Bitwise primitives proposal for Plutus

Koz Ross

February 16, 2022

1 Motivation

Bitwise operations are one of the most fundamental building blocks of algorithms and data structures. They can be used for a wide variety of applications, ranging from representing and manipulating sets of integers efficiently, to implementations of cryptographic primitives, to fast searches. Their wide availability, law-abiding behaviour and efficiency are the key reasons why they are widely used, and widely depended on.

At present, Plutus lacks meaningful support for bitwise operations, which significantly limits what can be usefully done on-chain. While it is possible to mimic some of these capabilities with what currently exists, and it is always possible to introduce new primitives for any task, this is extremely unsustainable, and often leads to significant inefficiencies and duplication of effort.

We describe a list of bitwise operations, as well as their intended semantics, designed to address this problem.

2 Goals

To ensure a focused and meaningful proposal, we specify our goals below.

2.1 Useful primitives

The primitives provided should enable implementations of algorithms and data structures that are currently impossible or impractical. Furthermore, the primitives provided should have a high power-to-weight ratio: having them should enable as much as possible to be implemented.

2.2 Maintaining as many algebraic laws as possible

Bitwise operations, via Boolean algebras, have a long and storied history of algebraic laws, dating back to important results by the like of de Morgan, Post and many others. These algebraic laws are useful for a range of reasons: they guide implementations, enable easier testing (especially property testing) and in some cases much more efficient implementations. To some extent, they also formalize our intuition about how these operations 'should work'. Thus, maintaining as many of these laws in our implementation, and being clear about them, is important.

2.3 Allowing efficient, portable implementations

Providing primitives alone is not enough: they should also be efficient. This is not least of all because many would associate 'primitive operation' with a notion of being 'close to the machine', and therefore fast. Thus, it is on us to ensure that the implementations of the primitives we provide have to be implementable in an efficient way, across a range of hardware.

3 Non-goals

We also specify some specific non-goals of this proposal.

3.1 No partial operations

Partiality is an undesirable property in any language and any functionality. This is especially true of bitwise operations, which are fairly low-level and need highly-predictable semantics to allow implementers to use them with confidence and know exactly what answers to expect. Furthermore, especially off-chain, dealing with partiality is annoying and difficult to track down issues with: MLabs in particular have lost endless hours with division-by-zero bugs.

3.2 No metaphor-mixing between numbers and bits

A widespread legacy of C is the mixing of treatment of numbers and blobs of bits: specifically, the allowing of logical operations on representations of numbers. This applies to Haskell as much as any other language: according to the Haskell Report, it is in fact *required* that any type implementing `Bits` implement `Num` first. While GHC Haskell only mandates `Eq`, it still defines `Bits` instances for types clearly meant to represent numbers. This is a bad choice, as it creates complex situations and partiality in several cases, for arguably no real gain other than C-like bit twiddling code.

Even if two types share a representation, their type distinctness is meant to be a semantic or abstraction boundary: just because a number is represented as a blob of bits does not necessarily mean that arbitrary bit manipulations are sensible. However, by defining such a capability, we create several semantic problems:

- Some operations end up needing multiple definitions to take this into account. A good example are shifts: instead of simply having left or right shifts, we now have to distinguish *arithmetic* versus *logical* shifts, simply to take into account that a shift can be used on something which is meant to be a number, which could be signed. This creates unnecessary complexity and duplication of operations.
- As Plutus `BuiltinInteger`s are of arbitrary precision, certain bitwise operations are not well-defined on them. A good example is bitwise complement: the bitwise complement of 0 cannot be defined sensibly, and in fact, is partial in its `Bits` instance.
- Certain bitwise operations on `BuiltinInteger` would have quite undesirable semantic changes in order to be implementable. A good example are bitwise rotations: we should be able to 'decompose' a rotation left or right by n into two rotations (by m_1 and m_2 such that $m_1 + m_2 = n$) without changing the outcome. However, because trailing zeroes are not tracked by the implementation, this can fail depending on the choice of decomposition, which seems needlessly annoying for no good reason.
- Certain bitwise operations on `BuiltinInteger` would require additional arguments and padding to define them sensibly. Consider bitwise logical AND: in order to perform this sensibly on `BuiltinInteger`s we would need to specify what 'length' we assume they have, and some policy of 'padding' when the length requested is longer than one, or both, arguments. This feels unnecessary, and isn't even clear exactly how this should be done: for example, how would negative numbers be padded?

These complexities, and many more besides, are poor choices, owing more to the legacy of C than any real useful functionality. Furthermore, they feel like a casual and senseless undermining of type safety and its guarantees for very small and questionable gains. Therefore, defining bitwise operations on `BuiltinInteger` is not something we wish to support.

