symbiotic-data

Release 0.0.0

Feb 17, 2020

Contents:

Symbiotic-Data is an initiative intending to standardize primitive data types and their serialization formats, on different platforms, programming languages, and target encodings.

CHAPTER 1

Quick Links

GitHub	github.com/symbiotic-data
Home Page	symbiotic-data.io
Documentation	docs.symbiotic-data.io

CHAPTER 2

Motivation

Different platforms have different paradigms, capabilities, preconceptions, and opinions on how data should be formatted and stored internally, and this affects how the language or platform naturally tries to encode that data. This project aims to clarify some of those differences, and formalize data serialization standards. It is trying to act as a Rosetta Stone for various languages.

Most implementations of Symbiotic Data should be as painless and mostly cost-free — ideally being a *no-op*. But for some instances, it is necessary to restrict and refine a platform's natural understanding of a data type, in order to meet the criteria that constitutes its definition in Symbiotic-Data.

Similarly, serialization and de-serialization for Symbiotic-Data should be as simple, intuitive, and fast as possible, and rigorously defined for every data type.

CHAPTER 3

Supported Types

All supported types can be found in the documentation under *Data-Types*. The following classes of data are defined:

- *Primitives*: Elementary data types supported by most programming languages — Integers, Floating-point numbers, Boolean values, and Strings.
- *Casual Data*: Every-day useful data types for various tasks — Chronological data (like time, dates, etc.), and Network-relevant data (URIs, email addresses, etc.) are included under this classification.
- *Primitive Composites*: Elementary *container* data types — Fixed-size Arrays, Varying-size Vectors, Tuples, and Tagged Unions.

  We understand not every language supports parametric polymorphism and cannot implement "container types". However, the behavior is defined such that the user's specialization is still qualified under the same umbrella.
- *Sophisticated Composites*: Specialty container data types — Mappings, Recursive Mappings (Tries), and other complex containers may be placed in here.

  Sets (unique vectors) are not included due to this being a *serialization* definition library, not a data type library in generally. Similarly, qualified mappings are also not included (i.e. a `HashMap`), because we are only concerned with the different methods of serializing data, not storing it.

CHAPTER 4

Supported Target Encodings

Currently, only two target encodings are defined for Symbiotic-Data:

- JSON (JavaScript Object Notation), because of its wide utility and usage throughout the Internet — it is also a very likely suspect for incorrect encodings between languages.
- Raw binary stream — binary serialization is the foundation for all data storage and communication, and is therefore the foundation for this project as well. All types have strict definitions for how they're stored in a byte-wise (8-bit) string.

CHAPTER 5

Test Suite

The backbone of the *insurance policy* that guarantees these types are indeed *correctly* serialized and interpreted is due to the *Symbiote Test Suite*: a protocol for randomly generating data, and communicating that data over-the-wire to another platform.

Every implementation of Symbiotic-Data should also implement this test suite and protocol, to verify its correct behavior in accordance with the standard. Every *data type* in the standard declares what operations it supports.

5.1 Properties

The types defined in Symbiotic-Data may satisfy the definitions of certain algebraic objects, and by extension should have certain properties of behavior associated with their operations. You can see what properties and objects we use in the test suites in the *Algebraic Properties* page. Every *data type* in the standard declares what instances it supports, and by extension, how they are used in the test suite as testable operations.

Todo: Define the security and client identifier enrichment mechanisms for each serialization target;

ZeroMQ is used for binary data, over CurveMQ, with the server public key available in a TXT DNS record.

WebSocket is used for JSON, and secured through a TLS connections. However, the testing protocol is enriched with a UUID tag for identifying the client connected, similar to Router-Dealer sockets in ZeroMQ.

Todo: List the reference implementation server links for the serialization targets: `wss://ws.symbiotic-data.io` and `tcp://zmq.symbiotic-data.io`

5.2 Supported Platforms

Currently the supported platforms are:

- Haskell
  - symbiotic data implementation
  - symbiote test suite
- PureScript
  - symbiote test suite

There are planned implementations for:

- JavaScript (by use of the PureScript implementation)
- Rust
- Java / Clojure / Kotlin
- Objective-C / Swift
- C# / F#
- C / C++
- PHP
- Ruby
- Python
- Go
- D

If you have any other requested implementations or contributions, feel free to open an issue or pull request!

5.2.1 Symbiotic Data

Primitives

Unit

```
data Unit = Unit
```

JSON

Treats `Unit` as an empty string `""`

```
encodeJson :: Unit -> Json
encodeJson Unit = stringAsJson ""
```

Binary

Stores it as the byte `0`

```
encodeBinary :: Unit -> ByteString
encodeBinary Unit = byteAsByteString 0
```

Symbiote Test Suite Instance

Instances

This data type implements a number of instances from the *Algebraic Properties* specification:

```
instance Semigroup Unit where
  append Unit Unit = Unit
instance Monoid Unit where
  mempty = Unit
instance Eq Unit where
  eq Unit Unit = True
instance Ord Unit where
  compare Unit Unit = EQ
instance Enum Unit where
  pred Unit = Unit
  succ Unit = Unit
instance Bounded Unit where
  top = Unit
  bottom = Unit
instance BoundedEnum Unit where
  toEnum 0 = Just Unit
  toEnum _ = Nothing
  fromEnum Unit = 0
instance HeytingAlgebra Unit where
  ff = Unit
  tt = Unit
  implies Unit Unit = Unit
  disj Unit Unit = Unit
  conj Unit Unit = Unit
  not Unit = Unit
instance BooleanAlgebra Unit
instance Semiring Unit where
  zero = Unit
  one = Unit
  add Unit Unit = Unit
  mul Unit Unit = Unit
instance Ring Unit where
  sub Unit Unit = Unit
instance CommutativeRing Unit
```

Operations

```
data UnitOperation
  = UnitMonoid (MonoidOperation Unit)
  | UnitBoundedEnum (BoundedEnumOperation Unit)
  | UnitBooleanAlgebra (BooleanAlgebraOperation Unit)
  | UnitCommutativeRing  (CommutativeRingOperation Unit)

performUnit :: UnitOperation -> Unit -> Bool
performUnit op x = case op of
  UnitMonoid op' ->
    performMonoid op' x
  UnitBoundedEnum op' ->
    performBoundedEnum op' x
```

(continues on next page)

(continued from previous page)

```
  UnitBooleanAlgebra op' ->
    performBooleanAlgebra op' x
  UnitCommutativeRing op' ->
    performCommutativeRing op' x

encodeJson :: UnitOperation -> Json
encodeJson op = case op of
  UnitMonoid op' ->
    {"monoid": encodeJson op'}
  UnitBoundedEnum op' ->
    {"boundedEnum": encodeJson op'}
  UnitBooleanAlgebra op' ->
    {"booleanAlgebra": encodeJson op'}
  UnitCommutativeRing op' ->
    {"commutativeRing": encodeJson op'}

encodeBinary :: UnitOperation -> ByteString
encodeBinary op = case op of
  UnitMonoid op' ->
    (byteToByteString 0) ++ encodeBinary op'
  UnitBoundedEnum op' ->
    (byteToByteString 1) ++ encodeBinary op'
  UnitBooleanAlgebra op' ->
    (byteToByteString 2) ++ encodeBinary op'
  UnitCommutativeRing op' ->
    (byteToByteString 3) ++ encodeBinary op'
```

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Unit`.

```
"Unit"
```

Boolean

```
data Boolean = True | False
```

Note: Although throughout this text, in Haskell, we functionally use the `Bool` type to denote true/false values; that's only due to that language's design. For all intents and purposes, a Symbiotic-Data `Boolean` value is *functionally* the same as a operational true/false value, we opt to decide its name as `Boolean`, as opposed to `Bool`, specifically to distinguish it from a programming language's notions.

An implementation need not write a new data type for this Symbiotic-Data specification, however; it just needs to associate its *Topic* value as `Boolean`.

JSON

Uses standard JSON Boolean

```
encodeJson :: Boolean -> Json
encodeJson x = case x of
  True -> true
  False -> false
```

Binary

Uses standard bytes `0` as false, `1` as true

```
encodeBinary :: Boolean -> ByteString
encodeBinary x = case x of
  True -> byteAsByteString 1
  False -> byteAsByteString 0
```

Symbiote Test Suite Instance

Instances

This data type implements a number of instances from the *Algebraic Properties* specification:

```
instance Eq Boolean where
  eq True  True  = True
  eq False False = True
  eq _     _     = False
instance Ord Boolean where
  compare False True  = LT
  compare True  False = GT
  compare _     _     = EQ
instance Enum Unit where
  pred False = False
  pred True  = False
  succ False = True
  succ True  = True
instance Bounded Unit where
  top = True
  bottom = False
instance BoundedEnum Unit where
  toEnum 0 = Just False
  toEnum 1 = Just True
  toEnum _ = Nothing
  fromEnum False = 0
  fromEnum True  = 1
instance HeytingAlgebra Unit where
  ff = False
  tt = True
  implies p q = if p then q else True
  disj False False = False
  disj _     _     = True
  conj True True = True
  conj _    _    = False
  not False = True
  not True  = False
instance BooleanAlgebra Unit
```

Operations

```haskell
data BooleanOperation
  = BooleanBoundedEnum (BoundedEnumOperation Boolean)
  | BooleanBooleanAlgebra (BooleanAlgebraOperation Boolean)

performBoolean :: BooleanOperation -> Boolean -> Bool
performBoolean op x = case op of
  BooleanBoundedEnum op' ->
    performBoundedEnum op' x
  BooleanBooleanAlgebra op' ->
    performBooleanAlgebra op' x

encodeJson :: BooleanOperation -> Json
encodeJson op = case op of
  BooleanBoundedEnum op' ->
    {"boundedEnum": encodeJson op'}
  BooleanBooleanAlgebra op' ->
    {"booleanAlgebra": encodeJson op'}

encodeBinary :: BooleanOperation -> ByteString
encodeBinary op = case op of
  BooleanBoundedEnum op' ->
    (byteToByteString 1) ++ encodeBinary op'
  BooleanBooleanAlgebra op' ->
    (byteToByteString 2) ++ encodeBinary op'
```

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Boolean`.

```
"Boolean"
```

Integral

Signed

Int8

Range varies from `-2^7` to `2^7-1`

```haskell
data Int8 = -2^7 | -2^7+1 | ... | 2^7-2 | 2^7-1
```

JSON

Uses standard JSON Integers

```haskell
encodeJson :: Int8 -> Json
encodeJson x = intAsJson x
```

Binary

`Int8` s are converted to *Uint8's before storing as a byte - where the negative range is stored as the upper values in the ''Uint8'.*

```
encodeBinary :: Int8 -> ByteString
encodeBinary x =
  if x >= 0
  then byteAsByteString (intAsUint x)
  else byteAsByteString ((intAsUint (2^7 + x)) + 2^7)
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Int8`.

```
"Int8"
```

Int16

Range varies from `-2^15` to `2^15-1`

```
data Int16 = -2^15 | -2^15+1 | ... | 2^15-2 | 2^15-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Int16 -> Json
encodeJson x = intAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Int16 -> ByteString
encodeBinary x = intAsByteStringBE x
```

```
encodeBinary :: Int16 -> ByteString
encodeBinary x = intAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Int16`.

```
"Int16"
```

Int32

Range varies from `-2^31` to `2^31-1`

```
data Int32 = -2^31 | -2^31+1 | ... | 2^31-2 | 2^31-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Int32 -> Json
encodeJson x = intAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Int32 -> ByteString
encodeBinary x = intAsByteStringBE x
```

```
encodeBinary :: Int32 -> ByteString
encodeBinary x = intAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Int32`.

```
"Int32"
```

Int64

Range varies from `-2^63` to `2^63-1`

```
data Int64 = -2^63 | -2^63+1 | ... | 2^63-2 | 2^63-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Int64 -> Json
encodeJson x = intAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Int64 -> ByteString
encodeBinary x = intAsByteStringBE x
```

```
encodeBinary :: Int64 -> ByteString
encodeBinary x = intAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Int64`.

```
"Int64"
```

Unsigned

Uint8

Range varies from `0` to `2^8-1`

```
data Uint8 = 0 | 1 | ... | 2^8-2 | 2^8-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Uint8 -> Json
encodeJson x = uintAsJson x
```

Binary

```
encodeBinary :: Uint8 -> ByteString
encodeBinary x = byteAsByteString x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Uint8`.

```
"Uint8"
```

Uint16

Range varies from `0` to `2^16-1`

```

```

data Uint16 = 0 | 1 | . . . | 2^16-2 | 2^16-1

JSON

Uses standard JSON Integers

```
encodeJson :: Uint16 -> Json
encodeJson x = uintAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Uint16 -> ByteString
encodeBinary x = uintAsByteStringBE x
```

```
encodeBinary :: Uint16 -> ByteString
encodeBinary x = uintAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Uint16`.

```
"Uint16"
```

Uint32

Range varies from `0` to `2^32-1`

```
data Uint32 = 0 | 1 | ... | 2^32-2 | 2^32-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Uint32 -> Json
encodeJson x = uintAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Uint32 -> ByteString
encodeBinary x = uintAsByteStringBE x
```

```
encodeBinary :: Uint32 -> ByteString
encodeBinary x = uintAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Uint32`.

