Application Notes

Dynamically Defined Data Structures

01 July 2024

Table of Contents

1. Introduction

1.1. Abstract

This document defines an info set and wire-format (byte-stream) representation for members of the info set. It states requirements on implementations of this info set, and describes the semantics of various common operations, particularly comparison operations, on elements of the info set.

The name 'Dynamically Defined Data Structures', and the synonymous abbreviations 'DDDS' and 'D3S', refer to the info set defined by this document.

Elements of the info set are called values.

1.2. Document conventions

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this document are to be interpreted as described in **[Bra97]**.

Paragraphs in italics contain informative rather than normative text. The lead paragraph to tables and figures state whether their contents are normative or informative.

2. Info set

The space of values defined by D3S is partitioned — into *atomic* and *aggregate* values; each of these subspaces is further partitioned into types. The *atomic types* are `int`, `string`, `symbol` and `byte-block`; the *aggregate types* are `list`, `set` and `map`. Every D3S value is an element of precisely one of these types.

The wire format defined in Wire format imposes restrictions on the space of values beyond their abstract mathematical constraints. In particular, due to limitations in the wire-format representation, the spaces of integers, strings, symbols, byte-blocks and sets are finite: very large integers and strings have no representation. However, implementations may place lower limits on the space of values they can represent. The following sections specify lower bounds — i.e., 'minimum maxima' — on these limits.

These bounds represent the absolute minimum acceptable bounds; implementations are strongly encouraged to avoid imposing arbitrary limitations.

This is a specification of the D3S info set and encoding, and not of any particular protocol. Cooperating implementations may, and probably will, impose additional restrictions on the values that they are willing to accept; they may also relax restrictions stated here.

2.1. Types

The following sections specify the properties and values belonging to the seven types described above.

2.1.1. int: Integers

The type *integer* consists of an implementation-defined subset of the ring of integers.

Integers shall be compared solely on the basis of their numerical values. D3S does not distinguish between `fixnums` and `bignums`: they are all simply integers.

These names come from the Lisp community. A `fixnum` ('fixed-size number') is an integer which fits into a machine word, while a `bignum` may require an unbounded amount of storage. Even so, the concepts apply to many implementations and languages; for example, Python 2 has two integer types: `int` is a simple `fixnum` type, while `long` is a `bignum` type — Python steadily eroded the distinctions between these

types, and Python 3 no longer acknowledges any difference, having a single `int` *type for both.*

An integer value `a` shall compare equal (respectively, not equal, less than, etc.) to an integer value `b` if and only if the mathematical value of `a` is equal (respectively, not equal, less than, etc.) to that of `b`, regardless of the way in which `a` and `b` happen to be represented within the implementation.

A D3S implementation shall be able to represent every integer with absolute value less than 2^{32768}, if available memory permits.

2.1.2. string: Text strings

A `string` is a (finite, possibly empty) sequence of Unicode **[Con07]** characters; an implementation may impose limitations on the space of strings. Examples of such limitations are maximum length, or permitted characters.

A D3S implementation shall not attempt to canonicalize strings, for example, to prefer or avoid combining characters or a particular letter case.

Strings containing Unicode surrogate pairs are erroneous. An implementation should fail to construct such strings; a string containing a surrogate pair shall not compare equal to a string not containing a surrogate pair. Beyond this, the semantics of such erroneous strings are undefined.

It is permitted for a string to contain a character with code-point zero — a 'null character'. Such a character shall not be considered a string terminator.

An implementation shall be able to represent all strings containing fewer than `65536` characters drawn from `U+0009`, `U+000A`, `U+000D` and `U+0020` up to `U+007E`.

The above characters are the ASCII tab, linefeed, carriage return, space and the printable ASCII characters.

2.1.3. symbol: Symbolic names

A `symbol` is an object with a textual name. The name is a string (see string: Text strings); an implementation may impose stricter limitations on the names of symbols than it imposes on strings. Note that the types symbol and string are still disjoint: a symbol is not a string, though its name is a string.

Two symbols shall compare equal (respectively, not equal, less than, etc.) if and only if their names compare equal (respectively, not equal, less than, etc.).

