2.4. cbor.ak

use aiken/builtin.{

append\_bytearray, choose\_data, cons\_bytearray, decode\_utf8, index\_bytearray,

length\_of\_bytearray, quotient\_integer, remainder\_integer, serialise\_data,

un\_b\_data, un\_constr\_data, un\_i\_data, un\_list\_data, un\_map\_data,

}

use aiken/list

/// Serialise any value to binary, encoding using [CBOR](https://www.rfc-editor.org/rfc/rfc8949).

///

/// This is particularly useful in combination with hashing functions, as a way

/// to obtain a byte representation that matches the serialised representation

/// used by the ledger in the context of on-chain code.

///

/// Note that the output matches the output of [`diagnostic`](#diagnostic),

/// though with a different encoding. [`diagnostic`](#diagnostic) is merely a

/// textual representation of the CBOR encoding that is human friendly and

/// useful for debugging.

///

/// ```aiken

/// serialise(42) == #"182a"

/// serialise(#"a1b2") == #"42a1b2"

/// serialise([]) == #"80"

/// serialise((1, 2)) == #"9f0102ff"

/// serialise((1, #"ff", 3)) == #"9f0141ff03ff"

/// serialise([(1, #"ff")]) == #"a10141ff"

/// serialise(Some(42)) == #"d8799f182aff"

/// serialise(None) == #"d87a80"

/// ```

pub fn serialise(self: Data) -> ByteArray {

serialise\_data(self)

}

test serialise\_1() {

serialise(42) == #"182a"

}

test serialise\_2() {

serialise(#"a1b2") == #"42a1b2"

}

test serialise\_3() {

serialise([]) == #"80"

}

test serialise\_4() {

serialise((1, 2)) == #"9f0102ff"

}

test serialise\_5() {

serialise((1, #"ff", 3)) == #"9f0141ff03ff"

}

test serialise\_6() {

serialise([(1, #"ff")]) == #"9f9f0141ffffff"

}

test serialise\_7() {

serialise(Some(42)) == #"d8799f182aff"

}

test serialise\_8() {

serialise(None) == #"d87a80"

}

test serialise\_9() {

serialise([Pair(1, #"ff")]) == #"a10141ff"

}

/// Obtain a String representation of \_anything\_. This is particularly (and only) useful for tracing

/// and debugging. This function is expensive and should not be used in any production code as it

/// will very likely explodes the validator's budget.

///

/// The output is a [CBOR diagnostic](https://www.rfc-editor.org/rfc/rfc8949#name-diagnostic-notation)

/// of the underlying on-chain binary representation of the data. It's not as

/// easy to read as plain Aiken code, but it is handy for troubleshooting values

/// \_at runtime\_. Incidentally, getting familiar with reading CBOR diagnostic is

/// a good idea in the Cardano world.

///

/// ```aiken

/// diagnostic(42) == "42"

/// diagnostic(#"a1b2") == "h'A1B2'"

/// diagnostic([1, 2, 3]) == "[\_ 1, 2, 3]"

/// diagnostic([]) == "[]"

/// diagnostic((1, 2)) == "[\_ 1, 2]"

/// diagnostic((1, #"ff", 3)) == "[\_ 1, h'FF', 3]"

/// diagnostic([(1, #"ff")]) == "{\_ 1: h'FF' }"

/// diagnostic(Some(42)) == "121([\_ 42])"

/// diagnostic(None) == "122([])"

/// ```

pub fn diagnostic(self: Data) -> String {

do\_diagnostic(self, #"")

|> decode\_utf8

}

/// UTF-8 lookup table. Comes in handy to decipher the code below.

///

/// | Symbol | Decimal | Hex |

/// | --- | --- | --- |

/// | | 32 | 0x20 |

/// | ' | 39 | 0x27 |

/// | ( | 40 | 0x28 |

/// | ) | 41 | 0x29 |

/// | , | 44 | 0x2c |

/// | 0 | 48 | 0x30 |

/// | : | 58 | 0x3a |

/// | A | 65 | 0x41 |

/// | [ | 91 | 0x5b |

/// | ] | 93 | 0x5d |

/// | \_ | 95 | 0x5f |

/// | h | 104 | 0x68 |

/// | { | 123 | 0x7b |

/// | } | 125 | 0x7d |

fn do\_diagnostic(self: Data, builder: ByteArray) -> ByteArray {

choose\_data(

self,

{

// -------- Constr

let Pair(constr, fields) = un\_constr\_data(self)

// NOTE: This is fundamentally the same logic for serializing list. However, the compiler

// doesn't support mutual recursion just yet, so we can't extract that logic in a separate

// function.

//

// See [aiken-lang/aiken#389](https://github.com/aiken-lang/aiken/pull/389)

let builder =

when fields is {

[] -> append\_bytearray(#"5b5d29", builder)

\_ -> {

let (\_, bytes) =

list.foldr(

fields,

(#"5d", append\_bytearray(#"29", builder)),

fn(e: Data, st: (ByteArray, ByteArray)) {

(#"2c20", do\_diagnostic(e, append\_bytearray(st.1st, st.2nd)))

},

)

append\_bytearray(#"5b5f20", bytes)

}

}

let constr\_tag =

if constr < 7 {

121 + constr

} else if constr < 128 {

1280 + constr - 7

} else {

fail @"What are you doing? No I mean, seriously."