```
"Uint32"
```

Uint64

Range varies from `0` to `2^64-1`

```
data Uint64 = 0 | 1 | ... | 2^64-2 | 2^64-1
```

JSON

Uses standard JSON Integers

```
encodeJson :: Uint64 -> Json
encodeJson x = uintAsJson x
```

Binary

There are two byte encodings for any integer larger than 8 bits - big-endian or little-endian.

```
encodeBinary :: Uint64 -> ByteString
encodeBinary x = uintAsByteStringBE x
```

```
encodeBinary :: Uint64 -> ByteString
encodeBinary x = uintAsByteStringLE x
```

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Uint64`.

```
"Uint64"
```

Multiple Precision

Integer8

Arbitrary precision signed integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^8` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Integer8 -> Json
encodeJson x = stringAsJson (integerAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is an 8-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Integer8`.

```
"Integer8"
```

Integer16

Arbitrary precision signed integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^16` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Integer16 -> Json
encodeJson x = stringAsJson (integerAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 16-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Integer16`.

```
"Integer16"
```

Integer32

Arbitrary precision signed integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^32` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Integer32 -> Json
encodeJson x = stringAsJson (integerAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 32-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Integer32`.

```
"Integer32"
```

Integer64

Arbitrary precision signed integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^64` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Integer64 -> Json
encodeJson x = stringAsJson (integerAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 64-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Integer64`.

```
"Integer64"
```

Natural8

Arbitrary precision unsigned integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^8` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Natural8 -> Json
encodeJson x = stringAsJson (naturalAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is an 8-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Natural8`.

```
"Natural8"
```

Natural16

Arbitrary precision unsigned integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^16` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Natural16 -> Json
encodeJson x = stringAsJson (naturalAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 16-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Natural16`.

```
"Natural16"
```

Natural32

Arbitrary precision unsigned integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^32` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Natural32 -> Json
encodeJson x = stringAsJson (naturalAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 32-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Natural32`.

```
"Natural32"
```

Natural64

Arbitrary precision unsigned integer, implemented as (for instance) GNU MP, but with a max unrolled length of `2^64` bytes long.

JSON

Uses a string encoding of the integer value, because not every platform can support very large integer values during JSON decoding.

```
encodeJson :: Natural64 -> Json
encodeJson x = stringAsJson (naturalAsString x)
```

Binary

Performed via cereal byte-unrolling, but with the concern that the length of unrolled bytes is a 64-bit unsigned integer.

Symbiote Test Suite Instance

- Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Natural64`.

```
"Natural64"
```

Floating Point

Float32

A binary32 implementation of IEEE 754

JSON

Uses standard JSON Numbers

```
encodeJson :: Float32 -> Json
encodeJson x = floatAsJson x
```

Binary

There are two byte encodings for any floating point number - big-endian or little-endian.

```
encodeBinary :: Float32 -> ByteString
encodeBinary x = floatAsByteStringBE x
```

```
encodeBinary :: Float32 -> ByteString
encodeBinary x = floatAsByteStringLE x
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Float32`.

```
"Float32"
```

Float64

A binary64 implementation of IEEE 754

JSON

Uses standard JSON Numbers

```
encodeJson :: Float64 -> Json
encodeJson x = floatAsJson x
```

Binary

There are two byte encodings for any floating point number - big-endian or little-endian.

```
encodeBinary :: Float64 -> ByteString
encodeBinary x = floatAsByteStringBE x
```

```
encodeBinary :: Float64 -> ByteString
encodeBinary x = floatAsByteStringLE x
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Float64`.

```
"Float64"
```

Scientific

A scientific notation implementation

JSON

Encoded as a JSON String, in canonical scientific notation - an exponential field (`*10^n`) is always present, even when `n == 0`, and prefixes its sign in all cases (i.e. `9e3` is `9e+3`). Likewise, the coefficient is always `-10 < c < 10` - no engineering notation is allowed. Furthermore, the coefficient _never_ includes trailing zeros - i.e. `9.230e+0` is `9.23e+0`. Moreover, when the value clearly doesn't need a decimal place, it should be omitted - i.e. `9.0e+3` is `9e+3`.

```
encodeJson :: Scientific -> Json
encodeJson x = stringAsJson (scientificToString x)
```

Binary

Uses the same UTF8 string format as JSON, but limited to a *String32*.

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Scientific`.

```
"Scientific"
```

Ratio

A (lossless) rational number implementation, by ratios.

```
data Ratio a = Ratio a a
```

JSON

Encoded as a tuple of the two already encoded values

```
encodeJson :: Ratio Json -> Json
encodeJson (Ratio x y) = [x,y]
```

Binary

Encoded as a tuple of the two already encoded values

```
encodeBinary :: Ratio ByteString -> ByteString
encodeBinary (Ratio x y) = x ++ y
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Ratio Int32 Int32`.

```
"Ratio"
```

UTF-8 Strings

Char

All characters must be valid UTF-8 characters, especially with respect to surrogate codes between `0xD800` and `0xDFFF` - with respect to RFC 3629. Conversion a 'la CESU-8 may or may not be defined with this data type.

JSON

Uses standard JSON Strings

```
encodeJson :: Char -> Json
encodeJson x = charAsJson x
```

Binary

Encodes to a ByteString as standard UTF-8.

```
encodeBinary :: Char -> ByteString
encodeBinary x = utf8AsByteString x
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Char`.

```
"Char"
```

String8

Where the length of the string is at most `2^8` characters long

```
data String8 = Vector8 Char
```

JSON

Uses standard JSON Strings

```
encodeJson :: String8 -> Json
encodeJson x = stringAsJson x
```

Binary

Encodes to a ByteString as a `Vector8` of `Char` s

```
encodeBinary :: String8 -> ByteString
encodeBinary x = vector8ToByteString (map utf8AsByteString (string8AsVector8 x))
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `String8`.

```
"String8"
```

String16

Where the length of the string is at most `2^16` characters long

```
data String16 = Vector16 Char
```

JSON

Uses standard JSON Strings

```
encodeJson :: String16 -> Json
encodeJson x = stringAsJson x
```

Binary

Encodes to a ByteString as a `Vector16` of `Char` s

```
encodeBinary :: String16 -> ByteString
encodeBinary x = vector16ToByteString (map utf8AsByteString (string16AsVector16 x))
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `String16`.

```
"String16"
```

String32

Where the length of the string is at most `2^32` characters long

```
data String32 = Vector32 Char
```

JSON

Uses standard JSON Strings

```
encodeJson :: String32 -> Json
encodeJson x = stringAsJson x
```

Binary

Encodes to a ByteString as a `Vector32` of `Char` s

```
encodeBinary :: String32 -> ByteString
encodeBinary x = vector32ToByteString (map utf8AsByteString (string32AsVector32 x))
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `String32`.

```
"String32"
```

String64

Where the length of the string is at most `2^64` characters long

```
data String64 = Vector64 Char
```

JSON

Uses standard JSON Strings

```
encodeJson :: String64 -> Json
encodeJson x = stringAsJson x
```

Binary

Encodes to a ByteString as a `Vector64` of `Char` s

```
encodeBinary :: String64 -> ByteString
encodeBinary x = vector64ToByteString (map utf8AsByteString (string64AsVector64 x))
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `String64`.

```
"String64"
```

Casual

Chronological

Date

Any date system that keeps track of year, month, and day. Years are biased in the Common Era, and can range from `-2^15` to `2^15-1`.

```
data Date = Date
  (year :: Int16)
  (month :: Uint8)
  (day :: Uint8)
```

JSON

Formatted as an ISO 8601 Calendar Date / "military date" string `YYYYMMDD`.

```
encodeJson :: Date -> Json
encodeJson x = stringAsJson (iso8601 "YYYYMMDD" x)
```

Binary

Encoded directly as one 16-bit signed integer as the year, and two bytes as the month and day. Although there could be a way to encode a practical calendar date as 21-bits (using a 13-bit year, 4-bit month, and 5-bit day), the conversions would be considerable overhead when dealing with large amounts of date data. And "practical" in the sense of Ancient History (3000 B.C.E.) being the limit of dating capability.

```
encodeByteString :: Date -> ByteString
encodeByteString (Date year month day) =
  (intAsByteStringBE year)
    ++ (uintAsByteString month)
    ++ (uintAsByteString day)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Date`.

```
"Date"
```

Time

Any time system that keeps track of timezone, hour, minute, second, and millisecond. Milliseconds are included because most modern systems can emit logs with millisecond precision, and is a likely use case.

```
data Time = Time
  (tzhour :: Int8)
  (tzminute :: Uint8)
  (hour :: Uint8)
  (minute :: Uint8)
  (second :: Uint8)
  (millisecond :: Uint16)
```

JSON

Formatted as an ISO 8601 Time string `hhmmss.sss`.

```
encodeJson :: Time -> Json
encodeJson x = stringAsJson (iso8601 "hhmmss.sss" x)
```

Binary

Encoded directly as 5 bytes for timezone, hour, minute, and second, and one 16-bit unsigned integer for milliseconds. Although there could be a way to encode a practical time as 38-bits (5-bit hour and tzhour, 6-bit minute, tzminute and second, 10-bit millisecond), the conversions would be considerable overhead when dealing with large amounts of time data.

```
encodeByteString :: Time -> ByteString
encodeByteString
  (Time tzhour tzminute hour minute second millisecond) =
    (intAsByteString tzhour)
      ++ (uintAsByteString tzminute)
      ++ (uintAsByteString hour)
      ++ (uintAsByteString minute)
      ++ (uintAsByteString second)
      ++ (uintAsByteStringBE millisecond)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `Time`.

```
"Time"
```

DateTime

Can be represented internally as any "sane" date / time system.

```
data DateTime = Tuple Date Time
```

JSON

Formatted as an ISO 8601 Combined String

```
encodeJson :: DateTime -> Json
encodeJson x = stringAsJson (iso8601 x)
```

Binary

Concatenation of both formats (total of 11 bytes).

```
encodeByteString :: DateTime -> ByteString
encodeByteString (Tuple date time) =
  (encodeByteStringDate date)
    ++ (encodeByteStringTime time)
```

Todo:

- Intervals, Durations

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `DateTime`.

```
"DateTime"
```

URI-Like

IPV4

```
data IPV4 = IPV4 Uint8 Uint8 Uint8 Uint8
```

JSON

Formatted as a string to remain unambiguous

```
encodeJson :: IPV4 -> Json
encodeJson x = stringAsJson (ipv4AsString x)
```

Binary

Encoded directly as 4 bytes

```
encodeByteString :: IPV4 -> ByteString
encodeByteString (IPV4 a b c d) =
  (uintAsByteStringBE a)
    ++ (uintAsByteStringBE b)
    ++ (uintAsByteStringBE c)
    ++ (uintAsByteStringBE d)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `IPV4`.

```
"IPV4"
```

IPV6

```
data IPV6 =
  IPV6
    Uint16 Uint16 Uint16 Uint16
    Uint16 Uint16 Uint16 Uint16
```

JSON

Formatted as a string to remain unambiguous

```
encodeJson :: IPV6 -> Json
encodeJson x = stringAsJson (ipv6AsString x)
```

Binary

Encoded directly as 16 bytes

```
encodeByteString :: IPV6 -> ByteString
encodeByteString (IPV6 a b c d e f g h) =
  (uintAsByteStringBE a)
    ++ (uintAsByteStringBE b)
    ++ (uintAsByteStringBE c)
    ++ (uintAsByteStringBE d)
    ++ (uintAsByteStringBE e)
    ++ (uintAsByteStringBE f)
    ++ (uintAsByteStringBE g)
    ++ (uintAsByteStringBE h)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `IPV6`.

```
"IPV6"
```

URI

Should be a valid URI with Percent Encoding for all reserved, non-valid, and UTF-8 characters in their appropriate components in the URI, while the query may have `x-www-form-urlencoded` data.

JSON

Formatted as its string representation

```
encodeJson :: URI -> Json
encodeJson x = stringAsJson (uriAsString x)
```

Binary

Encoded as a UTF-8 *String32* (though there are only ASCII characters allowed) - other implementations of `URI8` etc may exist in a future version.

```
encodeByteString :: URI -> ByteString
encodeByteString x = utf8AsByteString (uriAsString x)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `URI`.

```
"URI"
```

Email Address

Should be represented in a vaild ASCII Email Address format, as per Wikipedia / RFC 5322.

JSON

Formatted as the string representation

```
encodeJson :: EmailAddress -> Json
encodeJson x = stringAsJson (emailAddressAsString x)
```

Binary

Encoded as a UTF-8 `String16` (though there are only ASCII characters allowed)

```
encodeByteString :: EmailAddress -> ByteString
encodeByteString x = utf8AsByteString (emailAddressAsString x)
```

Todo:

- International Email Addresses a ‘la https://en.wikipedia.org/wiki/International_email

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed solely as `EmailAddress`.

```
"EmailAddress"
```

Primitive Composites

Collections

Array

A size-indexed array of homogeneous data.

```
data Array (n :: Nat) a where
  Nil :: Array 0 a
  Cons :: a -> Array n a -> Array (n + 1) a
```

JSON

Uses standard JSON Arrays

```
encodeJson :: Array n Json -> Json
encodeJson x = arrayAsJson x
```

Binary

Ommits a size parameter, because the size is encoded in the type signature.

```
encodeBinary :: Array n ByteString -> ByteString
encodeBinary x = case x of
  Nil -> emptyByteString
  Cons y ys -> y ++ (encodeBinary ys)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Array 20 Int32` (length of 20 `Int32` s).

