An overview of the Plutus Core cost model

10th October 2024

1 Introduction

Plutus Core, or more precisely Untyped Plutus Core (UPLC for short), is the on-chain language of the Cardano blockchain. It is a low-level language which is used for purposes such as transaction validation, enabling the use of *smart contracts* on Cardano. Scripts will generally be written in some higher-level language and then compiled into Plutus Core. Plutus Core scripts are assigned costs which depend on their execution time and memory consumption, and a user must pay a charge based on the script cost in order to have their script executed on the chain. These charges have two purposes:

- To make sure that users pay for the computational resources which they consume.
- To deter the use of overly-expensive scripts (and in particular, ones which run forever) in order to avoid denial-of-service attacks on the chain.

This document provides an outline of the costing mechanism and the parameters that it depends upon. A full specification will be published at some time in the future.

2 The Plutus Core Cost Model

Plutus Core is essentially the untyped lambda calculus augmented with some "built-in" types (`integer`, `bool`, ...) and "built-in" functions ("builtins" for short) which carry out integer arithmetic, cryptographic calculations, and so on. There are currently 10 basic lambda-calculus operations (`var`, `lam`, `apply`, `force`, `delay`, `constant`, `builtin`, `error`, `constr`, and `case`) and 75 built-in functions, most of which are underlain by Haskell or C library functions.

For a full description of the language and the built-in functions, see the Plutus Core specification.[1]

We have a *cost model* which assigns CPU and memory costs to every UPLC script. CPU costs are measured in units called `ExCPU`: one `ExCPU` unit is notionally equivalent to one picosecond of CPU time on a dedicated benchmarking machine. Memory usage is measured in `ExMemory` units, with one unit being

[1]This can be built from source at `https://github.com/IntersectMBO/plutus/tree/master/doc/plutus-core-spec`; a prebuilt version of the specification is available as a PDF at `https://ci.iog.io/job/input-output-hk-plutus/master/x86_64-linux.packages.plutus-core-spec/latest/download/1`.

equal to 8 bytes, or one 64-bit word. Every built-in type has a size measure which calculates the number of `ExMemory` units required to hold a given value of the type, and these measures are also used to find the sizes of inputs to built-in functions.

2.1 CPU costing

We assign a constant CPU and memory cost to each of the basic lambda-calculus operations of the Plutus Core evaluator (which we sometimes call *machine steps*). Currently all of the basic operations have the same cost. There is also a (very small) startup cost for the evaluator. Built-in functions are more complicated: each has two associated *costing functions* (one for CPU and one for memory) which assign a cost to calling the function based on the size of its arguments. For example the CPU cost of calling `addInteger` is a linear function of the maximum of the sizes of its arguments, since in the worst case we have to traverse the entirety of both arguments in order to calculate the output. The basic "shape" of a CPU costing function (constant, linear in the size of one of the function's arguments, linear in the sum of the sizes of two of the function's arguments, etc.) is arrived at from an understanding of the expected behaviour of the function and also by running benchmarks and examining the results to check that nothing unexpected is happening. Values for the coefficients of the costing function are obtained by running microbenchmarks (*budgeting benchmarks*) on our reference machine. These run the builtins with inputs of varying sizes and record the execution time, and then we use the R statistical system to fit a function of the expected shape to the data and infer the coefficients which give the best fit (and we check that we do get a good fit to the expected "shape"): the coefficients are then stored in a JSON file which is used by the Plutus Core evaluator to construct suitable Haskell versions of the costing functions.

Experiments on our reference machine indicate that CPU costs obtained from the cost model predict the actual execution times of scripts reasonably well, underestimating actual times by a maximum of about 5% and overestimating by a maximum of about 15% (which is to be expected since our costing functions are based on worst-case behaviour). Other machines will have different CPU speeds and different architectures, but experiments show that actual times are still roughly proportional to the CPU cost assigned by the cost model.

2.2 Memory costing

Memory costs for built-in functions are measured using costing functions similar to the ones for CPU consumption, but the costing functions measure only the size of the *output* of a function and do not attempt to take any account of memory allocated during the execution of the builtin. No inference is performed: we obtain memory costing functions simply by looking at the definition of the function. We also make no attempt to account for garbage collection in the Haskell heap during script evaluation. This is a very crude measure of memory usage (for example, a cryptographic signature verification function might perform a lengthy computation using lots of memory in the C heap, but we only take account of the memory used by its boolean return value, so the memory cost will be 1 `ExMemory` unit). Our memory model produces an upper bound for the maximum amount of non-garbage-collectable memory that a Plutus Core program might allocate during its execution. The memory cost of a program mostly serves as a guard against runaway memory allocation, and we regard CPU costing as our primary measure of cost.

