HaskLedger

HL-P01 · PAPER · V1.0.0

HASKLEDGER: CARDANO CONTRACTS AS A HASKELL EDSL

Vinit K Inamke · Tech Lead, Konma Labz
vinit@konma.io ORCID 0009-0009-5112-1711 github.com/VINIT-INAMKE
linkedin.com/in/vinit-inamke
Release v1.0.0 · September 2026

ABSTRACT

HaskLedger is a Haskell embedded domain-specific language for Cardano validators and minting policies. Contracts are written in a `Contract` monad, and running one builds a typed graph in Covenant, the intermediate representation from MLabs. The c2uplc code generator turns that graph into Untyped Plutus Lambda Calculus, and HaskLedger writes a PlutusV3 envelope. This paper covers the language, the compilation path and why it uses only Plutus builtins, how contracts read on-chain data, the security guards, testing off-chain and on the Preview testnet, measured costs against PlutusTx, and the current limits.

CONTENTS

01 Introduction
02 The eDSL
03 Compilation
04 Plutus Data on-chain
05 Security guards
06 Testing
07 Measurement
08 Limitations
09 Related work
10 Future work

01 INTRODUCTION

A Cardano script sees one thing: the transaction trying to spend a UTxO or mint a token, handed to it as a `ScriptContext`. It accepts by returning and rejects by failing. On-chain it is a program in Untyped Plutus Lambda Calculus (UPLC), and every byte and every step of it is paid for in fees.

HaskLedger lets you write these scripts in Haskell, with `do`-notation, integer literals and infix operators such as `.==`, `.&&` and `.>=`. It does not generate Plutus itself. It builds a graph in Covenant, an intermediate representation (IR) developed by MLabs, hands the graph to c2uplc for code generation, and writes the result as a PlutusV3 text envelope that `cardano-cli` reads.[1]

```
guardedDeadline :: Validator
guardedDeadline = validator "guarded-deadline" $ do
  requireAll
    [ ("correct redeemer", asInt theRedeemer .== 42)
    , ("past deadline",    txValidRange `after` 1769904000000)
    ]
```

Underneath, `after` walks the validity interval, its lower bound, the time inside it and the flag saying whether the bound is closed. The contract states the rule; the combinator deals with the encoding.

Three design choices shape the library, and the sections below argue for each:

- Contracts read the transaction straight from Plutus Data and touch only the fields they use. There is no decoding step and no runtime library, which is most of why the scripts are small.
- Branching uses Plutus builtins with explicit `delay` and `force` rather than Covenant's typed `match` and `ctor'`, which c2uplc does not yet compile correctly in every case HaskLedger needs.
- Values carry the lambda depth at which they were built, plus a recipe to rebuild them, so a datum field used inside a list predicate still refers to the right variable.

HaskLedger v1.0.0 ships thirteen example contracts, spending validators and one minting policy, each run on the Cardano Preview testnet (PlutusV3, Conway era) with transactions that should pass and, where the contract has something to refuse, transactions that should be refused.[2] It is released under the Apache 2.0 licence, and it is young: datums are read by field index, `.&&` and `.||` evaluate both sides, the payout guards count lovelace only, and there is no CIP-57 blueprint output.

1 Covenant and c2uplc are both vendored in the HaskLedger repository, and the vendored c2uplc carries local fixes.

2 always-succeeds has nothing to refuse, and the recorded run skipped token-gate's refused case.

02 THE EDSL

Validators and the Contract monad

A contract is a `Validator`, made from a name and a body. `validator` builds a spending validator and `mintingPolicy` a minting policy. They have the same type; the second exists so the intent shows. The body has type `Contract Expr` and ends in

one of three forms: `require label check`, which fails the script when the check is false; `requireAll`, which tests a list of labelled checks in order and fails on the first false one; or `pass`, which accepts.[3]

`Contract` is Covenant's graph builder, `ASGBuilder`, with the current lambda depth carried in a reader:

```
newtype Contract a = Contract (ReaderT Depth ASGBuilder a)

data Expr = Expr
  { exprLevel  :: Depth          -- lambda depth where exprRef
is valid
  , exprRef    :: Ref            -- the Covenant node, valid at
exprLevel
  , exprRecipe :: Contract Expr  -- how to rebuild it at any
other depth
  }
```

Running a body evaluates nothing on-chain; it adds nodes to a graph. An `Expr` is one of those nodes plus the recipe that built it, for reasons the Compilation section explains.