```
"Array"
```

Vector8

A dynamically sized array that limits the max size to `2^8` elements

JSON

Uses standard JSON Arrays

```
encodeJson :: Vector8 Json -> Json
encodeJson x = arrayAsJson x
```

Binary

Prefixes the length of the array as a 8-bit unsigned integer, big-endian, before concatenating all contents.

```
encodeBinary :: Vector8 ByteString -> ByteString
encodeBinary x = (uintAsByteStringBE l) ++ (concatVector8 x)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Vector8 Int32`.

```
"Vector8"
```

Vector16

A dynamically sized array that limits the max size to `2^16` elements

JSON

Uses standard JSON Arrays

```
encodeJson :: Vector16 Json -> Json
encodeJson x = arrayAsJson x
```

Binary

Prefixes the length of the array as a 16-bit unsigned integer, big-endian, before concatenating all contents.

```
encodeBinary :: Vector16 ByteString -> ByteString
encodeBinary x = (uintAsByteStringBE l) ++ (concatVector16 x)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Vector16 Int32`.

```
"Vector16"
```

Vector32

A dynamically sized array that limits the max size to `2^32` elements

JSON

Uses standard JSON Arrays

```
encodeJson :: Vector32 Json -> Json
encodeJson x = arrayAsJson x
```

Binary

Prefixes the length of the array as a 32-bit unsigned integer, big-endian, before concatenating all contents.

```
encodeBinary :: Vector32 ByteString -> ByteString
encodeBinary x = (uintAsByteStringBE l) ++ (concatVector32 x)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Vector32 Int32`.

```
"Vector32"
```

Vector64

A dynamically sized array that limits the max size to `2^64` elements

JSON

Uses standard JSON Arrays

```
encodeJson :: Vector64 Json -> Json
encodeJson x = arrayAsJson x
```

Binary

Prefixes the length of the array as a 64-bit unsigned integer, big-endian, before concatenating all contents.

```
encodeBinary :: Vector64 ByteString -> ByteString
encodeBinary x = (uintAsByteStringBE l) ++ (concatVector64 x)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Vector64 Int32`.

```
"Vector64"
```

Maybe

Standard option type

```
data Maybe a = Nothing | Maybe a
```

JSON

Uses standard JSON `null` if `Nothing`, otherwise just use its JSON - leverages backtracking

```
encodeJson :: Maybe Json -> Json
encodeJson x = case x of
  Nothing -> nullJson
  Just y -> y
```

Binary

Use a prefix byte flag to avoid backtracking

```
encodeBinary :: Maybe ByteString -> ByteString
encodeBinary x = case x of
  Nothing -> byteAsByteString 0
  Just y -> (byteAsByteString 1) ++ y
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Maybe Int32`.

```
"Maybe"
```

Tuple

```haskell
data Tuple a b = Tuple a b
```

JSON

Uses a standard JSON Array to hold the two elements

```haskell
encodeJson :: Tuple Json Json -> Json
encodeJson (Tuple x y) = [x,y]
```

Binary

Is equivalent to an array of size 2, therefore avoids a size prefix

```haskell
encodeBinary :: Tuple ByteString ByteString -> ByteString
encodeBinary (Tuple x y) = x ++ y
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Tuple Int32 Int32`.

```
"Tuple"
```

Either

```haskell
data Either a b = Left a | Right b
```

JSON

Flags each case with a unique object key

```
encodeJson :: Either Json Json -> Json
encodeJson x = case x of
  Left y -> {"l": y}
  Right z -> {"r": z}
```

Binary

Flags each case with a byte prefix

```
encodeBinary :: Either ByteString ByteString -> ByteString
encodeBinary x = case x of
  Left y -> (byteAsByteString 0) ++ y
  Right z -> (byteAsByteString 1) ++ z
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Either Int32 Int32`.

```
"Either"
```

Sophisticated Composites

Mappings

StringMap8

Mapping where `String8` s are the keys - can be implemented as a hash-map internally, or as a JSON object as the case with JavaScript.

JSON

Serialized as a JSON object

```
encodeJson :: StringMap8 Json -> Json
encodeJson x = stringMap8AsJson x
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 8-bit unsigned integer.

```
encodeBinary :: StringMap8 ByteString -> ByteString
encodeBinary x = concatVector8 (map tupleToByteString (stringMap8AsVector8 x))
  where
    tupleToByteString :: Tuple String ByteString -> ByteString
    tupleToByteString (Tuple k v) = (encodeByteString k) ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringMap8 Int32`.

```
"StringMap8"
```

StringMap16

Mapping where `String16` s are the keys - can be implemented as a hash-map internally, or as a JSON object as the case with JavaScript.

JSON

Serialized as a JSON object

```
encodeJson :: StringMap16 Json -> Json
encodeJson x = stringMap16AsJson x
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 16-bit unsigned integer.

```
encodeBinary :: StringMap16 ByteString -> ByteString
encodeBinary x = concatVector16 (map tupleToByteString (stringMap16AsVector16 x))
  where
    tupleToByteString :: Tuple String ByteString -> ByteString
    tupleToByteString (Tuple k v) = (encodeByteString k) ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringMap16 Int32`.

```
"StringMap16"
```

StringMap32

Mapping where *String32* s are the keys - can be implemented as a hash-map internally, or as a JSON object as the case with JavaScript.

JSON

Serialized as a JSON object

```
encodeJson :: StringMap32 Json -> Json
encodeJson x = stringMap32AsJson x
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 32-bit unsigned integer.

```
encodeBinary :: StringMap32 ByteString -> ByteString
encodeBinary x = concatVector32 (map tupleToByteString (stringMap32AsVector32 x))
  where
    tupleToByteString :: Tuple String ByteString -> ByteString
    tupleToByteString (Tuple k v) = (encodeByteString k) ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringMap32 Int32`.

```
"StringMap32"
```

StringMap64

Mapping where `String64` s are the keys - can be implemented as a hash-map internally, or as a JSON object as the case with JavaScript.

JSON

Serialized as a JSON object

```
encodeJson :: StringMap64 Json -> Json
encodeJson x = stringMap64AsJson x
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 64-bit unsigned integer.

```
encodeBinary :: StringMap64 ByteString -> ByteString
encodeBinary x = concatVector64 (map tupleToByteString (stringMap64AsVector64 x))
  where
    tupleToByteString :: Tuple String ByteString -> ByteString
    tupleToByteString (Tuple k v) = (encodeByteString k) ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringMap64 Int32`.

```
"StringMap64"
```

Map8

Polymorphic mapping - can be implemented any way: B-Tree, or unordered - serialization does not restrict the implementation.

JSON

Serialized as an array of arrays / tuples.

```
encodeJson :: Map8 Json Json -> Json
encodeJson x = map tupleToJson (map8AsVector8 x)
  where
    tupleToJson :: Tuple Json Json -> Json
    tupleToJson (Tuple k v) = [k,v]
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 8-bit unsigned integer.

```
encodeBinary :: Map8 ByteString ByteString -> ByteString
encodeBinary x = concatVector8 (map tupleToByteString (map8AsVector8 x))
  where
    tupleToByteString :: Tuple ByteString ByteString -> ByteString
    tupleToByteString (Tuple k v) = k ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Map8 Int32 Int32`.

```
"Map8"
```

Map16

Polymorphic mapping - can be implemented any way: B-Tree, or unordered - serialization does not restrict the implementation.

JSON

Serialized as an array of arrays / tuples.

```
encodeJson :: Map16 Json Json -> Json
encodeJson x = map tupleToJson (map16AsVector16 x)
  where
    tupleToJson :: Tuple Json Json -> Json
    tupleToJson (Tuple k v) = [k,v]
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 16-bit unsigned integer.

```
encodeBinary :: Map16 ByteString ByteString -> ByteString
encodeBinary x = concatVector16 (map tupleToByteString (map16AsVector16 x))
  where
    tupleToByteString :: Tuple ByteString ByteString -> ByteString
    tupleToByteString (Tuple k v) = k ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Map16 Int32 Int32`.

```
"Map16"
```

Map32

Polymorphic mapping - can be implemented any way: B-Tree, or unordered - serialization does not restrict the implementation.

JSON

Serialized as an array of arrays / tuples.

```
encodeJson :: Map32 Json Json -> Json
encodeJson x = map tupleToJson (map32AsVector32 x)
  where
    tupleToJson :: Tuple Json Json -> Json
    tupleToJson (Tuple k v) = [k,v]
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 32-bit unsigned integer.

```
encodeBinary :: Map32 ByteString ByteString -> ByteString
encodeBinary x = concatVector32 (map tupleToByteString (map32AsVector32 x))
  where
    tupleToByteString :: Tuple ByteString ByteString -> ByteString
    tupleToByteString (Tuple k v) = k ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Map32 Int32 Int32`.

```
"Map32"
```

Map64

Polymorphic mapping - can be implemented any way: B-Tree, or unordered - serialization does not restrict the implementation.

JSON

Serialized as an array of arrays / tuples.

```
encodeJson :: Map64 Json Json -> Json
encodeJson x = map tupleToJson (map64AsVector64 x)
  where
    tupleToJson :: Tuple Json Json -> Json
    tupleToJson (Tuple k v) = [k,v]
```

Binary

Encodes as a dynamically sized array of key-value tuples, where the size is a 64-bit unsigned integer.

```
encodeBinary :: Map64 ByteString ByteString -> ByteString
encodeBinary x = concatVector64 (map tupleToByteString (map64AsVector64 x))
  where
    tupleToByteString :: Tuple ByteString ByteString -> ByteString
    tupleToByteString (Tuple k v) = k ++ v
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Map64 Int32 Int32`.

```
"Map64"
```

Tries

StringTrie8

Recursive `StringMap8`, with values along the way.

```
data StringTrie8 a = StringMap8 (Tuple (Maybe a) (StringTrie8 a))
```

JSON

Uses a standard JSON Object as the key index

```
encodeJson :: StringTrie8 Json -> Json
encodeJson x = stringMap8AsObject (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (StringTrie8 Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: StringTrie8 ByteString -> ByteString
encodeByteString x = encodeByteStringVector8 (stringMap8AsVector8 (map␣
↪tupleToByteString x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (StringTrie8 ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringTrie8 Int32`.

```
"StringTrie8"
```

StringTrie16

Recursive `StringMap16`, with values along the way.

```
data StringTrie16 a = StringMap16 (Tuple (Maybe a) (StringTrie16 a))
```

JSON

Uses a standard JSON Object as the key index

```
encodeJson :: StringTrie16 Json -> Json
encodeJson x = stringMap16AsObject (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (StringTrie16 Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: StringTrie16 ByteString -> ByteString
encodeByteString x = encodeByteStringVector16 (stringMap16AsVector16 (map␣
↪tupleToByteString x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (StringTrie16 ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringTrie16 Int32`.

```
"StringTrie16"
```

StringTrie32

Recursive `StringMap32`, with values along the way.

```
data StringTrie32 a = StringMap32 (Tuple (Maybe a) (StringTrie32 a))
```

JSON

Uses a standard JSON Object as the key index

```
encodeJson :: StringTrie32 Json -> Json
encodeJson x = stringMap32AsObject (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (StringTrie32 Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: StringTrie32 ByteString -> ByteString
encodeByteString x = encodeByteStringVector32 (stringMap32AsVector32 (map␣
↪tupleToByteString x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (StringTrie32 ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringTrie32 Int32`.

```
"StringTrie32"
```

StringTrie64

Recursive `StringMap64`, with values along the way.

```
data StringTrie64 a = StringMap64 (Tuple (Maybe a) (StringTrie64 a))
```

JSON

Uses a standard JSON Object as the key index

```
encodeJson :: StringTrie64 Json -> Json
encodeJson x = stringMap64AsObject (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (StringTrie64 Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: StringTrie64 ByteString -> ByteString
encodeByteString x = encodeByteStringVector64 (stringMap64AsVector64 (map␣
↪tupleToByteString x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (StringTrie64 ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `StringTrie64 Int32`.

```
"StringTrie64"
```

Trie8

Recursive `Map8`, with values along the way.

```
data Trie8 k a = Map8 k (Tuple (Maybe a) (Trie8 k a))
```

JSON

Uses nested Arrays

```
encodeJson :: Trie8 Json Json -> Json
encodeJson x = map8AsVector8 (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (Trie8 Json Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: Trie8 ByteString ByteString -> ByteString
encodeByteString x = encodeByteStringVector8 (map8AsVector8 (map tupleToByteString x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (Trie8 ByteString ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Trie8 Int32 Int32`.

```
"Trie8"
```

Trie16

Recursive `Map16`, with values along the way.

```
data Trie16 k a = Map16 k (Tuple (Maybe a) (Trie16 k a))
```

JSON

Uses nested Arrays

```
encodeJson :: Trie16 Json Json -> Json
encodeJson x = map16AsVector16 (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (Trie16 Json Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: Trie16 ByteString ByteString -> ByteString
encodeByteString x = encodeByteStringVector16 (map16AsVector16 (map tupleToByteString␣
↪x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (Trie16 ByteString ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Trie16 Int32 Int32`.

```
"Trie16"
```

Trie32

Recursive `Map32`, with values along the way.