There are legitimate cases where a conversion from `BuiltinInteger` to `BuiltinByteString` is desirable; this conversion should be provided, and be both explicit and specified in a way that is independent of the machine or the implementation of `BuiltinInteger`, as well as total and round-tripping. Arguably, it is also desirable to provide built-in support for `BuiltinByteString` literals specified in a way convenient to their treatment as blobs of bytes (for example, hexadecimal or binary notation), but this is outside the scope of this proposal.

4 Proposed operations

We propose several classes of operations. Firstly, we propose two operations for inter-conversion between `BuiltinByteString` and `BuiltinInteger`, whose semantics are specified in Subsection 5.2:

`integerToByteString :: BuiltinInteger -> BuiltinByteString`: Convert a number to a bitwise representation.

`byteStringToInteger :: BuiltinByteString -> BuiltinInteger:` Reinterpret a bitwise representation as a number.

We also propose several logical operations on `BuiltinByteString`s, whose semantics are specified in Subsection 5.3:

`andByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString:` Perform a bitwise logical AND on the arguments, producing a result whose size is the minimum of the sizes of its arguments.

`iorByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString:` Perform a bitwise logical IOR on the arguments, producing a result whose size is the minimum of the sizes of its arguments.

`xorByteString :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString:` Perform a bitwise logical XOR on the arguments, producing a result whose size is the minimum of the sizes of its arguments.

`complementByteString :: BuiltinByteString -> BuiltinByteString:` Complement all the bits in the argument, producing a result whose size is the same as the argument.

Lastly, we define the following additional operations, whose semantics are specified in Subsection 5.4:

`shiftByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinByteString:` Performs a bitwise shift of the first argument by the absolute value of the second argument, with padding, the direction being indicated by the sign of the second argument.

`rotateByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinByteString:` Performs a bitwise rotation of the first argument by the absolute value of the second argument, the direction being indicated by the sign of the second argument.

`popCountByteString :: BuiltinByteString -> BuiltinInteger:` Returns the number of 1 bits in the argument.

`bitAtByteString :: BuiltinByteString -> BuiltinInteger -> BuiltinBool:` Returns the bit of the first argument at the position given by the second argument, from the right.

`findFirstZeroByteString :: BuiltinByteString -> BuiltinInteger:` Returns the index of the rightmost 0 bit in the argument.

`findFirstOneByteString :: BuiltinByteString -> BuiltinInteger:` Returns the index of the rightmost 1 bit in the argument.

5 Semantics

5.1 Preliminaries

We define $\mathbb{N}^+ = \{x \in \mathbb{N} \mid x \neq 0\}$. We assume that `BuiltinInteger` is a faithful representation of $\mathbb{Z}$. A *bit sequence* $s = s_n s_{n-1} \dots s_0$ is a sequence such that for all $i \in \{0, 1, \dots, n\}$, $s_i \in \{0, 1\}$. A bit sequence $s = s_n s_{n-1} \dots s_0$ is a *byte sequence* if $n = 8k - 1$ for some $k \in \mathbb{N}$. We denote the *empty bit sequence* (and, indeed, byte sequence as well) by $\emptyset$.

Let $i \in \mathbb{N}^+$. We define the sequence $\texttt{binary}(i) = (d_0, m_0), (d_1, m_1), \dots$ as

1. $m_0 = i \mod 2$, $d_0 = \frac{i}{2}$ if i is even, and $\frac{i-1}{2}$ if it is odd.
2. $m_j = d_{j-1} \mod 2$, $d_j = \frac{d_{j-1}}{2}$ if d_j is even, and $\frac{d_{j-1}-1}{2}$ if it is odd.

5.2 Representation of `BuiltinInteger` as `BuiltinByteString` and conversions

We describe the translation of `BuiltinInteger` into `BuiltinByteString` which is implemented as the `integerToByteString` primitive. Let $i \in \mathbb{N}^+$. We represent i as the bit sequence $s = s_n s_{n-1} \dots s_0$, such that:

1. $\sum_{j \in \{0,1,\dots,n\}} s_j \cdot 2^j = i$; and
2. $s_n = 0$.
3. Let $\texttt{binary}(j) = (d_0, m_0), (d_1, m_1), \dots$. For any $j \in \{0, 1, \dots, n-1\}$, $s_j = m_j$; and
4. $n + 1 = 8k$ for the smallest $k \in \mathbb{N}^+$ consistent with the previous requirements.

For 0, we represent it as the sequence `00000000` (one zero byte). We represent any $i \in \{x \in \mathbb{Z} \mid x < 0\}$ as the twos-complement of the representation of its additive inverse. We observe that any such sequence is by definition a byte sequence.