Expressions and binding

Most combinators take and return `Contract Expr`, a description of how to compute a value. Results can be named with `<-`, which gives a bare `Expr`, and passed back with `pure`; a `let` names a computation without running it. Both give the same script, because identical computations become one node. The vesting contract has the usual shape: take the datum apart, then state the rules.

```
vesting :: Validator
vesting = validator "vesting" $ do
  datum <- theDatum
  dFields <- unconstrFields datum
  beneficiary <- nthField 0 dFields
  deadline <- nthField 1 dFields
  let outputs = asList txOutputs
  requireAll
    [ ("past vesting deadline",         txValidRange `after`
asInt (pure deadline))
    , ("signed by beneficiary",         signedBy (pure
beneficiary))
    , ("pays full amount to beneficiary", paysAtLeast outputs
(pure beneficiary) ownValue)
    , ("single script input",           singleOwnScriptInput)
    ]
```

The monadic arguments are deliberate. A combinator that takes a `Contract Expr` can run its argument wherever it needs it, including inside a lambda it builds itself, and the user can write `asInt theRedeemer .== 42` without binding anything first. Integer literals, `+`, `-` and `*` come from a `Num` instance for `Contract Expr`.

Every value is either Plutus Data or a plain builtin value, and the programmer has to know which. `asInt`, `asByteString`, `asList` and `asMap` unwrap Data, and `mkIntData` and `mkByteStringData` wrap it. `.==` and `.>=` compare integers,

3 Labels are not written into the script, so a refused transaction does not say which `require` failed.

`equalsData` compares Data as it is, and `equalsByteString` compares bytes. Checks have the type `Contract Condition`, but `Condition` is another name for `Expr`: it records intent and enforces nothing.

What the library covers

The public API is what `import HaskLedger` exports: accessors for all 16 `TxInfo` fields and for inputs and outputs, `after` and `before`, `signedBy` and signature verifiers, `valueOf` and minting helpers, payout guards, list functions such as `anyList` and `countList`, the lazy `case` functions, hashing and BLS12-381 operations, and `traceMsg`. Internally the modules are layered, from the core `Contract` module and the builtin lifters up to the `Validator` kit and the `Compile` back end.

Parameters fixed at compile time

A Haskell function that returns a `Validator` gives a family of scripts. Its arguments are baked into the compiled script, so each set has its own script hash, and so its own address or policy id. The one-shot NFT policy, `oneShotNFT :: ByteString -> Integer -> Validator`, takes the seed UTxO it must consume this way. Values that must never change suit compile-time parameters; values that differ from one lock to the next belong in the datum.

03 COMPILATION

Four stages

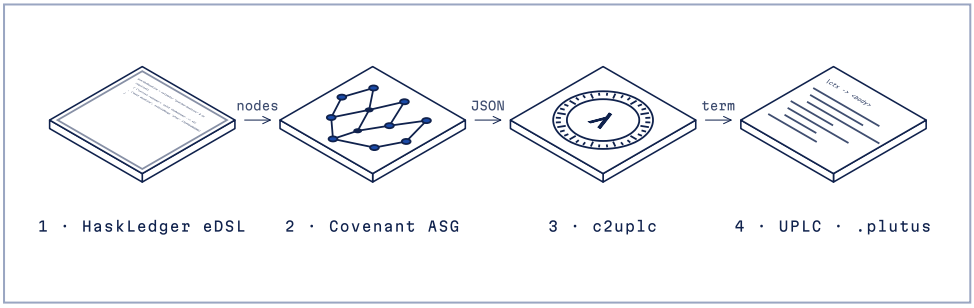


FIGURE · PIPELINE A contract goes from the Haskell eDSL to a Covenant graph, through c2uplc to UPLC, and out as a PlutusV3 envelope.

1. **The contract builds a graph**, a Covenant abstract syntax graph (ASG) of function nodes, builtin calls, literals and arguments.
2. **Covenant checks it**, type-checking each node as it is added, so a builtin given the wrong number of arguments fails at compile time.
3. **c2uplc generates UPLC** from the graph.
4. **HaskLedger packages the term.** It renames variables so every binding is unique, converts names to de Bruijn indices, serialises the program and writes a `PlutusScriptV3` text envelope.

