHash b
    then just record st { tip = blockPoint b ; count = count st + acc }
    else nothing
```

5 Key properties

This is Agda, so we can prove some properties. Since we check very few things, and just repeating properties that we do check isn't interesting, here's the only non-trivial property that I could think of: That the counter in the ledger state does change.

We present three proofs here, one using the STS, one using the step function generated from the STS, and one using the applyBlockTo function to show why using the STS is more convenient for proofs: Instead of tracing the function control flow to extract properties, we can simply pattern-match for them. After we obtained fact that the sum of changes induced by transactions is non-zero, the proofs are identical.

```
lemma : ∀ {x y} → y ≢ 0ℤ → x ≢ x + y
lemma y≢0 eq = y≢0 (identityʳ-unique _ _ (sym eq))

LEDGER-property₁ : _ ⊢ s ⇀⦇ b ,LEDGER⦈ s' → count s ≢ count s'
LEDGER-property₁ (LEDGER-inductive acc≢0 _) = lemma acc≢0

LEDGER-property₂ : LEDGER-step _ s b ≡ just s' → count s ≢ count s'
LEDGER-property₂ {s} {b} eq with Equivalence.to (≡-just⇔STS {s = s} {sig = b}) eq
... | (LEDGER-inductive acc≢0 _) = lemma acc≢0

LEDGER-property₃ : applyBlockTo b s ≡ just s' → count s ≢ count s'
LEDGER-property₃ {b = b} h with
  (Σˡ[ x ← Block.body b ] txDelta x) ≟ 0ℤ | computeBlockHash b ≟ Block.blockHash b | h
... | no acc≢0 | yes _ | refl = lemma acc≢0
... | no _  | no _ | ()
... | yes _ | _     | ()
```

Figure 6: Three proofs that the counter doesn't stay constant