An implementation shall be able to represent all symbols whose names consist of fewer than 256 characters drawn from `U+0030` up to `U+0039`, `U+0041` up to `U+005A`, `U+005F`, and `U+0061` up to `U+007A`.

The above required characters are the ASCII digits, letters and underscore.

Symbols and strings are of different type. This makes it possible to optimize symbol comparison and symbol lookup.

2.1.4. byte-block: Octet strings

A `byte-block` is a (finite, possibly empty) sequence of octets, i.e., nonnegative integers with absolute value less than 255.

An implementation shall be able to represent all byte blocks whose length is less than `65536` octets, subject to available memory.

Byte blocks are distinguished from strings. Under certain circumstances it may be necessary to recode text strings, such as for presentation to a user, or for communication with an external system. It is incorrect to recode byte-blocks.

2.1.5. list: Ordered sequences of values

A `list` is an ordered (finite, possibly empty) sequence of values, called the *elements* of the list. The elements of a list need not be distinct.

Two lists shall compare equal if and only if they have the same number of elements, and their respective elements are equal. The relative ordering of lists is not specified.

The number of elements in a list is called the *length* of the list. Each individual element of a list is assigned an integer index based on its position in the list: if the length of a list is `n`, then the indices shall be the integers `0`,`1`,⋯,`n-1` in turn.

An implementation shall be able to represent all lists whose length is less than 256, subject to available memory.

2.1.6. set: Unordered collections of distinct atomic values

A `set` is an unordered (possibly empty) collection of distinct atomic values, called the *elements* of the set. A set shall not contain an aggregate value; an implementation shall fail to construct a set which is invalid in this regard.

Two sets shall compare equal if and only if every element of one is also an element of the other. The relative ordering of sets is not specified.

An implementation shall be able to represent all sets containing fewer than 256 elements.

Sets are distinguished from lists precisely because sets are not ordered. This feature makes them useful for containing flags, i.e., values whose presence or absence is significant. The wire format defines a canonical representation for sets.

2.1.7. map: Associative arrays

A `map` (also known as an associative array, dictionary, or partial function) is an unordered collection of distinct ordered pairs of values. The individual pairs are called *associations*; the two elements of an association are the `key` and the `value` respectively. The key of an association shall be an atomic value; an implementation shall fail to construct a map containing an association whose key is an aggregate value. There is no restriction on the type of an association's value. A map shall not contain two associations whose keys compare equal.

Two maps shall compare equal if and only if, for every association in one, there is an association in the other for which the keys and values compare equal. The relative order of maps is not specified.

An implementation shall be able to represent all maps containing fewer than 256 associations.

2.2. Other restrictions

2.2.1. Circularity

The *subvalues* of a value are defined as follows.

1. An atomic value has no subvalues.
2. The subvalues of a list or set are the elements of the list or set, together with the subvalues of those elements.
3. The subvalues of a map are the keys and values of the map's associations, together with the subvalues of those keys and values.

A value is *circular* if it is equal to one of its subvalues. A value is *infinite* if it has infinitely many distinct subvalues.

An implementation shall not construct an infinite or circular value.

3. Wire format

This section specifies an encoding for D3S values as octet sequences.

Each D3S value may have many distinct encodings. There is a unique canonical encoding of each D3S value.

The flexibility permitted by the existence of multiple encodings permits efficient implementations. The canonical encoding is useful in specific circumstances, such as where the encodings are to be hashed, but determining the canonical encoding of a value may be significantly more computationally expensive than merely constructing an arbitrary encoding.

3.1. Wire format specification

This section describes the wire format encoding in terms of how to *decode* it If an encoding decodes to a particular value then that it is an *encoding* of that value.

Encoding are *self-delimiting,* i.e., it's possible to determine where an encoding ends. This fact makes it possible to concatenate encodings without causing ambiguity.

3.1.1. Raw integer format

Non-negative integers are used throughout the encoding, to represent integer values, symbols, and to signify lengths. Such integers are always represented radix-256, most significant octet first.

3.1.2. Encoding structure

An encoding of a value consists of three logical parts:

- a *format* code, which describes how to interpret the remaining octets of the encoding as a value;
- an *indicator*, which is a non-negative integer; and
- a *payload*, whose length and nature is determined by the format and indicator.