```
data Trie32 k a = Map32 k (Tuple (Maybe a) (Trie32 k a))
```

JSON

Uses nested Arrays

```
encodeJson :: Trie32 Json Json -> Json
encodeJson x = map32AsVector32 (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (Trie32 Json Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: Trie32 ByteString ByteString -> ByteString
encodeByteString x = encodeByteStringVector32 (map32AsVector32 (map tupleToByteString␣
↪x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (Trie32 ByteString ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Trie32 Int32 Int32`.

```
"Trie32"
```

Trie64

Recursive `Map64`, with values along the way.

```
data Trie64 k a = Map64 k (Tuple (Maybe a) (Trie64 k a))
```

JSON

Uses nested Arrays

```
encodeJson :: Trie64 Json Json -> Json
encodeJson x = map64AsVector64 (map tupleToJson x)
  where
    tupleToJson :: Tuple (Maybe Json) (Trie64 Json Json) -> Json
    tupleToJson (Tuple v y) = [maybeToJson v, encodeJson y]
```

Binary

Encoded as a series of dynamically sized arrays - uses composite `encodeByteString` instances for each level.

```
encodeByteString :: Trie64 ByteString ByteString -> ByteString
encodeByteString x = encodeByteStringVector64 (map64AsVector64 (map tupleToByteString␣
↪x))
  where
    tupleToByteString :: Tuple (Maybe ByteString) (Trie64 ByteString ByteString) ->␣
↪ByteString
    tupleToByteString (Tuple v y) = (maybeToByteString v) ++ (encodeByteString y)
```

Symbiote Test Suite Instance

Topic

String literal *Topic* used in the *Symbiote Test Suite* for this data type. Composed as `Trie64 Int32 Int32`.

```
"Trie64"
```

5.2.2 Algebraic Properties

The various data types defined in Symbiotic-Data form various algebraic objects, and by extension support certain properties over their operations.

The superclass structure is as follows:

```
Semigroup
  => Monoid

Eq
  => Ord
    => Enum ------\
                  => BoundedEnum
    => Bounded ---/

HeytingAlgebra
  => BooleanAlgebra

Semiring
  => Ring
    => DivisionRing ------\
    => CommutativeRing     => Field
      => EuclideanRing ---/
```

That is, a `Field` is both a `DivisionRing` and a `EuclideanRing`, following the PureScript taxonomy.

Semigroup

Semigroups have one operation defined on it, `append`, and it should be associative.

```
class Semigroup a where
  append :: a -> a -> a
```

(continues on next page)

(continued from previous page)

```
isAssociative :: Semigroup a =>
  a -> a -> a -> Bool
isAssociative x y z =
  append (append x y) z == append x (append y z)
```

Operations

```
data SemigroupOperation a
  = SemigroupAssociative a a

performSemigroup :: Semigroup a =>
  SemigroupOperation a -> a -> Bool
performSemigroup op x = case op of
  SemigroupAssociative y z ->
    isAssociative x y z
```

JSON

```
encodeJson :: SemigroupOperation Json -> Json
encodeJson op = case op of
  SemigroupAssociative y z ->
    {"associative": {"y": y, "z": z}}
```

Binary

```
encodeBinary :: SemigroupOperation ByteString -> ByteString
encodeBinary op = case op of
  SemigroupAssociative y z -> y ++ z
```

Monoid

Monoids are a superclass of *Semigroup* and inherit all of their faculties. They have one additional operation defined on it, `mempty`, and it should be the identity element over `append`.

```
class Semigroup a => Monoid a where
  mempty :: a

isLeftIdentity :: Monoid a =>
  a -> Bool
isLeftIdentity x = append mempty x == x

isRightIdentity :: Monoid a =>
  a -> Bool
isRightIdentity x = append x mempty == x
```

Operations

```
data MonoidOperation a
  = MonoidSemigroup (SemigroupOperation a)
  | MonoidLeftIdentity
  | MonoidRightIdentity

performMonoid :: Monoid a =>
  MonoidOperation a -> a -> Bool
performMonoid op x = case op of
  MonoidSemigroup op' ->
    performSemigroup op' x
  MonoidLeftIdentity ->
    isLeftIdentity x
  MonoidRightIdentity ->
    isRightIdentity x
```

JSON

```
encodeJson :: MonoidOperation Json -> Json
encodeJson op = case op of
  MonoidSemigroup op' ->
    {"semigroup": encodeJson op'}
  MonoidLeftIdentity ->
    "leftIdentity"
  MonoidRightIdentity ->
    "rightIdentity"
```

Binary

```
encodeBinary :: MonoidOperation ByteString -> ByteString
encodeBinary op = case op of
  MonoidSemigroup op' ->
    (byteAsByteString 0) ++ encodeBinary op'
  MonoidLeftIdentity ->
    (byteAsByteString 1)
  MonoidRightIdentity ->
    (byteAsByteString 2)
```

Eq

Eq has one operation defined on it, `eq`, and it should be an equivalence relation (reflexive, symmetric, and transitive), and should support negation.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y) -- default instance
```

(continues on next page)

(continued from previous page)

```
not :: Bool -> Bool
not True = False
not False = True

implies :: Bool -> Bool -> Bool
implies True True = True
implies True False = False
implies False True = True
implies False False = True
-- alternative definition
implies p q = if p then q else True

isReflexive :: Eq a =>
  a -> Bool
isReflexive x = x == x

isSymmetric :: Eq a =>
  a -> a -> Bool
isSymmetric x y = (x == y) `implies` (y == x)

isTransitive :: Eq a =>
  a -> a -> a -> Bool
isTransitive x y z = ((x == y) && (y == z)) `implies` (x == z)

hasNegation :: Eq a =>
  a -> a -> Bool
hasNegation x y = (x /= y) `implies` (not (x == y))
```

Operations

```
data EqOperation a
  = EqReflexive
  | EqSymmetry a
  | EqTransitive a a
  | EqNegation a

performEq :: Eq a =>
  EqOperation a -> a -> Bool
performEq op x = case op of
  EqReflexive ->
    isReflexive x
  EqSymmetric y ->
    isSymmetry x y
  EqTransitive y z ->
    isTransitive x y z
  EqNegation y ->
    hasNegation x y
```

JSON

```
encodeJson :: EqOperation Json -> Json
encodeJson op = case op of
```

(continues on next page)

(continued from previous page)

```
  EqReflexive ->
    "reflexive"
  EqSymmetry y ->
    {"symmetry": y}
  EqTransitive y z ->
    {"transitive": {"y": y, "z": z}}
  EqNegation y ->
    {"negation": y}
```

Binary

```
encodeBinary :: EqOperation ByteString -> ByteString
encodeBinary op = case op of
  EqReflexive ->
    (byteAsByteString 0)
  EqSymmetry y ->
    (byteAsByteString 1) ++ y
  EqTransitive y z ->
    (byteAsByteString 2) ++ y ++ z
  EqNegation y ->
    (byteAsByteString 3) ++ y
```

Ord

Ord is a superclass of *Eq* and inherit all of its faculties. It has one additional operation defined on it, `compare`, and it should facilitate a partial order through the `Ordering` type.

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  compare :: a -> a -> Ordering

(<=) :: Ord a => a -> a -> Bool
x <= y = case compare x y of
  LT -> True
  Eq -> True
  GT -> False

isReflexive :: Ord a =>
  a -> Bool
isReflexive x = x <= x

isAntisymmetric :: Ord a =>
  a -> a -> Bool
isAntisymmetric x y = ((x <= y) && (y <= x)) `implies` (x == y)

isTransitive :: Ord a =>
  a -> a -> a -> Bool
isTransitive x y z = ((x <= y) && (y <= z)) `implies` (x <= z)
```

Operations

```
data OrdOperation a
  = OrdEq (EqOperation a)
  | OrdReflexive
  | OrdAntiSymmetry a
  | OrdTransitive a a

performOrd :: Ord a =>
  OrdOperation a -> a -> Bool
performOrd op x = case op of
  OrdEq op' ->
    performEq op' x
  OrdReflexive ->
    isReflexive x
  OrdAntiSymmetry y ->
    isAntiSymmetric x y
  OrdTransitive y z ->
    isTransitive x y z
```

JSON

```
encodeJson :: OrdOperation Json -> Json
encodeJson op = case op of
  OrdEq op' ->
    {"eq": encodeJson op'}
  OrdReflexive ->
    "reflexive"
  OrdAntiSymmetry y ->
    {"antisymmetry": y}
  OrdTransitive y z ->
    {"transitive": {"y": y, "z": z}}
```

Binary

```
encodeBinary :: OrdOperation ByteString -> ByteString
encodeBinary op = case op of
  OrdEq op' ->
    (byteAsByteString 0) ++ encodeBinary op'
  OrdReflexive ->
    (byteAsByteString 1)
  OrdAntiSymmetry y ->
    (byteAsByteString 2) ++ y
  OrdTransitive y z ->
    (byteAsByteString 3) ++ y ++ z
```

Enum

Enum is **not** a superclass of *Ord*, but it uses its faculties in testing. It has two operations defined on it, `pred`, `succ`. `pred` and `succ` should be opposite — isomorphic over composition.

```haskell
class Enum a where
  pred :: a -> a
  succ :: a -> a

predsucc :: Enum a =>
  a -> Bool
predsucc x = (pred (succ x)) == x

succpred :: Enum a =>
  a -> Bool
succpred x = (succ (pred x)) == x
```

Operations

```haskell
data EnumOperation a
  = EnumOrd (OrdOperation a)
  | EnumPredSucc
  | EnumSuccPred

performEnum :: Enum a => Ord a =>
  EnumOperation a -> a -> Bool
performEnum op x = case op of
  EnumOrd op' ->
    perfromOrd op' x
  EnumPredSucc ->
    predsucc x
  EnumSuccPred ->
    succpred x
```

JSON

```haskell
encodeJson :: EnumOperation Json -> Json
encodeJson op = case op of
  EnumOrd op' ->
    {"ord": enocdeJson op'}
  EnumPredSucc ->
    "predsucc"
  EnumSuccPred ->
    "succpred"
```

Binary

```haskell
encodeBinary :: EnumOperation ByteString -> ByteString
encodeBinary op = case op of
  EnumOrd op' ->
    (byteAsByteString 0) ++ encodeBinary op'
  EnumPredSucc ->
    (byteAsByteString 1)
  EnumSuccPred ->
    (byteAsByteString 2)
```

Bounded

Bounded is **not** a superclass of *Ord*, but it uses its faculties in testing. It has two values defined on it, `top` and `bottom`. Types that are Bounded are bounded lattices.

```
class Bounded a where
  top :: a
  bottom :: a

isBetween :: Bounded a => Ord a =>
  a -> Bool
isBetween x = (x <= top) && (bottom <= x)
```

Operations

```
data BoundedOperation a
  = BoundedOrd (OrdOperation a)
  | BoundedBetween

performBounded :: Bounded a => Ord a =>
  BoundedOperation a -> a -> Bool
performBounded op x = case op of
  BoundedOrd op' ->
    performOrd op' x
  BoundedBetween ->
    isBetween x
```

JSON

```
encodeJson :: BoundedOperation Json -> Json
encodeJson op = case op of
  BoundedOrd op' ->
    {"ord": encodeJson op'}
  BoundedBetween ->
    "between"
```

Binary

```
encodeBinary :: BoundedOperation ByteString -> ByteString
encodeBinary op = case op of
  BoundedOrd op' ->
    (byteToByteString 0) ++ encodeBinary op'
  BoundedBetween ->
    (byteToByteString 1)
```

BoundedEnum

BoundedEnum is a superclass of both *Enum* and *Bounded*, and inherit all of their faculties. It has two additional operations defined on it, `toEnum` and `fromEnum`. They should be opposite functions to each other, and furthermore,

`fromEnum` should be homomorphic over `compare`. BoundedEnums are total orders.

```
class (Bounded a, Enum a) => BoundedEnum a where
  toEnum :: Int -> Maybe a
  fromEnum :: a -> Int

compareHom :: BoundedEnum a => Ord a =>
  a -> a -> Bool
compareHom x y = (compare x y) == (compare (fromEnum x) (fromEnum y))

fromPredMapping :: BoundedEnum a =>
  a -> Bool
fromPredMapping x = (fromEnum (pred x)) == (pred (fromEnum x))

fromSuccMapping :: BoundedEnum a =>
  a -> Bool
fromSuccMapping x = (fromEnum (succ x)) == (succ (fromEnum x))

toFromIsomorphism :: Enum a =>
  a -> Bool
toFromIsomorphism x = (toEnum (fromEnum x)) == (Just x)
```

Operations

```
data BoundedEnumOperation a
  = BoundedEnumEnum (EnumOperation a)
  | BoundedEnumBounded (BoundedOperation a)
  | BoundedEnumCompareHom a
  | BoundedEnumFromPred
  | BoundedEnumFromSucc
  | BoundedEnumToFromIso

performBoundedEnum :: BoundedEnum a => Ord a =>
  BoundedEnumOperation a -> a -> Bool
performBoundedEnum op x = case op of
  BoundedEnumEnum op' ->
    performEnum op' x
  BoundedEnumBounded op' ->
    performBounded op' x
  BoundedEnumCompareHom y ->
    compareHom x y
  BoundedEnumFromPred ->
    fromPredMapping x
  BoundedEnumFromSucc ->
    fromSuccMapping x
  BoundedEnumToFromIso ->
    toFromIsomorphism x
```

JSON

```
encodeJson :: BoundedEnumOperation Json -> Json
encodeJson op = case op of
  BoundedEnumEnum op' ->
    {"enum": encodeJson op'}
```

(continues on next page)