`compileToEnvelope` runs all four stages in memory. `compileToJSON` stops after the second and writes the graph as JSON, for inspection or for another Covenant back end.[4] The v1.0.0 toolchain is GHC 9.12.2, Covenant 1.3.0, c2uplc 1.0.0 with local fixes, and plutus-core 1.51.0.0. A different version of any of them can change the generated script, and with it the script hash, the address and the policy id.

4 That split is the reason to go through an IR: back-end improvements, such as support for new Plutus versions, can reach HaskLedger contracts without changes to HaskLedger.

Strictness and delayed branches

Every validator compiles to a single lambda that takes the `ScriptContext` and either returns unit or fails. UPLC is call-by-value, and builtins such as `IfThenElse` and `ChooseData` take all their branches as arguments. Left alone, every branch of every conditional would run, and for `require`, whose failure branch calls `error`, that would be fatal. HaskLedger delays the branches that must not run. Each one is a one-argument function wrapped in `delay`. The builtin picks one of them, `force` unwraps it, and it is applied to the value being matched.

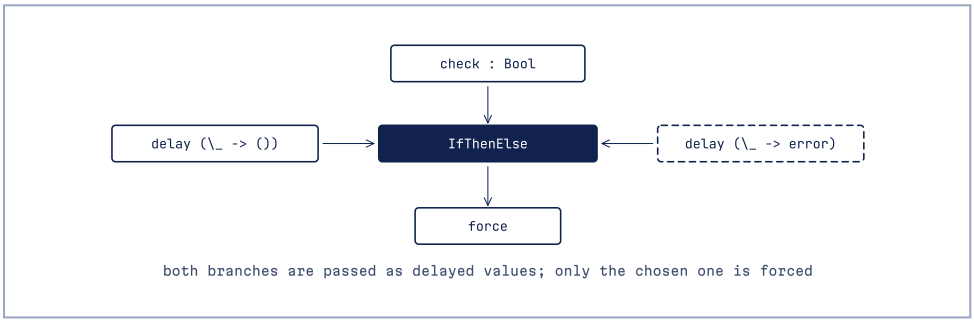


FIGURE · LAZINESS IN REQUIRE A require delays its failure branch, the builtin picks one of the two delayed branches, and only that one is forced and run.

The branch receives the value as its own argument instead of reaching into the enclosing scope, which keeps its variable references simple. `require`, `requireAll` and the `case` functions work this way, and only the branch taken runs. `ifThenElse`, `.&&`, `.||`, `chooseList` and `chooseData` are plain builtin applications, so all of their arguments run.

Sharing

Covenant hash-conses the graph: when a combinator builds a node that already exists, it gets the existing node back. A contract that reads `theDatum` in five places has one node for it, and c2uplc binds shared nodes with a `let`, so the value is computed once in that scope.

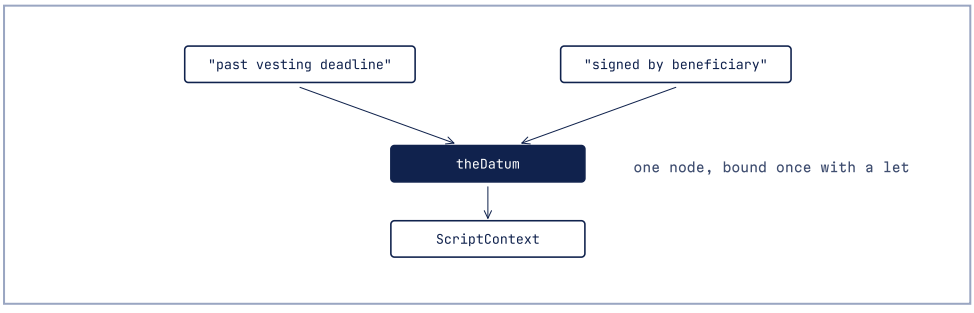


FIGURE · SHARING Covenant hash-conses the graph, so two uses of the same computation point at one shared node.