}

builder

|> append\_bytearray(#"28", \_)

|> from\_int(constr\_tag, \_)

},

{

// -------- Map

let elems = un\_map\_data(self)

when elems is {

[] -> append\_bytearray(#"7b7d", builder)

\_ -> {

let (\_, bytes) =

list.foldr(

elems,

(#"207d", builder),

fn(e: Pair<Data, Data>, st: (ByteArray, ByteArray)) {

let value =

do\_diagnostic(e.2nd, append\_bytearray(st.1st, st.2nd))

(

#"2c20",

do\_diagnostic(e.1st, append\_bytearray(#"3a20", value)),

)

},

)

append\_bytearray(#"7b5f20", bytes)

}

}

},

{

// -------- List

let elems = un\_list\_data(self)

when elems is {

[] -> append\_bytearray(#"5b5d", builder)

\_ -> {

let (\_, bytes) =

list.foldr(

elems,

(#"5d", builder),

fn(e: Data, st: (ByteArray, ByteArray)) {

(#"2c20", do\_diagnostic(e, append\_bytearray(st.1st, st.2nd)))

},

)

append\_bytearray(#"5b5f20", bytes)

}

}

},

// -------- Integer

self

|> un\_i\_data

|> from\_int(builder),

{

// -------- ByteArray

let bytes = un\_b\_data(self)

bytes

|> encode\_base16(

length\_of\_bytearray(bytes) - 1,

append\_bytearray(#"27", builder),

)

|> append\_bytearray(#"6827", \_)

},

)

}

fn encode\_base16(bytes: ByteArray, ix: Int, builder: ByteArray) -> ByteArray {

if ix < 0 {

builder

} else {

let byte = index\_bytearray(bytes, ix)

let msb = byte / 16

let lsb = byte % 16

let builder =

cons\_bytearray(

msb + if msb < 10 {

48

} else {

55

},

cons\_bytearray(

lsb + if lsb < 10 {

48

} else {

55

},

builder,

),

)

encode\_base16(bytes, ix - 1, builder)

}

}

fn from\_int(i: Int, digits: ByteArray) -> ByteArray {

if i == 0 {

append\_bytearray(#"30", digits)

} else if i < 0 {

append\_bytearray(#"2d", from\_int(-i, digits))

} else {

do\_from\_int(

quotient\_integer(i, 10),

cons\_bytearray(remainder\_integer(i, 10) + 48, digits),

)

}

}

fn do\_from\_int(i: Int, digits: ByteArray) -> ByteArray {

if i <= 0 {

digits

} else {

do\_from\_int(

quotient\_integer(i, 10),

cons\_bytearray(remainder\_integer(i, 10) + 48, digits),

)

}

}

test diagnostic\_1() {

diagnostic(42) == @"42"

}

test diagnostic\_2() {

diagnostic(#"a1b2") == @"h'A1B2'"

}

test diagnostic\_3() {

diagnostic([1, 2, 3]) == @"[\_ 1, 2, 3]"

}

test diagnostic\_4() {

diagnostic([]) == @"[]"

}

test diagnostic\_5() {

diagnostic((1, 2)) == @"[\_ 1, 2]"

}

test diagnostic\_6() {

diagnostic((1, #"ff", 3)) == @"[\_ 1, h'FF', 3]"

}

test diagnostic\_7() {

diagnostic([(1, #"ff")]) == @"[\_ [\_ 1, h'FF']]"

}

test diagnostic\_7\_alt() {

diagnostic([Pair(1, #"ff")]) == @"{\_ 1: h'FF' }"

}

test diagnostic\_8() {

diagnostic(Some(42)) == @"121([\_ 42])"

}

test diagnostic\_9() {

diagnostic(None) == @"122([])"

}

test diagnostic\_10() {

let xs: List<(Int, Int)> =

[]

diagnostic(xs) == @"[]"

}

test diagnostic\_10\_alt() {

let xs: Pairs<Int, Int> =

[]

diagnostic(xs) == @"{}"

}

type Foo {

foo: Bar,

}

type Bar {

A

B(Int)

}

test diagnostic\_11() {

diagnostic(Foo { foo: A }) == @"121([\_ 121([])])"

}

test diagnostic\_12() {

diagnostic(Foo { foo: B(42) }) == @"121([\_ 122([\_ 42])])"

}

type Baz {

a0: Int,

b0: ByteArray,

}

test diagnostic\_13() {

diagnostic(Baz { a0: 14, b0: #"ff" }) == @"121([\_ 14, h'FF'])"

}

test diagnostic\_14() {

diagnostic([0]) == @"[\_ 0]"

}

test diagnostic\_15() {

diagnostic(-42) == @"-42"

}

test diagnostic\_16() {

diagnostic([-1, 0, 1]) == @"[\_ -1, 0, 1]"

}