(continued from previous page)

```
  BoundedEnumBounded op' ->
    {"bounded": encodeJson op'}
  BoundedEnumCompareHom y ->
    {"compareHom": y}
  BoundedEnumFromPred ->
    "fromPred"
  BoundedEnumFromSucc ->
    "fromSucc"
  BoundedEnumToFromIso ->
    "toFromIso"
```

Binary

```
encodeBinary :: BoundedEnumOperation ByteString -> ByteString
encodeBinary op = case op of
  BoundedEnumEnum op' ->
    (byteAsByteString 0) ++ encodeBinary op'
  BoundedEnumBounded op' ->
    (byteAsByteString 1) ++ encodeBinary op'
  BoundedEnumCompareHom y ->
    (byteAsByteString 2) ++ y
  BoundedEnumFromPred ->
    (byteAsByteString 3)
  BoundedEnumFromSucc ->
    (byteAsByteString 4)
  BoundedEnumToFromIso ->
    (byteAsByteString 5)
```

HeytingAlgebra

HeytingAlgebra has six operations defined on it, `ff`, `tt`, `implies`, `conj`, `disj`, and `not`. It should form a heyting algebra.

```
class HeytingAlgebra a where
  ff :: a
  tt :: a
  implies :: a -> a -> a
  conj :: a -> a -> a
  disj :: a -> a -> a
  not :: a -> a

isDisjAssociative :: HeytingAlgebra a =>
  a -> a -> a -> Bool
isDisjAssociative x y z = (disj x (disj y z)) == (disj (disj x y) z)

isConjAssociative :: HeytingAlgebra a =>
  a -> a -> a -> Bool
isConjAssociative x y z = (conj x (conj y z)) == (conj (conj x y) z)

isDisjCommutative :: HeytingAlgebra a =>
  a -> a -> Bool
```

(continues on next page)

(continued from previous page)

```
isDisjCommutative x y = (disj x y) == (disj y x)

isConjCommutative :: HeytingAlgebra a =>
  a -> a -> Bool
isConjCommutative x y = (conj x y) == (conj y x)

disjConjAbsorption :: HeytingAlgebra a =>
  a -> a -> Bool
disjConjAbsorption x y = (disj x (conj x y)) == x

conjDisjAbsorption :: HeytingAlgebra a =>
  a -> a -> Bool
conjDisjAbsorption x y = (conj x (disj x y)) == x

isDisjIdempotent :: HeytingAlgebra a =>
  a -> Bool
isDisjIdempotent x = (disj x x) == x

isConjIdempotent :: HeytingAlgebra a =>
  a -> Bool
isConjIdempotent x = (conj x x) == x

disjIdentity :: HeytingAlgebra a =>
  a -> Bool
disjIdentity x = (disj x ff) == x

conjIdentity :: HeytingAlgebra a =>
  a -> Bool
conjIdentity x = (conj x tt) == x

implicationTop :: HeytingAlgebra a =>
  a -> Bool
implicationTop x = (implies x x) == tt

implicationApplication :: HeytingAlgebra a =>
  a -> a -> Bool
implicationApplication x y = (conj x (implies x y)) == (conj x y)

implicationConclusion :: HeytingAlgebra a =>
  a -> a -> Bool
implicaitonConclusion x y = (conj y (implies x y)) == y

implicationDistributive :: HeytingAlgebra a =>
  a -> a -> a -> Bool
implicationDistributive x y z = (implies x (conj y z)) == (conj (implies x y)␣
↪(implies x z))

hasCompliment :: HeytingAlgebra a =>
  a -> Bool
hasCompliment x = (not a) == (implies x ff)
```

Operations

```
data HeytingAlgebraOperation a
```

(continues on next page)

(continued from previous page)

```
  = DisjAssociative a a
  | ConjAssociative a a
  | DisjCommutative a
  | ConjCommutative a
  | DisjConjAbsorption a
  | ConjDisjAbsorption a
  | DisjIdempotent
  | ConjIdempotent
  | DisjIdentity
  | ConjIdentity
  | ImplicationTop
  | ImplicationApplication a
  | ImplicationConclusion a
  | ImplicationDistribution a a
  | Compliment

performHeytingAlgebra :: HeytingAlgebra a =>
  HeytingAlgebraOperation a -> a -> Bool
performHeytingAlgebra op x = case op of
  DisjAssociative y z ->
    isDisjAssociatve x y z
  ConjAssociative y z ->
    isConjAssociatve x y z
  DisjCommutative y ->
    isDisjCommutative x y
  ConjCommutative y ->
    isConjCommutative x y
  DisjConjAbsorption y ->
    disjConjAbsorption x y
  ConjDisjAbsorption y ->
    conjDisjAbsorption x y
  DisjIdempotent ->
    isDisjIdempotent x
  ConjIdempotent ->
    isConjIdempotent x
  DisjIdentity ->
    disjIdentity x
  ConjIdentity ->
    conjIdentity x
  ImplicationTop ->
    implicationTop x
  ImplicationApplication y ->
    implicationApplication x y
  ImplicationConclusion y ->
    implicationConclusion x y
  ImplicationDistributive y z ->
    implicationDistributive x y z
  Compliment ->
    hasCompliment x
```

JSON

```
encodeJson :: HeytingAlgebraOperation Json -> Json
encodeJson op = case op of
```

(continues on next page)

(continued from previous page)

```
  DisjAssociative y z ->
    {"disjAssociative": {"y": y, "z": z}}
  ConjAssociative y z ->
    {"conjAssociative": {"y": y, "z": z}}
  DisjCommutative y ->
    {"disjCommutative": y}
  ConjCommutative y ->
    {"conjCommutative": y}
  DisjConjAbsorption y ->
    {"disjConjAbsorption": y}
  ConjDisjAbsorption y ->
    {"conjDisjAbsorption": y}
  DisjIdempotent ->
    "disjIdempotent"
  ConjIdempotent ->
    "conjIdempotent"
  DisjIdentity ->
    "disjIdentity"
  ConjIdentity ->
    "conjIdentity"
  ImplicationTop ->
    "implicationTop"
  ImplicationApplication y ->
    {"implicationApplication": y}
  ImplicationConclusion y ->
    {"implicationConclusion": y}
  ImplicationDistributive y z ->
    {"implicationDistributive": {"y": y, "z": z}}
  Compliment ->
    "compliment"
```

Binary

```
encodeBinary :: HeytingAlgebraOperation ByteString -> ByteString
encodeBinary op = case op of
  DisjAssociative y z ->
    (byteToByteString 0) ++ y ++ z
  ConjAssociative y z ->
    (byteToByteString 1) ++ y ++ z
  DisjCommutative y ->
    (byteToByteString 2) ++ y
  ConjCommutative y ->
    (byteToByteString 3) ++ y
  DisjConjAbsorption y ->
    (byteToByteString 4) ++ y
  ConjDisjAbsorption y ->
    (byteToByteString 5) ++ y
  DisjIdempotent ->
    (byteToByteString 6)
  ConjIdempotent ->
    (byteToByteString 7)
  DisjIdentity ->
    (byteToByteString 8)
  ConjIdentity ->
```

(continues on next page)

(continued from previous page)

```
    (byteToByteString 9)
  ImplicationTop ->
    (byteToByteString 10)
  ImplicationApplication y ->
    (byteToByteString 11) ++ y
  ImplicationConclusion y ->
    (byteToByteString 12) ++ y
  ImplicationDistributive y z ->
    (byteToByteString 13) ++ y ++ z
  Compliment ->
    (byteToByteString 14)
```

BooleanAlgebra

BooleanAlgebra is a superclass of *HeytingAlgebra* and inherit all of its faculties. It has no additional operations defined on it, but it should facilitate a boolean algebra, by supporting the law of the excluded middle.

```
class HeytingAlgebra a => BooleanAlgebra a

hasLawOfExcludedMiddle :: BooleanAlgebra a =>
  a -> Bool
hasLawOfExcludedMiddle x = (disj x (not x)) == tt
```

Operations

```
data BooleanAlgebraOperation a
  = BooleanAlgebraHeytingAlgebra (HeytingAlgebraOperation a)
  | LawOfExcludedMiddle

performBooleanAlgebra :: BooleanAlgebra a =>
  BooleanAlgebraOperation a -> a -> Bool
performBooleanAlgebra op x = case op of
  BooleanAlgebraHeytingAlgebra op' ->
    performHeytingAlgebra op' x
  LawOfExcludedMiddle ->
    hasLawOfExcludedMiddle x
```

JSON

```
encodeJson :: BooleanAlgebraOperation Json -> Json
encodeJson op = case op of
  BooleanAlgebraHeytingAlgebra op' ->
    {"heytingAlgebra": encodeJson op'}
  LawOfExcludedMiddle ->
    "lawOfExcludedMiddle"
```

Binary

```
encodeBinary :: BooleanAlgebraOperation ByteString -> ByteString
encodeBinary op = case op of
  BooleanAlgebraHeytingAlgebra op' ->
    (byteToByteString 0) ++ encodeBinary op'
  LawOfExcludedMiddle ->
    (byteToByteString 1)
```

Semiring

Semiring has four operations defined on it, `zero`, `one`, `add`, and `mul`. It should form a semiring.

```
class Semiring a where
  zero :: a
  one :: a
  add :: a -> a -> a
  mul :: a -> a -> a

associative :: (a -> a -> a) -> a -> a -> a -> Bool
associative f x y z = (f x (f y z)) == (f (f x y) z)

commutative :: (a -> a -> a) -> a -> a -> Bool
commutative f x y = (f x y) == (f y x)

distributive :: (a -> a) -> (a -> a -> a) -> a -> a -> Bool
distributive f g x y = (f (g x y)) == (g (f x) (f y))

isCommutativeMonoid :: Semiring a =>
  a -> a -> a -> Bool
isCommutativeMonoid x y z =
  (associative add x y z)
    && (commutative add x y)
    -- zero is the empty element for add
    && ((add x zero) == x)

isMonoid :: Semiring a =>
  a -> a -> a -> Bool
isMonoid x y z =
  (associative mul x y z)
    -- one is the empty element for mul
    && ((mul x one) == x)

isLeftDistributive :: Semiring a =>
  a -> a -> a -> Bool
isLeftDistributive x y z =
  distributive (\q -> mul x q) add y z

isRightDistributive :: Semiring a =>
  a -> a -> a -> Bool
isRightDistributive x y z =
  distributive (\q -> mul q x) add y z

hasAnnihilation :: Semiring a =>
```

(continues on next page)

(continued from previous page)

```
  a -> Bool
hasAnnihilation x =
  ((mul x 0) == (mul 0 x))
    && ((mul x 0) == 0)
```

Operations

```
data SemiringOperation a
  = SemiringCommutativeMonoid a a
  | SemiringMonoid a a
  | SemiringLeftDistributive a a
  | SemiringRightDistributive a a
  | SemiringAnnihilation

performSemiring :: Semiring a =>
  SemiringOperation a -> a -> a -> Bool
performSemiring op x = case op of
  SemiringCommutativeMonoid y x ->
    isCommutativeMonoid x y z
  SemiringMonoid y z ->
    isMonoid x y z
  SemiringLeftDistributive y z ->
    isLeftDistributive x y z
  SemiringRightDistributive y z ->
    isRightDistributive x y z
  SemiringAnnihilation ->
    hasAnnihilation x
```

JSON

```
encodeJson :: SemiringOperation Json -> Json
encodeJson op = case op of
  SemiringCommutativeMonoid y z ->
    {"commutativeMonoid": {"y": y, "z": z}}
  SemiringMonoid y z ->
    {"monoid": {"y": y, "z": z}}
  SemiringLeftDistributive y z ->
    {"leftDistributive": {"y": y, "z": z}}
  SemiringRightDistributive y z ->
    {"rightDistributive": {"y": y, "z": z}}
  SemiringAnnihilation ->
    "annihilation"
```

Binary

```
encodeBinary :: SemiringOperation ByteString -> ByteString
encodeBinary op = case op of
  SemiringCommutativeMonoid y z ->
    (byteToByteString 0) ++ y ++ z
  SemiringMonoid y z ->
```

(continues on next page)

(continued from previous page)

```
    (byteToByteString 1) ++ y ++ z
  SemiringLeftDistributive y z ->
    (byteToByteString 2) ++ y ++ z
  SemiringRightDistributive y z ->
    (byteToByteString 3) ++ y ++ z
  SemiringAnnihilation ->
    (byteToByteString 4)
```

Ring

Ring is a superclass of Semiring and inherit all of its faculties. It has one additional operation defined on it, `sub`, and it should facilitate an additive inverse.

```
class Semiring a => Ring a where
  sub :: a -> a -> a

isAdditiveInverse :: Ring a =>
  a -> Bool
isAdditiveInverse x = (sub x x) == zero
```

Operations

```
data RingOperation a
  = RingSemiring (SemiringOperation a)
  | RingAdditiveInverse

performRing :: Ring a =>
  RingOperation a -> a -> Bool
performRing op x = case op of
  RingSemiring op' ->
    performSemiring op' x
  RingAdditiveInverse ->
    isAdditiveInverse x
```

JSON

```
encodeJson :: RingOperation Json -> Json
encodeJson op = case op of
  RingSemiring op' ->
    {"semiring": encodeJson op'}
  RingAdditiveInverse ->
    "additiveInverse"
```

Binary

```
encodeBinary :: RingOperation ByteString -> ByteString
encodeBinary op = case op of
  RingSemiring op' ->
    (byteAsByteString 0) ++ encodeBinary op'
  RingAdditiveInverse ->
    (byteAsByteString 1)
```