2.3 Cost models and cost model parameters

A Plutus Core *cost model* consists of CPU and memory costs for all of the basic Plutus Core operations together with CPU and memory costing functions for all of the built-in functions. The specific cost model used when a script is evaluated on the chain may depend on the Plutus language version and the protocol version. This allows us to account for factors such as improvements in the efficiency of the evaluator, and also to maintain backwards compatiblity and ensure that the evaluation costs of scripts already on the chain remain unchanged, which is important when the history of the chain is replayed.

A specific set of values for the machine step costs and the coefficients of the costing functions is referred to as a set of *cost model parameters* (although *cost model coefficients* might be a better name). The Cardano ledger protocol parameters contain cost model parameters for each Plutus language version and each protocol version, and these are supplied to the Plutus Core evaluator before a script is evaluated to ensure that the execution costs are calculated appropriately.

2.4 Measuring costs during script execution

Plutus Core scripts are deterministic: it is always known in advance exactly what the arguments of a script will be when it is executed on the chain, and hence it is also known exactly how the execution will proceed. This means that an overall cost can be assigned to a program by running it and adding up the total costs of the machine steps and the builtin calls. This is done by running the Plutus Core evaluator in *counting mode.* When a program is run on the chain the submitter supplies an expected CPU and memory budget (which they pay for in Ada). To prevent denial of service it is necessary to check that the submitter has not lied about the budget, and this is done by running the evaluator in *restricting mode*: it is supplied with the claimed budget and this is decremented as evaluation proceeds, with an error occurring if either the CPU or memory component ever become negative. Keeping track of costs adds some overhead to execution times, and to minismise this we allow some *slippage* in machine step costs. A charge is made for the execution of each machine step, but the on-chain evaluator only checks that the total budget has not been exceeded once every 200 machine steps: this allows a program to perform slightly more computation than has been budgeted for but reduces the costing overhead significantly. Slippage only applies to machine steps: the cost of evaluating a built-in function is calculated just before it is called and it is checked that calling it willnot cause the script's budget to be exceeded, so we never call a function if its execution will be too expensive.

There are per-script limits for CPU and memory usage of scripts on the chain. These are the `steps` and `memory` components of the `maxTxExecution-Units` protocol parameter and at the time of writing they have values of 10,000,000,000 `ExCPU` and 14,000,000 `ExMemory` respectively. There are also per-block bounds in `maxBlockExecutionUnits` on total CPU usage (20,000,000,000 `ExCPU`, twice the maximum script CPU usage) and total memory usage (62,000,000 `ExMemory`, about 4.4 times the maximum script memory usage). Another relevant protocol parameter is `executionUnitPrices` which relates abstract costs to real-world Ada costs: in Protocol Version 9 this specifies monetary costs of 7.21×10^{-5} Ada per `ExCPU` unit and 5.77×10^{-2} Ada per `ExMemory` unit.

3 Costing functions for Plutus Core builtins

Tables 1 and 2 show the forms of the CPU and memory costing functions for the Plutus Core built-in functions on Cardano for PlutusV3 and Protocol Version 9 (the Protocol Version in effect on the Cardano chain at the time of writing). They also include some new built-in functions (extra bitwise operations on bytestrings and the `RIPEMD-160` hash function) which we expect will become available in Protocol Version 10: see Section 4.3.5 of the Plutus Core specification.

The symbols $x, y, z, \ldots$ refer to the arguments of the costing functions, which are the *sizes* of the actual arguments of the builtin; occasionally we will need to refer to the actual *value* of an argument, and in that case we use the symbols $\mathbf{x}, \mathbf{y}, \mathbf{z}, \ldots$.

Symbols such as $a, b, c, \ldots$ refer to the coefficients of the costing functions. Concrete values for these coefficients are stored in the Cardano protocol parameters: see Section 3.1 for more on this.