To interpret a byte sequence $s = s_n s_{n-1} \dots s_0$ as a `BuiltinInteger`, we use the following process:

1. If s is `00000000`, then the result is 0.
2. Otherwise, if $s_n = 1$, let s' be the twos-complement of s. Then the result is the additive inverse of the result of interpreting s'.
3. Otherwise, the result is $\sum_{i \in \{0,1,\dots,n\}} s_i \cdot 2^i$.

The above interpretation is implemented as the `byteStringToInteger` primitive. We observe that `byteStringToInteger` and `integerToByteString` form an isomorphism. More specifically:

```
byteStringToInteger . integerToByteString =
integerToByteString . byteStringToInteger =
id
```

5.3 Bitwise logical operations on `BuiltinByteString`

Throughout, let $s = s_n s_{n-1} \dots s_0$ and $t = t_m t_{m-1} \dots t_0$ be two byte sequences.

We describe the semantics of `andByteString`. For inputs s and t, the result is the byte sequence $u = u_{\min\{n,m\}} u_{\min\{n,m\}-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, \min\{n, m\}\}$, we have

$$u_i = \begin{cases} 1 & s_i = t_i = 1 \\ 0 & \text{otherwise} \end{cases}$$

For `iorByteString`, for inputs s and t, the result is the byte sequence $u = u_{\min\{n,m\}} u_{\min\{n,m\}-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, \min\{n, m\}\}$, we have

$$u_i = \begin{cases} 1 & s_i = 1 \\ 1 & t_i = 1 \\ 0 & \text{otherwise} \end{cases}$$

For `xorByteString`, for inputs s and t, the result is the byte sequence $u = u_{\min\{n,m\}} u_{\min\{n,m\}-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, \min\{n, m\}\}$, we have

$$u_i = \begin{cases} 1 & s_i \neq t_i \\ 0 & \text{otherwise} \end{cases}$$

We observe that each of `andByteString`, `iorByteString` and `xorByteString` describes a commutative and associative operation. Furthermore, $\emptyset$ is an annihilating element for each of these operations.

We now describe the semantics of `complementByteString`. For input s, the result is the byte sequence $u = u_n u_{n-1} \dots u_0$ such that for all $i \in \{0, 1, \dots, n\}$ we have

$$u_i = \begin{cases} 1 & s_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

We observe that `complementByteString` is self-inverting.

5.4 Mixed operations

Throughout this section, let $s = s_n s_{n-1} \dots s_0$ and $t = t_m t_{m-1} \dots t_0$ be byte sequences, and let $i \in \mathbb{Z}$.

We describe the semantics of `shiftByteString`, assuming the arguments are s and i. The result is the byte sequence $u_n u_{n-1} \dots u_0$, such that for all $j \in \{0, 1, \dots, n\}$, we have

$$u_j = \begin{cases} s_{j+i} & j - i \in \{0, 1, \dots, n\} \\ 0 & \text{otherwise} \end{cases}$$

We observe that for k, ℓ with the same sign and any `bs`, we have

```
shiftByteString (shiftBytestring bs k) l = shiftByteString bs (k + l)
```

We now describe `rotateByteString`, assuming the same inputs as the description of `shiftByteString` above. The result is the byte sequence $u_n u_{n-1} \dots u_0$ such that for all $j \in \{0, 1, \dots, n\}$, we have $u_j = s_{j+i \mod (n+1)}$. We observe that for any k, ℓ, and any `bs`, we have

```
rotateByteString (rotateByteString bs k) l = rotateByteString bs (k + l)
```

We also note that

```
rotateByteString bs 0 = shiftByteString bs 0 = bs
```

For `popCountByteString` with argument s, the result is

$$\sum_{j \in \{0,1,\dots,n\}} s_j$$

We observe that for any `bs` and `bs'`, we have

```
popCountByteString bs + popCountByteString bs' =
popCountByteString (appendByteString bs bs')
```

We now describe the semantics of `bitAtByteString` The result of `bitAtByteString` on s and i is s_i if $0 \leq i \leq n$, and 0 otherwise.

For `findFirstZeroByteString` and `findFirstOneByteString`, we assume s as the argument. Throughout, let $I = \{0, 1, \dots, n+1\}$. If $\nexists k \in I - \{n+1\}$ such that $s_k = 0$, then `findFirstZeroByteString` returns $n+1$; analogously for $s_k = 1$ and `findFirstOneByteString`. Otherwise, both operations return $j \in I - \{n+1\}$ such that:

- For `findFirstZeroByteString`, `bitAtByteString` on s and j returns 0, and $\nexists k < j \in I$ such that `bitAtByteString` on s and k returns 0.
- For `findFirstOneByteString`, `bitAtByteString` on s and j returns 1, and $\nexists k < j \in I$ such that `bitAtByteString` on s and k returns 1.