DivisionRing

DivisionRing is a superclass of *Ring* and inherit all of its faculties. It has one additional operation defined on it, `recip`, and it should facilitate a multiplicative inverse.

```
class Ring a => DivisionRing a where
  recip :: a -> a

isInverse :: DivisionRing a =>
  a -> Bool
isInverse x = (x /= zero) `implies` ((mul x (recip x)) == one)
```

Operations

```
data DivisionRingOperation a
  = DivisionRingRing (RingOperation a)
  | DivisionRingInverse

performDivisionRing :: DivisionRing a =>
  DivisionRingOperation a -> a -> Bool
performDivisionRing op x = case op of
  DivisionRingRing op' ->
    performRing op' x
  DivisionRingInverse ->
    isInverse x
```

JSON

```
encodeJson :: DivisionRingOperation Json -> Json
encodeJson op = case op of
  DivisionRingRing op' ->
    {"ring": encodeJson op'}
  DivisionRingInverse ->
    "inverse"
```

Binary

```
encodeBinary :: DivisionRingOperation ByteString -> ByteString
encodeBinary op = case op of
  DivisionRingRing op' ->
```

(continues on next page)

(continued from previous page)

```
    (byteToByteString 0) ++ encodeBinary op'
  DivisionRingInverse ->
    (byteToByteString 1)
```

CommutativeRing

CommutativeRing is a superclass of *Ring* and inherit all of its faculties. It has no additional operation defined on it, but assumes `mul` is commutative.

```
class Ring a => CommutativeRing a

isCommutative :: CommutativeRing a =>
  a -> a -> Bool
isCommutative x y = (mul x y) == (mul y x)
```

Operations

```
data CommutativeRingOperation a
  = CommutativeRingRing (RingOperation a)
  | CommutativeRingCommutative a

performCommutativeRing :: CommutativeRing a =>
  CommutativeRingOperation a -> a -> Bool
performCommutativeRing op x = case op of
  CommutativeRingRing op' ->
    performRing op' x
  CommutativeRingCommutative y ->
    isCommutative x y
```

JSON

```
encodeJson :: CommutativeRingOperation Json -> Json
encodeJson op = case op of
  CommutativeRingRing op' ->
    {"ring": encodeJson op'}
  CommutativeRingCommutative y ->
    {"commutative": y}
```

Binary

```
encodeBinary :: CommutativeRingOperation ByteString -> ByteString
encodeBinary op = case op of
  CommutativeRingRing op' ->
    (byteToByteString 0) ++ encodeBinary op'
  CommutativeRingCommutative y ->
    (byteToByteString 1) ++ y
```

EuclideanRing

EuclideanRing is a superclass of *CommutativeRing* and inherit all of its faculties. It has three additional operations defined on it, `mod`, `div`, and `degree`. It should facilitate a Euclidean domain, however, we can only test for the integral domain (due to language compatibility).

```
class CommutativeRing a => EuclideanRing a where
  degree :: a -> Int
  mod :: a -> a -> a
  div :: a -> a -> a

isIntegralDomain :: EuclideanRing a =>
  a -> a -> Bool
isIntegralDomain x y = ((x /= zero) && (y /= zero)) `implies` ((mul x y) /= zero)
```

Operations

```
data EuclideanRingOperation a
  = EuclideanRingCommutativeRing (CommutativeRingOperation a)
  | EuclideanRingIntegralDomain a

performEuclideanRing :: EuclideanRing a =>
  EuclideanRingOperation a -> a -> Bool
performEuclideanRing op x = case op of
  EuclideanRingCommutativeRing op' ->
    performCommutativeRing op' x
  EuclideanRingIntegralDomain y ->
    isIntegralDomain x y
```

JSON

```
encodeJson :: EuclideanRingOperation Json -> Json
encodeJson op = case op of
  EuclideanRingCommutativeRing op' ->
    {"commutativeRing": encodeJson op'}
  EuclideanRingIntegralDomain y ->
    {"integralDomain": y}
```

Binary

```
encodeBinary :: EuclideanRingOperation ByteString -> ByteString
encodeBinary op = case op of
  EuclideanRingCommutativeRing op' ->
    (byteToByteString 0) ++ encodeBinary op'
  EuclideanRingIntegralDomain y ->
    (byteToByteString 1) ++ y
```

Field

Field is a superclass of both *DivisionRing* and *EuclideanRing*, and inherit all of their faculties. It has no additional operation defined on it, but is a field.

```
class (DivisionRing a, EuclideanRing a) => Field a
```

Operations

```
data FieldOperation a
  = FieldDivisionRing (DivisionRingOperation a)
  | FieldEuclideanRing (EuclideanRingOperation a)

performField :: Field a =>
  FieldOperation a -> a -> a -> Bool
performField op x = case op of
  FieldDivisionRing op' ->
    performDivisionRing op' x
  FieldEuclideanRing op' ->
    performEuclideanRing op' x
```

JSON

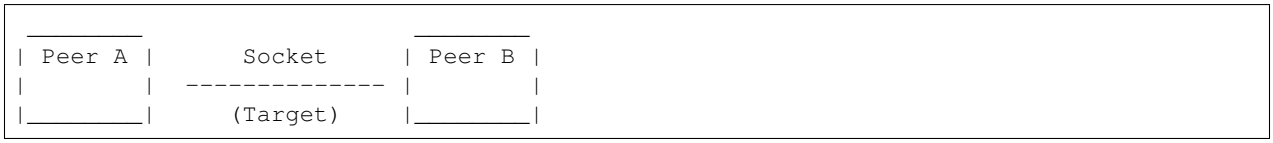
```
encodeJson :: FieldOperation Json -> Json
encodeJson op = case op of
  FieldDivisionRing op' ->
    {"divisionRing": encodeJson op'}
  FieldEuclideanRing op' ->
    {"euclideanRing": encodeJson op'}
```

Binary

```
encodeBinary :: FieldOperation ByteString -> ByteString
encodeBinary op = case op of
  FieldDivisionRing op' ->
    (byteToByteString 0) ++ encodeBinary op'
  FieldEuclideanRing op' ->
    (byteToByteString 1) ++ encodeBinary op'
```

5.2.3 Symbiote Test Suite

The test suite is a simple idea — there are two peers `A` and `B` (each on potentially different platforms), with some peer-to-peer communications channel between them, which we'll call `Socket` (for instance, ZeroMQ or a WebSocket). `Socket` only understands some target data type, which we'll also call `Target` (for instance, `ByteString` or `ArrayBuffer` for ZeroMQ, and `Json` for WebSockets).



Now, both `Peer A` and `Peer B` understand some "idea" of a data type, which we'll call `Type T`. The whole purpose of this test suite is to verify that both peers have an *identical* understanding of `Type T`, with respect to how *T operates*, **and** how it *translates* to-and-from `Target`.

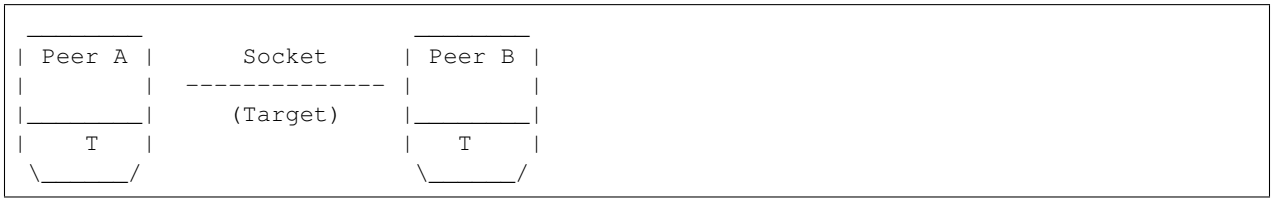
So, each peer has an idea of what `T` is, and some static set of operations that can be performed on `T`. The "operations" on `T` need to be extrapolated into a data type as well, such that every return type when performed on a `T` is homogeneous. For instance, if I have functions `f :: T -> T` and `g :: T -> Bool`, then I need to come up with a data type that that can combine them. For instance:

```
data OperationsOnT
  = F
  | G

perform :: OperationsOnT -> T -> Either T Bool
perform op x = case op of
  F -> Left (f x)
  G -> Right (g x)
```

`Either T Bool` is the "homogeneous return type" of *all* the supported operations on my type `T`.

As we proceed, our idea of the network might look like this:



The entire purpose of this protocol is to test that remote procedure calls are identical to local calls, when the data and operation are the same — given that the serialization method is identical, as well.

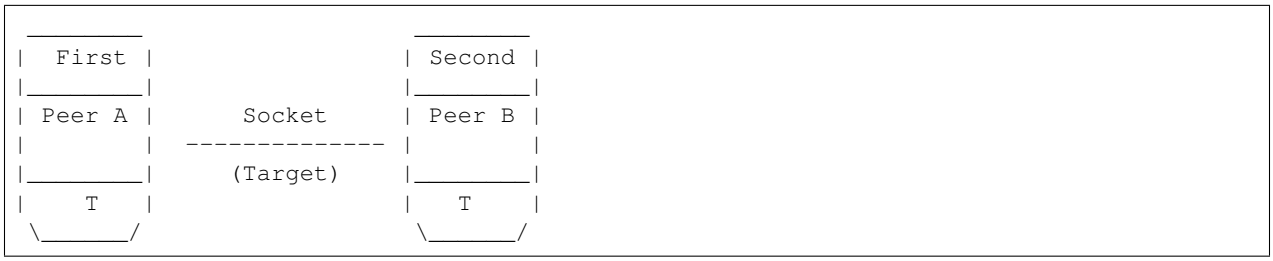
First and Second

Now for the actual protocol.

The protocol assumes its being run on a 1-to-1 communication mechanism — there's no support for broadcasted and subscribed messages; only *one* other peer will be tested while the other is running. This implies a constraint on how our protocol will operate — one peer needs to be the "first" to operate, and the other needs to be the "second".

It's not rocket science — if we're generating data and expecting the other party to operate on that data, then someone has to go first while the other listens for it. If they both went at the same time, there would be a race condition. If they both were expecting data, then there'd be a deadlock. Therefore, there should be some decision made by the programmer (you) on which peer will be the "first" and which will be "second". It doesn't make a difference in the semantic integrity of the test suite — they both will generate and consume the same amount of random data.

For simplicity's sake, let's make `Peer A` the first, and `Peer B` the second.



Generation

Now, both peers will take turns in generating, consuming, and exchanging data. `First` will be (you guessed it) the first one to take a step in that dance. Before they start, though, let's talk about data generation for a quick second.

If you've ever heard of QuickCheck or property-based testing, what is about to be said should come to no surprise for you. For everyone else, though, it might seem a bit alien.

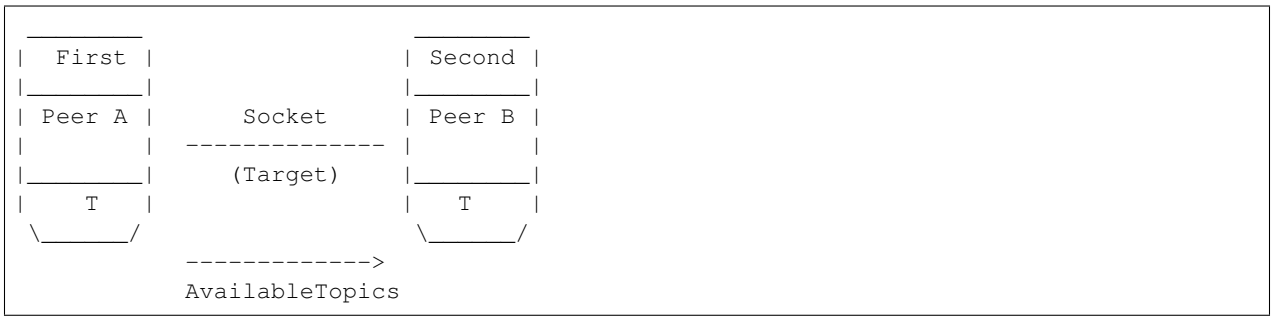
The act of generating random data implies that there's some **"size"** associated with how large that data should be.

Think about it; if we just didn't care about the general size of that random data, and we were generating an array, the computer could just generate an infinite set of data just as likely as an empty array. What's going to tell it to stop? A `size` parameter, that's what.

So, in our test suite, every data type we'll be verifying (through generation) **must** have some maximum size (as an integer), which we'll approach starting from `0`.

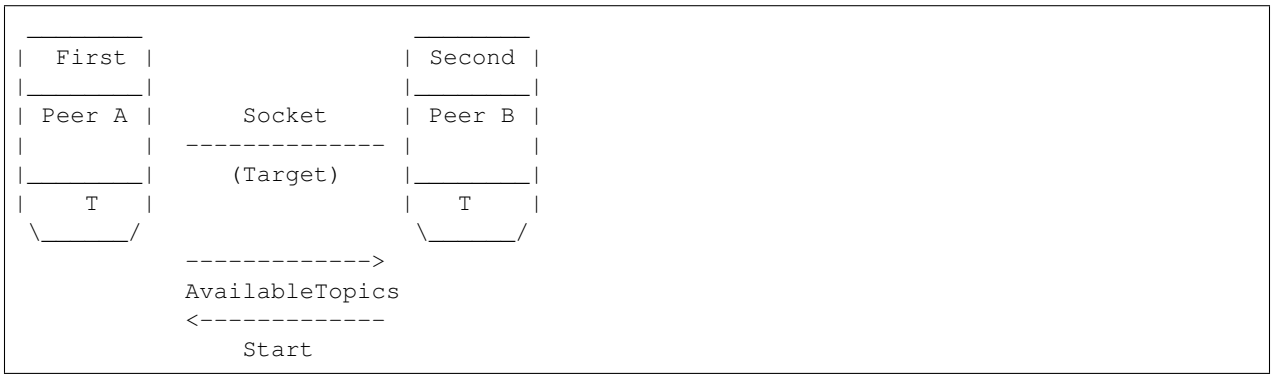
The first thing that our peers do is agree upon a set of types to test on, and their respective sizes - each peer will take a turn generating the data, incrementing their own counters until they each hit the maximum (around the same time).