Variable references across lambdas

UPLC names a variable by its de Bruijn index, the argument of the lambda so many levels up. That is the hard part of an embedded language that nests functions, because a value built at one depth is often used inside a lambda at another. In the multisig contract, datum fields bound at the top are used inside the predicate given to `countList`, which becomes a fold's step function:

```
multisig :: Validator
multisig = validator "multisig-2of3" $ do
  datum <- theDatum
  dFields <- unconstrFields datum
  threshold <- nthField 0 dFields
  s1 <- nthField 1 dFields
  s2 <- nthField 2 dFields
  s3 <- nthField 3 dFields
  let sigs = asList txSignatories
  let count = countList
        (\sig -> equalsData sig (pure s1) .|| equalsData sig
(pure s2) .|| equalsData sig (pure s3))
        sigs
  require "enough authorized signers" $ count .>= asInt (pure
threshold)
```

An earlier design represented an expression as a bare Covenant reference. Covenant's `arg` resolves against the scope at construction time, so a reference built at the validator's depth and reused inside a lambda silently meant that lambda's own argument. The contract tests caught it: in the dumped UPLC for multisig, the chain meant to unwrap the script context was unwrapping the fold's accumulator.[5]

The fix is the `Expr` record. `resolve` returns the cached node when a value is used at the depth where it was built, and reruns the recipe anywhere else. Arguments carry the depth of the lambda that owns them, a de Bruijn level, and each use computes the index as the current depth minus the owner's. `withLam` wraps Covenant's `lam`, increments the depth inside the body and hands the body correctly levelled arguments.

Rerunning a recipe is safe because the builder's effects are hash-consed inserts: at the same depth it returns the existing nodes, and at a deeper one it emits correctly shifted references. Two costs remain: compilation retraverses a recipe once per expression and depth, and a value used inside a list function is recomputed inside the loop body. The `spike-sz` suite checks references at several depths through the whole pipeline.

c2uplc can also give two bindings the same internal name when hash-consed code is reused across scopes, so HaskLedger renames every lambda-bound variable before converting to de Bruijn indices. The vendored c2uplc carries four local patches to its scope handling as well.

Why builtins only

Covenant also offers typed pattern matching and constructors, `match` and `ctor'`, and lazy lambdas through `lazyLam`. c2uplc compiles these through a separate transformation stage that does not yet produce correct code in every case HaskLedger

5 Every contract test that failed at the time had this cause: a user predicate capturing a datum or context value inside a fold step or a branch lambda.

needs. HaskLedger therefore uses only Plutus builtins, lambdas, `delay` and `force`, and Covenant's `cata`, which all go through c2uplc's direct code generation path. The one exception is the empty list, `ctor "List" "Nil"`, which c2uplc handles specially.

Most combinators are a builtin applied to arguments fetched with `resolveM`, wrapped in `expr` so the result carries its recipe. Branches are lambdas built with `withLam`, delayed with `thunk` and selected with `force`. The cost is no typed matching on datums through Covenant today; the gain is that everything HaskLedger emits goes through the one code path its tests exercise.

Folds

The list functions use Covenant's `cata`, a structural fold that c2uplc compiles to a recursive function walking the list once. They are right folds with no early exit, so `anyList` visits every element even after a match, and their accumulators are Data. Covenant's type checker and c2uplc's code generator also disagree about the order of a `cata` handler's arguments, and agree only when element and accumulator have the same type.[6] So HaskLedger's folds keep the accumulator in the element type, and never nest one fold inside another over a different element type.

6 This was reported to the Covenant maintainers with a minimal reproduction.

04 PLUTUS DATA ON-CHAIN

A contract reads its transaction by walking Plutus Data, a tree of five kinds of node: `I` for an integer, `B` for bytes, `List`, `Map`, and `Constr`, which holds a constructor tag and a list of fields.

Constructors and fields

Reading field `n` of a `Constr` means `unConstrData` to unwrap it, `sndPair` to take the field list, `tailList` applied `n` times, and `headList` for the element.