Function	Arity	CPU	Memory
`addInteger`	2	$a + b \cdot \max(x, y)$	$a + b \cdot \max(x, y)$
`andByteString`	2	$a + by + cz$	$a + b \cdot \max(y, z)$
`appendByteString`	2	$a + b(x + y)$	$a + b(x + y)$
`appendString`	2	$a + b(x + y)$	$a + b(x + y)$
`bData`	1	constant	constant
`blake2b_224`	1	$a + bx$	constant
`blake2b_256`	1	$a + bx$	constant
`bls12_381_G1_add`	2	constant	constant
`bls12_381_G1_compress`	1	constant	constant
`bls12_381_G1_equal`	2	constant	constant
`bls12_381_G1_hashToGroup`	2	$a + bx$	constant
`bls12_381_G1_neg`	1	constant	constant
`bls12_381_G1_scalarMul`	2	$a + bx$	constant
`bls12_381_G1_uncompress`	1	constant	constant
`bls12_381_G2_add`	2	constant	constant
`bls12_381_G2_compress`	1	constant	constant
`bls12_381_G2_equal`	2	constant	constant
`bls12_381_G2_hashToGroup`	2	$a + bx$	constant
`bls12_381_G2_neg`	1	constant	constant
`bls12_381_G2_scalarMul`	2	$a + bx$	constant
`bls12_381_G2_uncompress`	1	constant	constant
`bls12_381_finalVerify`	2	constant	constant
`bls12_381_millerLoop`	2	constant	constant
`bls12_381_mulMlResult`	2	constant	constant
`byteStringToInteger`	2	$c_0 + c_1 y + c_2 y^2$	$a + by$
`chooseData`	6	constant	constant
`chooseList`	3	constant	constant
`chooseUnit`	2	constant	constant
`complementByteString`	1	$a + bx$	$a + bx$
`consByteString`	2	$a + by$	$a + b(x + y)$

Table 1: Costing functions for Plutus Core built-in functions (1)

Function	Arity	CPU	Memory
`constrData`	2	constant	constant
`countSetBits`	1	$a+bx$	$a+bx$
`decodeUtf8`	1	$a+bx$	$a+bx$
`divideInteger`	2	See Note 1	$a+b\cdot\max(x-y,c)$
`encodeUtf8`	1	$a+bx$	$a+bx$
`equalsByteString`	2	$\begin{cases} a+bx & \text{if } x=y \\ c & \text{if } x\neq y \end{cases}$	constant
`equalsData`	2	$a+b\cdot\min(x,y)$	constant
`equalsInteger`	2	$a+b\cdot\min(x,y)$	constant
`equalsString`	2	$\begin{cases} a+bx & \text{if } x=y \\ c & \text{if } x\neq y \end{cases}$	constant
`findFirstSetBit`	1	$a+bx$	constant
`fstPair`	1	constant	constant
`headList`	1	constant	constant
`iData`	1	constant	constant
`ifThenElse`	3	constant	constant
`indexByteString`	2	constant	constant
`integerToByteString`	3	$c_0+c_1z+c_2z^2$	$\begin{cases} z & \text{if } \mathbf{y}=0 \\ \lceil(\lvert\mathbf{y}\rvert-1)/8\rceil+1 & \text{if } \mathbf{y}\neq 0 \end{cases}$
`keccak_256`	1	$a+bx$	constant
`lengthOfByteString`	1	constant	constant
`lessThanByteString`	2	$a+b\cdot\min(x,y)$	constant
`lessThanEqualsByteString`	2	$a+b\cdot\min(x,y)$	constant
`lessThanEqualsInteger`	2	$a+b\cdot\min(x,y)$	constant
`lessThanInteger`	2	$a+b\cdot\min(x,y)$	constant
`listData`	1	constant	constant
`mapData`	1	constant	constant
`mkCons`	2	constant	constant
`mkNilData`	1	constant	constant
`mkNilPairData`	1	constant	constant
`mkPairData`	2	constant	constant
`modInteger`	2	See Note 1	$a+by$
`multiplyInteger`	2	$a+bxy$	$a+b(x+y)$
`nullList`	1	constant	constant
`orByteString`	2	$a+by+cz$	$a+b\cdot\max(y,z)$
`quotientInteger`	2	See Note 1	$a+b\cdot\max(x-y,c)$
`readBit`	2	constant	constant
`remainderInteger`	2	See Note 1	$a+by$
`replicateByte`	2	$a\mathbf{x}$	$\mathbf{x}$
`ripemd_160`	1	$a+bx$	constant
`rotateByteString`	2	$a+bx$	$a+bx$
`serialiseData`	1	$a+bx$	$a+bx$
`sha2_256`	1	$a+bx$	constant
`sha3_256`	1	$a+bx$	constant
`shiftByteString`	2	$a+bx$	$a+bx$
`sliceByteString`	3	$a+bz$	$a+bz$
`sndPair`	1	constant	constant
`subtractInteger`	2	$a+b\cdot\max(x,y)$	$a+b\cdot\max(x,y)$
`tailList`	1	constant	constant
`trace`	2	constant	constant
`unBData`	1	constant	constant
`unConstrData`	1	constant	constant
`unIData`	1	constant	constant
`unListData`	1	constant	constant
`unMapData`	1	constant	constant
`verifyEcdsaSecp256k1Signature`	3	constant	constant
`verifyEd25519Signature`	3	$a+by$	constant
`verifySchnorrSecp256k1Signature`	3	$a+by$	constant
`writeBits`	2	$a+b\cdot\mathrm{length}(\mathbf{y})$	$a+by$
`xorByteString`	2	$a+by+cz$	$a+b\cdot\max(y,z)$