The type's name is typically used in the message, but for our intents and purposes, we'll call it a "topic", because it's just a string. If the programmer chooses to use a different topic than the type's name, then there's no issue, so long as both peers expect the same topic/size pairings. We'll call this message `AvailableTopics`, and it's sent by `First`.



Just to be clear, `AvailableTopics` is a mapping of *topics* (usually type names) to *sizes*.

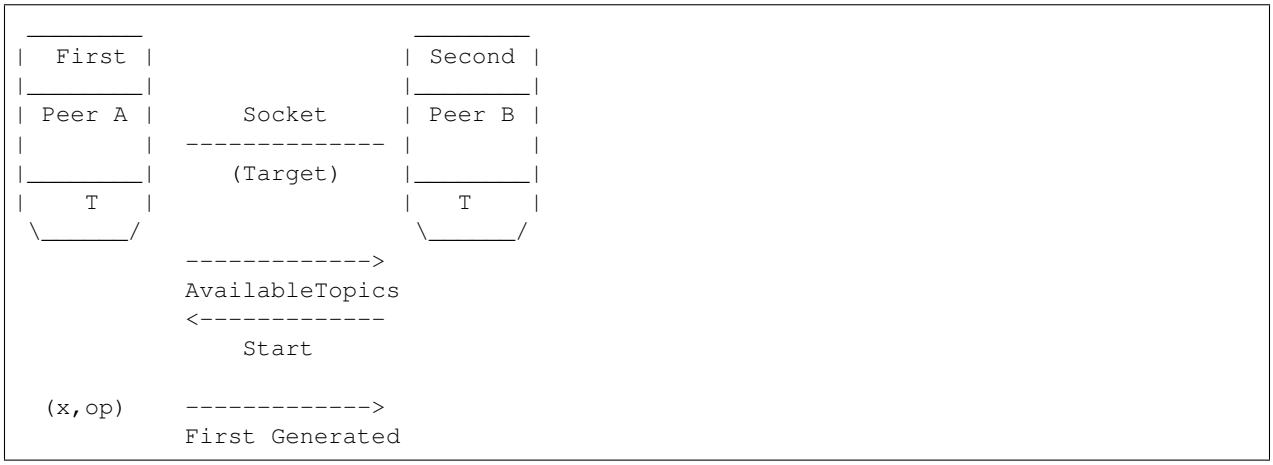
If `Peer B` disagrees with the topics or sizes, it'll throw an error, and reflect that error back to `Peer A` so it can explode loudly as well. If all is well, then `Peer B` will tell `Peer A` to start, with the subset of `Peer A`'s topics actually in use for this session:



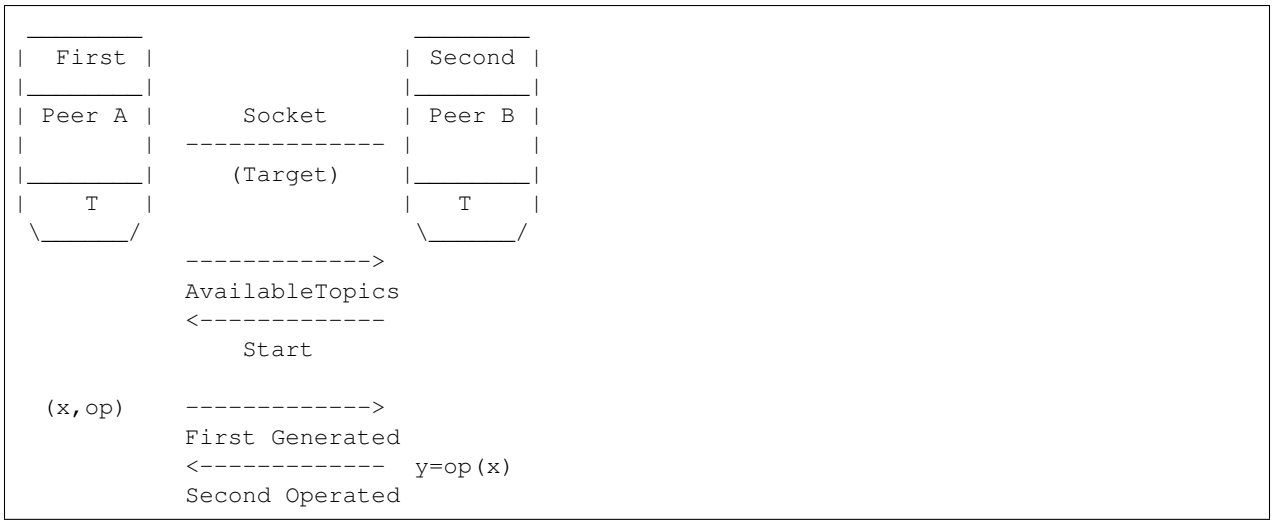
First Generating, Second Operating

Now everything is ready to go! Let's get started. First thing is first, `First` will generate a random value of type `T` (which we'll call `x :: T`), and a random *operation* on type `T` as `OperationsOnT` (which we'll call `op :: OperationsOnT`), with a size-index of `0`.

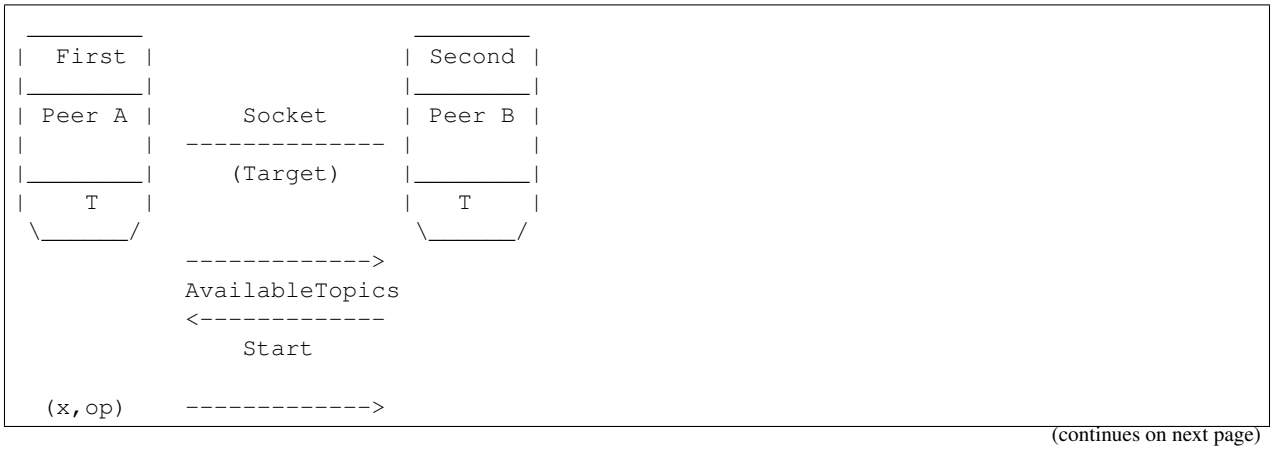
Here, it will pack them both together in a message called `Generated` and send it over the wire to `Second` (serialized in the type `Target`, remember).



`Second` will then decode the message and see a value and an operation. All it has to do from here is perform the operation `op` on `x`, and get a result (which we'll call `y`). It sends this result back to `First` as a message called `Second Operated`.



Next, `First` has to verify that `y` is indeed `op(x)`. If it's not, explode loudly, and tell `Second` to do the same. If it's good, then tell `Second` it's their turn, with a message called `YourTurn`.



(continues on next page)

(continued from previous page)

```
            First Generated
            <-------------  y=op(x)
 y=op(x)?   Second Operated
   true     ------------->
               YourTurn
```

First Operating, Second Generating

The tables have turned! `First` is now operating, and `Second` gets to generate. The entire process is exactly the same, just reversed:

```
 ________                    ________
|  First |                  | Second |
|________|                  |________|
| Peer A |      Socket      | Peer B |
|        |  --------------  |        |
|________|     (Target)     |________|
|    T   |                  |   T    |
 \______/                    \______/
            ------------->
            AvailableTopics
            <-------------
                Start

  (x,op)    ------------->
            First Generated
            <-------------  y=op(x)
 y=op(x)?   Second Operated
   true     ------------->
               YourTurn

            <-------------  (x,op)
           Second Generated
  y=op(x)   ------------->
            First Operated y=op(x)?
            <-------------   true
               YourTurn
```

Perfect. Each peer has now both generated and operated once, and has increased their size counter by *1*. They will keep doing this until one of them reaches the max generation size. Can you guess which one will be... first... in doing so?

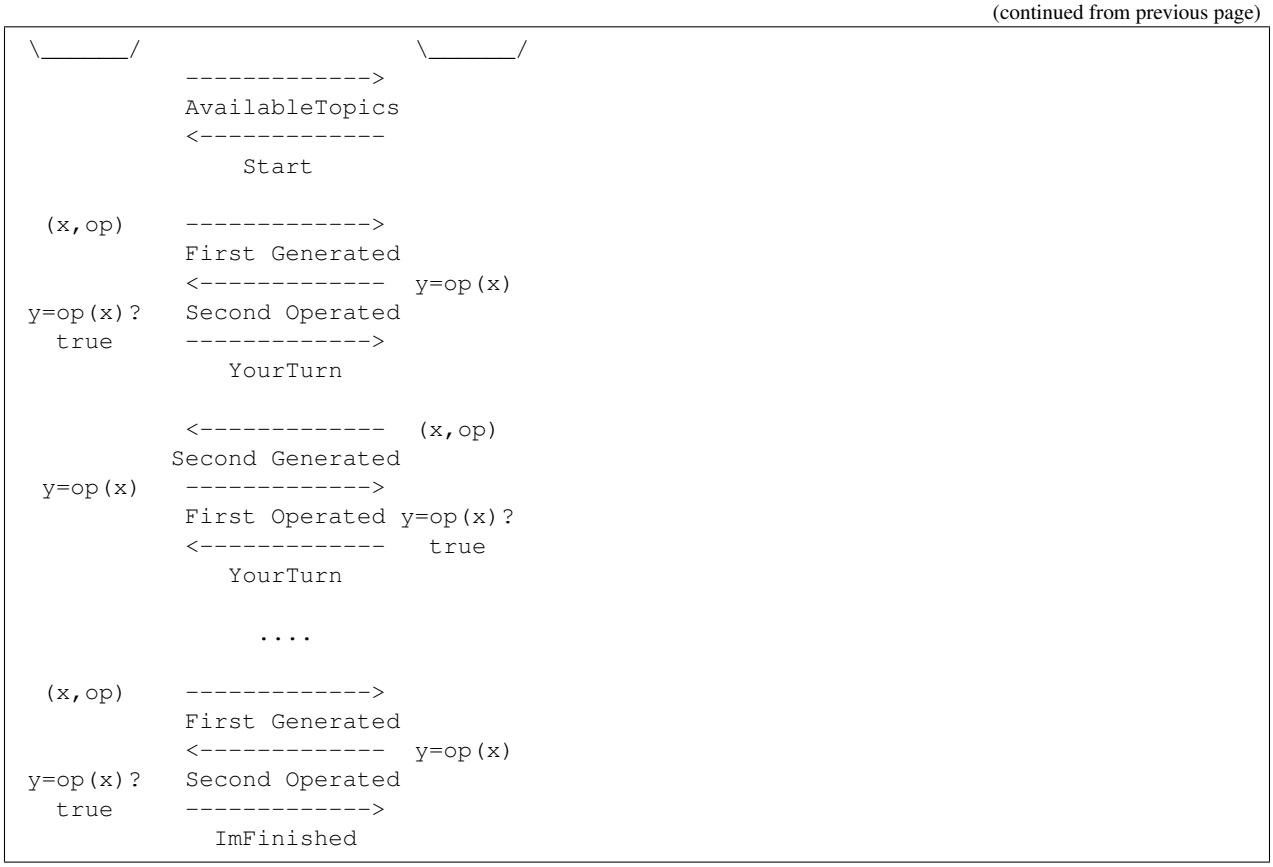
I'm Finished

I guess I gave it away — `First` generates first, so it must also be the first to be finished generating. It signals this by sending `ImFinished` instead of `YourTurn`.