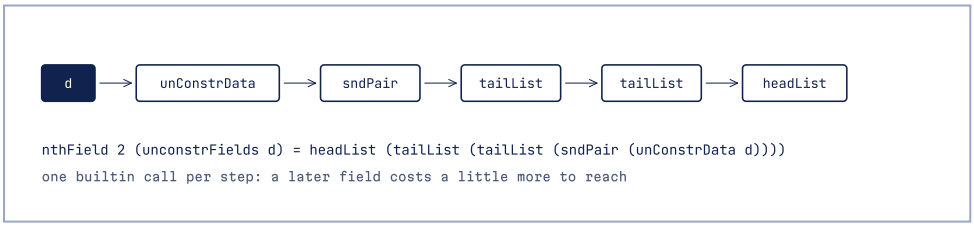


FIGURE · READING A FIELD Reading field n of a Constr takes one unConstrData, one sndPair, n calls to tailList and a final headList.

Each step is one builtin call, so a later field costs a little more to reach than an earlier one. In HaskLedger this is `unconstrFields` followed by `nthField n`, counting from 0.

The ledger types

Types that only wrap bytes or a number, such as `PubKeyHash`, `CurrencySymbol`, `TokenName`, `POSIXTime` and `Lovelace`, are stored bare as `B` or `I`. Records and sum types are `Constr` nodes:

- `ScriptContext` is `Constr 0 [txInfo, redeemer, scriptInfo]`, and a spending `ScriptInfo` is `Constr 1 [txOutRef, maybeDatum]`.

- `TxInfo` has 16 fields in the Plutus V3 order. Inputs are field 0, outputs field 2, the mint field 4, the validity range field 7 and the signatories field 8.
- `TxOut` is `Constr 0 [address, value, outputDatum, maybeReferenceScript]`. An address starts with its credential, `Constr 0 [keyHash]` for a key or `Constr 1 [scriptHash]` for a script.
- `OutputDatum` is `Constr 0 []` for none, `Constr 1 [hash]` for a hash and `Constr 2 [datum]` for an inline datum.
- `Value` is a `Map` from currency symbol to a `Map` from token name to an `I` quantity. ADA uses the empty symbol and the empty name.
- A validity bound is `Constr 0 [extended, closed]`, where `Extended` is `Constr 1 [I time]` for a finite time, `Constr 0 []` for minus infinity and `Constr 2 []` for plus infinity.
- `Maybe` is `Constr 0 [x]` for `Just` and `Constr 1 []` for `Nothing`.

How the combinators use it

`theDatum` reads field 1 of the spending `ScriptInfo` and takes the value out of `Just`. `after` requires the lower bound of the validity interval to be finite and treats an open bound as one millisecond later than a closed one; `before` reads the upper bound. `paysAtLeast` reads an output's credential, then its hash. `inlineDatumEquals` builds `Constr 2 [datum]` and compares it with the output's datum field in one `equalsData` call. Times are POSIX milliseconds: `1769904000000` is 2026-02-01 00:00 UTC.[7]

7 The transaction has to set the bound being checked, with `--invalid-before` for `after` and `--invalid-hereafter` for `before`. Without it the check fails, which is the safe way round.

Reading only what is needed

This is where the small scripts come from. Idiomatic PlutusTx decodes the whole script context into Haskell types before the contract's logic runs. A HaskLedger contract reads only the fields it touches, a few builtin calls each: guarded-deadline reads the redeemer and one bound of the validity range. With no runtime library, no traces unless asked for, and shared computations bound once, little is left beyond the checks themselves.

The price is that the programmer, not the compiler, knows what each field holds. A wrong index, or `asInt` applied to bytes, fails when the script runs rather than when it compiles.

05 SECURITY GUARDS

A contract guards real value, cannot be patched once funds are locked, and runs against transactions built by people who want it to fail open. It can only refuse to approve a transaction, not stop one being built. The attacker controls the redeemer, the inputs (including other UTxOs under the same script), the outputs, what is minted, the validity range within what the node accepts, and which keys sign. The attacker does not control the datum of a UTxO already locked, signatures from keys they do not hold, or the fact that a UTxO can be spent only once. Checks must rest on that second list: the redeemer may select an action, never grant permission.

A security pass over the examples found four holes, all closed with guards that now ship in the library. Each example's source opens with a comment stating what it guarantees and what it does not.