Table 2: Costing functions for Plutus Core built-in functions (2)

Note 1. The CPU costing functions for the four integer division functions (`divideInteger`, `modInteger`, `quotientInteger` and `remainderInteger`) are rather complicated. For $x < y$ the cost is constant since essentially no work has to be done; however for $x \geq y$ the cost is a quadratic function of the sizes of the two inputs with a lower bound to ensure that the cost is never negative:

$$\text{CPU cost} = \begin{cases} \text{constant} & \text{if } x < y \\ \max(d, c_{00} + c_{10}x + c_{01}y + c_{20}x^2 + c_{11}xy + c_{02}y^2) & \text{if } x \geq y. \end{cases}$$

3.1 Cost model parameters in the protocol parameters

In the protocol parameters the cost model parameters take the form of a list of named constants which correspond to the coefficients $a, b, c, \ldots$ in Tables 1 and 2 above. An excerpt from the cost model parameters for Protocol Version 9 is shown in Figure 1. The tags are obtained from the names of the builtins and the various coefficients of the costing functions. We won't spell out the full details of how the tags are obtained, but reference to Tables 1 and 2 should make this fairly clear. For instance,

- For integer arguments X of size x and Y of size y, the cost of calling the `addInteger` function to add X to Y will be $420 \cdot \max(x, y) + 100788$ `ExCPU` units and $\max(x, y) + 1$ `ExMemory` units.

- For bytestring arguments X of size x and Y of size y, the cost of calling the `appendByteString` function to append Y to X will be $173(x + y) + 1000$ `ExCPU` units and $x + y + 1$ `ExMemory` units.

- The CPU cost of calling the `bData` function on a bytestring will be a constant 11183 `ExCPU` units and 32 `ExMemory` units, irrespective of the size of the input.

- The CPU cost of the Plutus Core evaluator's basic `apply` operation (and all of the other basic operations (machine steps)) is a constant 16000 `ExCPU` units and the memory cost is a constant 100 `ExMemory` units.

- For string arguments X of size x and Y of size y, the CPU cost of calling the `equalsString` function to check if X and Y are equal will be $1000 + 60594x$ `ExCPU` units if $x = y$ and 39184 `ExCPU` units if $x \neq y$; the memory cost will be a constant 1 `ExMemory` unit.

```
"costModels": {
    ...
    "PlutusScriptV3": {
        "addInteger-cpu-arguments-intercept": 100788,
        "addInteger-cpu-arguments-slope": 420,
        "addInteger-memory-arguments-intercept": 1,
        "addInteger-memory-arguments-slope": 1,
        "appendByteString-cpu-arguments-intercept": 1000,
        "appendByteString-cpu-arguments-slope": 173,
        "appendByteString-memory-arguments-intercept": 0,
        "appendByteString-memory-arguments-slope": 1,
        "appendString-cpu-arguments-intercept": 1000,
        "appendString-cpu-arguments-slope": 59957,
        "appendString-memory-arguments-intercept": 4,
        "appendString-memory-arguments-slope": 1,
        "bData-cpu-arguments": 11183,
        "bData-memory-arguments": 32,
        ...
        "cekApplyCost-exBudgetCPU": 16000,
        "cekApplyCost-exBudgetMemory": 100,
        "cekBuiltinCost-exBudgetCPU": 16000,
        "cekBuiltinCost-exBudgetMemory": 100,
        "cekCaseCost-exBudgetCPU": 16000,
        "cekCaseCost-exBudgetMemory": 100,
        ...
        "equalsString-cpu-arguments-constant": 39184,
        "equalsString-cpu-arguments-intercept": 1000,
        "equalsString-cpu-arguments-slope": 60594,
        "equalsString-memory-arguments": 1,
        ...
    }
    ...
}
```

Figure 1: Extract from cost model parameters