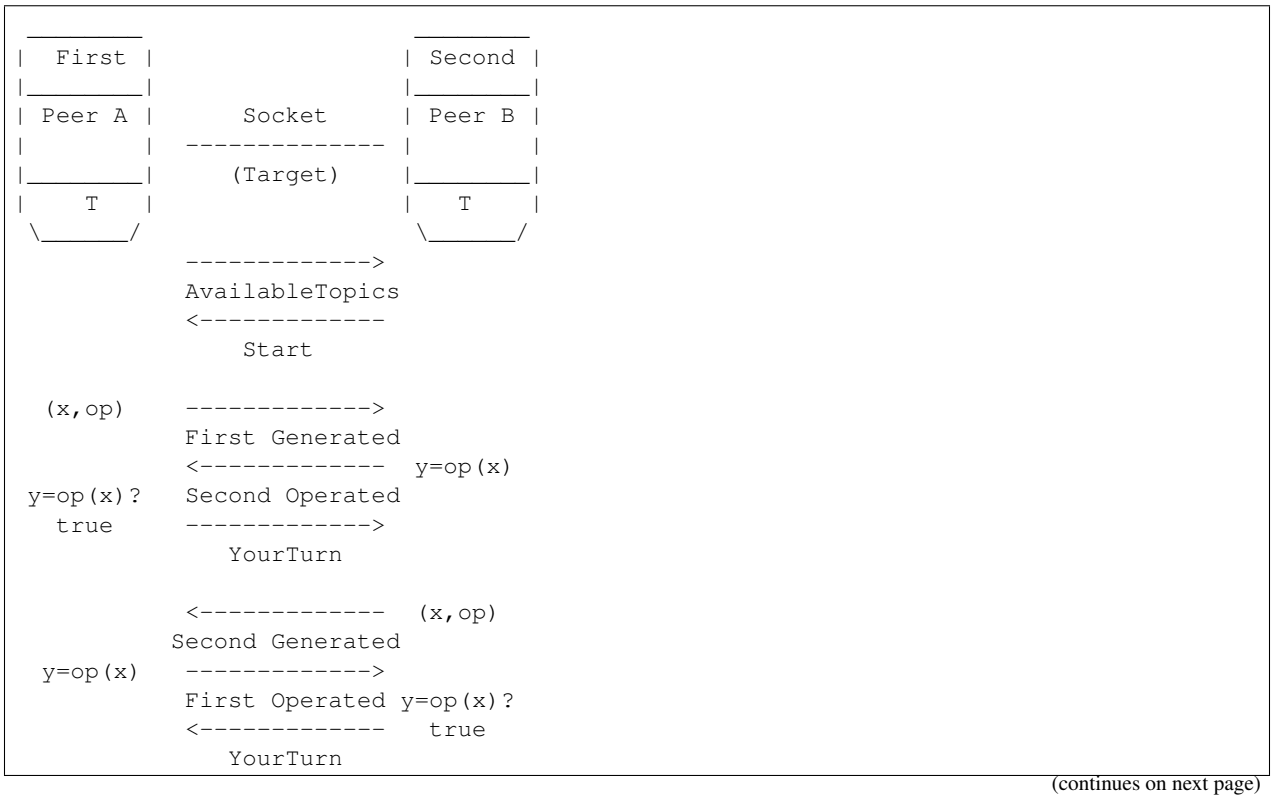
```
 ________                    ________
|  First |                  | Second |
|________|                  |________|
| Peer A |      Socket      | Peer B |
|        |  --------------  |        |
|________|     (Target)     |________|
|    T   |                  |   T    |
```

(continues on next page)

(continued from previous page)



After `Second` receives that `ImFinished` message, it will send `ImFinished` also once it has verified its last run.



(continues on next page)

(continued from previous page)

```
                ....

  (x,op)     ------------->
             First Generated
             <-------------  y=op(x)
y=op(x)?     Second Operated
  true       ------------->
               ImFinished

             <-------------  (x,op)
            Second Generated
 y=op(x)     ------------->
             First Operated y=op(x)?
             <-------------   true
               ImFinished
```

The topic's entire routine has been completed and verified. By using random generation, we can verify the *properties* of our data, instead of cherry-picked unit tests. At the same time, it will verify both implementations for encoding and decoding our data are also correct. It is a powerful tool that can help us catch edge cases we didn't consider before hand, or reveal fundamental misunderstandings about how our platform works under-the-hood.

Because our scenario only uses one topic `T`, both peers will know that they are finished, and exit without failure.

```
 ________                    ________
|  First |                  | Second |
|________|                  |________|
| Peer A |      Socket      | Peer B |
|        |  --------------  |        |
|________|     (Target)     |________|
|    T   |                  |   T    |
 \______/                    \______/
             ------------->
             AvailableTopics
             <-------------
                 Start

  (x,op)     ------------->
             First Generated
             <-------------  y=op(x)
y=op(x)?     Second Operated
  true       ------------->
                YourTurn

             <-------------  (x,op)
            Second Generated
 y=op(x)     ------------->
             First Operated y=op(x)?
             <-------------   true
                YourTurn

                  ....

  (x,op)     ------------->
             First Generated
             <-------------  y=op(x)
y=op(x)?     Second Operated
```

(continues on next page)

(continued from previous page)

```
   true      ------------->
               ImFinished

             <-------------   (x,op)
            Second Generated
  y=op(x)    ------------->
             First Operated y=op(x)?
             <-------------   true
               ImFinished

 _________                   _________
|  Done   |                 |  Done   |
|_________|                 |_________|
```

5.2.4 Symbiote Test Suite Definitions

Topic

A `Topic` is just an alias for a String with a maximum length of `2^32-1`, whatever that is for your platform. When serialized, it will be UTF-8 encoded.

```
data Topic = Topic String
```

Generate

Generating Unicode strings can be tricky business. It is common for operating systems to treat surrogate characters as special, and should therefore be avoided. See this Wikipedia article on UTF-16 (which is what JavaScript uses under-the-hood) for more details.

```
generate :: Gen Topic
generate = do
  characters <- arrayOf (arbitraryChar `suchThat` isNotSurrogate)
  return (Topic (arrayToString characters))
```

JSON

`Topic` values are encoded as a plain JSON string.

```
encodeJson :: Topic -> Json
encodeJson (Topic t) = stringAsJson t
```

Binary

This protocol is required to support JavaScript, which does not support 64-bit integers natively. Therefore, the encoding mechanism used here will do a standard UTF-8 encoding with a 32-bit size index of the bytestring.

```
encodeBinary :: Topic -> ByteString
encodeBinary (Topic t) =
```

(continues on next page)

(continued from previous page)

```
  (intAsByteStringBE (length t :: Int32))
    ++ (stringAsByteString t)
```

Size

JavaScript does not support 64-bit integers, so it is our lowest-common-denominator for this test suite, and we should use 32-bit integers whenever we need an `Int`.

```
data Size = Size Int32
```

Generate

```
generate :: Gen Size
generate = do
  size <- between 0 (2^31 - 1)
  return (Size size)
```

JSON

JavaScript can handle 32-bit integers, so we can directly use our `Size` in a JSON document.

```
encodeJson :: Size -> Json
encodeJson (Size s) = intAsJson s
```

Binary

```
encodeBinary :: Size -> ByteString
encodeBinary (Size s) = intAsByteStringBE s
```

AvailableTopics

In a platform's implementation, an `AvailableTopics` is just a mapping from `Topic` to `Size` — this could be a HashMap, a B-tree map, or a JSON object.

```
data AvailableTopics = AvailableTopics (Map Topic Size)
```

Generate

```
generate :: Gen AvailableTopics
generate = do
  pairs <- arrayOf (pairOf arbitraryTopic arbitrarySize)
  return (AvailableTopics (arrayToMap pairs))
```

JSON

This data type is the same as a JSON object of integers, so we'll use that for its JSON encoding.

```
encodeJson :: AvailableTopics -> Json
encodeJson (AvailableTopics ts) = mapAsJsonObject ts
```

Binary

A `Topic` tells us how many bytes it uses in its first 32-bit value, while `Size` is always 4 bytes because its a 32-bit integer. Therefore, a pair of a `Topic` and `Size` can be stored right next to each other, and the only thing we have have to worry about is storing *how many* pairs there are.

```
encodeBinary :: AvailableTopics -> ByteString
encodeBinary (AvailableTopics ts) =
  (intAsByteStringBE (length ts :: Int32))
    ++ (arrayAsByteString (map pairToByteString (mapToArray ts)))
  where
    pairToByteString :: (Topic, Size) -> ByteString
    pairToByteString (t,s) = (encodeBinary t) ++ (encodeBinary s)
```

Generating

A `Generating` message is sent by the peer doing the generating, and received by the peer operating. There are a few options to consider — it's an enumerated type. Furthermore, it's defined here generically over its serialized type, but the idea is the same.

```
data Generating target
  = Generated (value :: target) (operation :: target)
  | BadResult (result :: target)
  | YourTurn
  | ImFinished
  | NoParseOperated (result :: target)
```

Note that we are not using type `T` or `OperationsOnT` here — we may have many different types to deal with, and therefore can't be constrained to one universal type. However, there is only one `Target` type over the `Socket` we communicate over, and can therefore be defined against that.

JSON

This type's enumerated options are distinguished by varying keys in a JSON object.

```
encodeJson :: Generating Json -> Json
encodeJson x = case x of
  Generated value operation -> {"generated": {"value": value, "operation": operation}}
  BadResult result -> {"badResult": result}
  YourTurn -> stringAsJson "yourTurn"
  ImFinished -> stringAsJson "imFinished"
  NoParseOperated result -> {"noParseOperated": result}
```

Binary

The different enumerated options will be distinguished by a varying initial byte flag.

```
encodeBinary :: Generating ByteString -> ByteString
encodeBinary x = case x of
  Generated value operation ->
    (byteAsByteString 0)
      ++ (byteStringWithLength value)
      ++ (byteStringWithLength operation)
  BadResult result ->
    (byteAsByteString 1)
      ++ (byteStringWithLength result)
  YourTurn -> byteAsByteString 2
  ImFinished -> byteAsByteString 3
  NoParseOperated result ->
    (byteAsByteString 4)
      ++ (byteStringWithLength result)
```

Where `byteStringWithLength` prefixes the *ByteString*'s byte-length as a 32-bit integer.

Operating

An `Operating` message is sent by the peer doing the operating, and received by the peer that generated.

```
data Operating target
  = Operated (result :: target)
  | NoParseValue (value :: target)
  | NoParseOperation (operation :: target)
```

JSON

```
encodeJson :: Operating Json -> Json
encodeJson x = case x of
  Operated result -> {"operated": result}
  NoParseValue value -> {"noParseValue": value}
  NoParseOperation operation -> {"noParseOperation": operation}
```

Binary

```
encodeBinary :: Operating ByteString -> ByteString
encodeBinary x = case x of
  Operated result ->
    (byteAsByteString 0)
      ++ (byteStringWithLength result)
  NoParseValue value ->
    (byteAsByteString 1)
      ++ (byteStringWithLength value)
  NoParseOperation operation ->
    (byteAsByteString 2)
      ++ (byteStringWithLength operation)
```

Where `byteStringWithLength` prefixes the *ByteString*'s byte-length as a 32-bit integer.

First

These are the messages sent by the `First` party in the protocol.

```
data First target
  = Topics AvailableTopics
  | BadStartSubset
  | FirstGenerating Topic (Generating target)
  | FirstOperating Topic (Operating target)
```

JSON

```
encodeJson :: First Json -> Json
encodeJson x = case x of
  Topics availableTopics ->
    {"availableTopics": encodeJson availableTopics}
  BadStartSubset ->
    "badStartSubset"
  FirstGenerating topic generating ->
    { "firstGenerating":
      { "topic": encodeJson topic
      , "generating": encodeJson generating
      }
    }
  FirstOperating topic operating ->
    { "firstOperating":
      { "topic": encodeJson topic
      , "operating": encodeJson operating
      }
    }
```

Binary

```
encodeBinary :: First ByteString -> ByteString
encodeBinary x = case x of
  Topics availableTopics ->
    (byteAsByteString 0)
      ++ (encodeBinary availableTopics)
  BadStartSubset ->
    byteAsByteString 1
  FirstGenerating topic generating ->
    (byteAsByteString 2)
      ++ (encodeBinary topic)
      ++ (encodeBinary generating)
  FirstOperating topic operating ->
    (byteAsByteString 3)
      ++ (encodeBinary topic)
      ++ (encodeBinary operating)
```

Second

These are the messages sent by the `Second` party in the protocol.

```
data Second target
  = BadTopics AvailableTopics
  | Start (Set Topic)
  | SecondOperating (Operating target)
  | SecondGenerating (Generating target)
```

JSON

```
encodeJson :: Second Json -> Json
encodeJson x = case x of
  BadTopics availableTopics ->
    {"badTopics": encodeJson availableTopics}
  Start topics ->
    {"start": encodeJson topics}
  SecondOperating topic operating ->
    { "secondOperating":
      { "topic": encodeJson topic
      , "operating": encodeJson operating
      }
    }
  SecondGenerating topic generating ->
    { "secondGenerating":
      { "topic": encodeJson topic
      , "generating": encodeJson generating
      }
    }
```

Binary

```
encodeBinary :: Second ByteString -> ByteString
encodeBinary x = case x of
  BadTopics availableTopics ->
    (byteAsByteString 0)
      ++ (encodeBinary availableTopics)
  Start topics ->
    (byteAsByteString 1)
      ++ (encodeBinary topics)
  SecondOperating topic operating ->
    (byteAsByteString 2)
      ++ (encodeBinary topic)
      ++ (encodeBinary operating)
  SecondGenerating topic generating ->
    (byteAsByteString 3)
      ++ (encodeBinary topic)
      ++ (encodeBinary generating)
```