Datum hijack. When funds continue at the script address, the transaction builder picks the continuing output's datum. A treasury "deposit" could put the attacker's key in the datum in place of the admin's, and the attacker could then withdraw everything. A deposit now requires `inlineDatumEquals continuingOutput (pure admin)`:

```
inlineDatumEquals :: Contract Expr -> Contract Expr -> Contract
Condition
inlineDatumEquals outM datM =
  equalsData (txOutDatum outM) (constrData (mkInt 2) (consList
datM mkNil))
```

It builds the inline-datum wrapper and compares whole values instead of taking the output's datum apart. Under strict evaluation every branch runs on every transaction, so a guard that failed on an unexpected shape, such as an output with no datum, would fail the whole script. Building and comparing is total. Where the datum is meant to change, as in the oracle, the operator's signature gates the change.

Double satisfaction. Two UTxOs under the same script can be spent in one transaction. If each validator only checks that some output pays the seller, one payout satisfies both and the attacker keeps the second pot. `singleOwnScriptInput` requires exactly one input from the script's own address. Escrow, vesting, treasury, oracle and token-gate all use it.

Dust payouts. "Some output pays the beneficiary" is satisfied by an output of 1 lovelace. `paysAtLeast outputs pkh amount` sums the lovelace paid to that credential across all outputs and compares the total with the amount, usually `ownValue`, the lovelace being spent. Its fold uses the strict `ifThenElse`, so `lovelaceOf` runs on every output, which is safe because every value has an ADA entry.

Token smuggling. `mintedAmount tn .== 1` checks one token name, and the same transaction could mint any number of other names under the same policy. The one-shot policy also requires `ownMintTokenCount .== 1`, on burns as well as mints. Its seed UTxO is baked in at compile time and must be spent, and a UTxO can be spent only once, so the policy can mint only once. Reading the seed from the redeemer would let any UTxO do.

Some risks are documented rather than closed. A hash lock's preimage can be copied from the mempool, so a secret meant for one party also needs that party's signature. The ledger removes duplicate required signers, so a multisig datum that lists one key twice turns a 2-of-3 into a 2-of-2.[8] And the payout guards count lovelace only, so a contract that holds native tokens must check each one with `valueOf`.

8 Checking that the keys are distinct is left to off-chain code, before anything is locked.

06 TESTING

The rule the examples follow is that a contract is not tested until it has been seen to refuse the transactions it is meant to refuse.

Off-chain

An off-chain test builds a `ScriptContext` by hand as Plutus Data, compiles the validator, applies it and runs it on the evaluator the ledger uses. The result is success or failure exactly as on-chain, minus the node, in milliseconds. Here are the tests for a

small owner lock that expects the owner's key hash as its datum:

```
tests :: TestTree
tests = testGroup "OwnerLock"
  [ testCase "owner can spend" $
      assertEvalSuccess "owner" $ evalValidator ownerLock
(signedWith [owner])
  , testCase "stranger cannot spend" $
      assertEvalFailure "stranger" $ evalValidator ownerLock
(signedWith [stranger])
  , testCase "unsigned transaction fails" $
      assertEvalFailure "unsigned" $ evalValidator ownerLock
(signedWith [])
  , testCase "owner among several signers" $
      assertEvalSuccess "among" $ evalValidator ownerLock
(signedWith [stranger, owner])
  ]
```

Here `signedWith` builds its context with `mkTxInfoWithFields [(8, List (map B keys))]`, which sets field 8, the signatories, and leaves the rest empty. Other builders cover inputs, outputs, addresses, values, inline datums, minting contexts and validity bounds.

For each action a contract allows, the examples test that it succeeds, then that it fails when each rule is broken: the wrong key signs, or nobody does; the payout is 1 lovelace short or goes elsewhere; a second UTxO from the same script is spent; the validity range falls just either side of a deadline, or is missing; the redeemer names an action that does not exist; and for minting, the wrong quantity, a second token name, or a burn disguised as a mint. Each hole from the security pass has attack tests that must fail validation. Every combinator also has a test for the right result and, if it checks a value, one for a wrong value, since a test that compares a value with itself proves nothing.

There are six suites and a regression spike, `spike-sz`. Five cover the library, and `haskledger-contracts` covers the examples with their attack cases. The helpers live in `haskledger/test/TestHelper.hs`, in the test suite rather than the library.[9]

On the Preview testnet

The deploy scripts in `haskledger/deploy/` lock funds under each example, spend them with a valid transaction, then try invalid ones and check that `cardano-cli transaction build` reports a script failure. The logs kept in the repository record 28 test cases across the thirteen contracts. One more, token-gate's refused case, was skipped in the recorded run because the wallet held no UTxO without the gate token to spend from. A refused transaction never reaches the chain, because the node evaluates the script while building it. The evidence for a negative test is the `Script evaluation error` message, not a transaction hash, and it counts only if that message is what failed the build: a missing input or an unsynced node fails too, and proves nothing.

07 MEASUREMENT

Script size and execution cost are paid in fees and cap how many script transactions fit in a block.

9 Other projects can copy the file. It is Apache 2.0 licensed and depends only on `haskledger`, `covenant`, `c2uplc`, `plutus-core`, `tasty-hunit`, `bytestring` and `vector`.

Method

The harness in `haskledger/bench` needs no node. Size is the serialised script, the flat-encoded bytes a transaction witness carries. CPU steps and memory come from the Plutus evaluator with the default cost model of plutus-core 1.51.0.0, the one the chain charges with, and are the execution units a node reports. Inputs are the positive cases run on Preview, rebuilt as minimal script contexts, the same for both toolchains. The baseline is the same contracts written the usual PlutusTx way and compiled with the PlutusTx plugin; its compiled scripts are committed, so the comparison reruns without that toolchain.

Only five contracts have a PlutusTx baseline, all small teaching contracts: always-succeeds, redeemer-match, deadline, guarded-deadline and hash-lock. The application contracts have HaskLedger figures only.

Results

CONTRACT	SCRIPT BYTES		CPU STEPS		MEMORY	
always-succeeds	161		976,100		6,200	
	2,533	15.7×	25,561,498	26.2×	101,575	16.4×
redeemer-match	192		1,984,619		9,662	
	2,545	13.3×	25,794,575	13.0×	102,608	10.6×
deadline	372		10,710,190		32,383	
	3,258	8.8×	30,778,058	2.9×	132,941	4.1×
guarded-deadline	407		11,860,171		36,349	
	3,269	8.0×	31,118,972	2.6×	134,375	3.7×
hash-lock	223		3,833,672		14,082	
	2,561	11.5×	26,528,932	6.9×	105,077	7.5×

In each cell, HaskLedger on the first line and idiomatic PlutusTx below it, same contract and inputs, plutus-core 1.51.0.0 cost model. Ratios are PlutusTx over HaskLedger.

Ratios are PlutusTx over HaskLedger. The PlutusTx scripts were 8.0 to 15.7 times the size of the HaskLedger ones, and used 2.6 to 26.2 times as many CPU steps and 3.7 to 16.4 times as much memory. The ratios are largest for the contracts that do least and smallest for the two deadline contracts, whose own work of walking the validity interval is a bigger share of the total. That fits the explanation that most of the gap is the decoding step, which PlutusTx pays whatever the contract checks.

Real transactions carry larger contexts, which cost more under both toolchains. The PlutusTx decoding step grows faster with context size, so minimal contexts should understate the gap rather than overstate it, though full-size contexts have not been measured. Oracle and treasury, which walk the input and output lists, are the most expensive contracts in the report, and still use a small fraction of a transaction's memory limit. The report's script fee is only the execution part: memory and steps, each times its price at Preview's protocol parameters.[10]

10 The full transaction fee adds a fixed part and a per-byte part. A script carried in the transaction witness counts toward the bytes.

Scripts per block

CONTRACT	HASKLEDGER	PLUTUSTX
always-succeeds	10,000	610
redeemer-match	6,416	604
deadline	1,867	466
guarded-deadline	1,686	461
hash-lock	4,402	590

Validations that fit in one block's execution budget if each carries one script execution. Execution budget only; block size and other limits are not modelled.

This table gives how many executions of each script fit in one block's execution budget of 62,000,000 memory units and 20,000,000,000 CPU steps, whichever runs out first. It is the execution-budget bound in isolation, not a claim about protocol throughput. For small transactions the block body size of 90,112 bytes runs out before the execution budget does.

HaskLedger has not been benchmarked against Aiken or Plutarch.

Reproducing the numbers

Inside the Nix dev shell:

```
cabal run haskledger-bench                          # built-in
Preview parameters
cabal run haskledger-bench -- pparams.json          # your own
protocol parameters
cabal run haskledger-bench > bench-results.md       # keep the
report
```

The PlutusTx baseline has its own toolchain, GHC 9.6 with the PlutusTx plugin, and can be rebuilt with `cabal run baseline-gen` in `haskledger/bench/baseline-plutustx`.

08 LIMITATIONS

Several follow directly from the design choices above.

- **No typed datums.** Fields are read by index and converted by hand. A mistake is a run-time failure, not a compile error, and the defence is a test for every datum shape.
- **No short-circuit on booleans.** `.&&`, `.||` and `ifThenElse` evaluate both sides. Checks stay correct, but every branch runs, so a transaction must meet every branch's shape requirements: escrow's refund branch reads the upper bound even during a claim, so every escrow transaction needs both `--invalid-before` and `--invalid-hereafter`. The `case` functions help when a branch must not run.
- **No early exit.** List functions visit every element, so cost grows with list length.
- **Payout guards count lovelace only.** Native tokens riding along are not checked.

- **Test helpers are not part of the library.** They have to be copied to be used elsewhere.
- **No blueprint output.** HaskLedger writes `.plutus` envelopes, not CIP-57 `plutus.json` blueprints.
- **Parameters mean a recompile.** Each set of compile-time parameters needs its own compile.
- **Builtins only, on a patched back end.** HaskLedger cannot use Covenant's typed `match` and `ctor'` until c2uplc's transformation stage compiles them correctly, and it builds against its vendored c2uplc rather than an upstream release.
- **No failure reasons.** A refused transaction does not say which check failed.
- **One tested toolchain.** Only the versions in the Nix dev shell are tested, and the `riscv64-linux` build has been validated under QEMU only.

09 RELATED WORK

PlutusTx compiles ordinary Haskell with a GHC plugin through the Plutus compiler pipeline. It is the reference toolchain, has typed data and ordinary pattern matching, and is the easiest start for a Haskell programmer. It is the only toolchain HaskLedger has been measured against, and careful PlutusTx code can narrow the gap by reading Data directly, the way HaskLedger does.

Plutarch is also a Haskell eDSL, with its own code generator. It gives fine control over the generated code, has a typed layer over Plutus data, and is used by production protocols. HaskLedger aims instead at contracts that read like the rule they enforce, such as `txValidRange `after` deadline` or `paysAtLeast outputs seller amount`, with fewer concepts to learn. The price is less control and, for now, no typed data layer.

Aiken is a separate language designed for Cardano, with its own compiler and tools: a test runner with property-based tests, blueprint generation, a language server, a formatter, a standard library and thorough documentation. That is the bar for HaskLedger's tooling and documentation, and HaskLedger is not there yet. What it offers instead is Haskell: contracts, off-chain code and tests share one language and type system, and contracts are ordinary values that Haskell functions can generate and parameterise. Targeting Covenant rather than Plutus also keeps its front end apart from the back end.[11]

Plutarch and PlutusTx both have typed on-chain data, and get test runners and CIP-57 blueprints through Haskell libraries; HaskLedger has none of these yet. It has not been benchmarked against Aiken or Plutarch, and this paper makes no claim about which produces smaller or cheaper scripts. It fits Haskell teams that care about script size and cost and write rule-shaped contracts (who may sign, what must be paid, and when) rather than heavy data processing.

11 Aiken is widely used in production and Plutarch is used in production. PlutusTx is the reference toolchain. HaskLedger is new.

10 FUTURE WORK

The planned work follows the limits:

- **Datum-parametric versions of the remaining fixed-configuration examples**, following the pattern escrow, vesting, token-gate and multisig already use, so that one compiled script serves many locks.
- **Richer value checks**, starting with payout floors for native tokens.
- **Reporting which check failed** without reading UPLC.
- **The test helpers as a library module.**
- **Continuous integration** for the test suites and the benchmark.
- **PlutusTx baselines for escrow, vesting, multisig and treasury**, so the comparison covers contracts that walk the input and output lists.
- **Validation on physical RISC-V hardware**, beyond the QEMU runs.
- **Typed datums**, so that a wrong field index or field type becomes a compile error.
- **CIP-57 blueprints**, so off-chain tools can read a contract's interface.

The strict boolean operators stay by design, with the `case` functions as the lazy alternative, and the builtin-only rule stays until c2uplc's transformation stage handles Covenant's typed matching.