In order to save space, the format and indicator can be packed into a single octet.

In the following descriptions, `d` denotes the indicator value. The following formats are used.

Non-negative integer

The result shall simply be the integer `d`. The payload shall be empty.

Non-positive integer

The result shall be the integer `-d`.

It is possible that `d=0`*.*

The payload shall be empty.

String

The format and indicator shall be followed by a further `d`-octet payload containing a Unicode string encoded using UTF-8 **[Con07, 3.9]**. If the payload octets are ill-formed UTF-8, the encoding is invalid. The result shall be a string whose contents are the Unicode string encoded in the payload.

Symbol, by name

The format and indicator shall be followed by a further `d`-octet payload containing a Unicode string encoding using UTF-8. If the payload octets are not valid UTF-8, the encoding is invalid. The result shall be the symbol whose name is the Unicode string encoded in the payload.

Byte-block

The format and indicator shall be followed by a further `d`-octet payload. The result shall be a byte-block whose contents are precisely the payload.

List

The format and indicator shall be followed by a payload consisting of the concatenation of `d` further value encodings. The result shall be a list whose elements are the values encoded in the payload, in order.

Set

The format and indicator shall be followed by a payload consisting of the concatenation of `d` further value encodings. The result shall be a set whose elements are the values encoded in the payload.

Map

The format and indicator shall be followed by a payload consisting of the concatenation of `2d` further value encodings. The result shall be a map whose associations are constructed by taking the values encoded in the payload in pairs, first key, then value.

3.1.3. Specification

An encoding cannot be empty. The first octet of the encoding describes how to interpret the remaining octets. **Table 3.2.1** describes how to determine the format and indicator from this first octet. No other octet begins a valid encoding. Implementations shall not extend the wire format. An encoder shall not construct invalid encodings; a decoder shall reject invalid encodings.

Table 3.2.1: Encodings by the first octet (normative)

First octet	Interpretation
`000d dddd`$_2$	Format is 'non-negative integer'; indicator is binary `d dddd`.
`0010 dddd`$_2$	Format is 'string'; indicator is binary `dddd`.
`0011 dddd`$_2$	Format is 'symbol, by name'; indicator is binary `dddd`.
`1000 dddd`$_2$	Format is 'byte-block'; indicator is binary `dddd`.
`1001 dddd`$_2$	Format is 'list'; indicator is binary `dddd`.
`1010 dddd`$_2$	Format is 'set'; indicator is binary `dddd`.
`1011 dddd`$_2$	Format is 'map'; indicator is binary `dddd`.
`1100 tttt`$_2$	Format is given by `tttt`, as shown in **Table 3.2.2**; indicator is stored in the following octet.
`1101 tttt`$_2$	Format is given by `tttt`, as shown in **Table 3.2.2**; indicator is stored in the following 2 octets.
`1111 0000`$_2$	A padding octet. Ignore this octet and interpret the next as being the start of an encoding.
`1111 0010`$_2$	Format is given by the next octet, as shown in **Table 3.2.2**; indicator is stored in the following 4 octets.
`1111 0011`$_2$	Format is given by the next octet, as shown in **Table 3.2.2**; indicator is stored in the following 8 octets.
`1111 0100`$_2$	Format is 'non-negative integer'. The following octets shall encode a byte-block value; the indicator is the integer encoded in the payload of the byte-block.
`1111 0101`$_2$	Format is 'non-positive integer'. The following octets shall encode a byte-block value; the indicator is the integer encoded in the contents of the byte-block.

Table 3.2.2: Format codes (normative)

If the code is not one of those listed below, the encoding is invalid.

Code	Format
0000_2	Non-negative integer
0001_2	Non-positive integer
0010_2	String
0100_2	Symbol, by name
0101_2	Byte-block
1000_2	List
1001_2	Set
1010_2	Map

3.2. Canonical format

The following rules describe how to construct the canonical encoding of a value.

3.2.1. Ordering

Sets and maps do not impose an ordering on their contents.

Types already describes how values of particular atomic types order relative to each other; these orderings are combined to form a total ordering of all atomic values by defining an ordering on *types*:

integer < *symbol* < *string* < *byte-block*.

If x and y are two atomic values, then x shall compare less than y if and only if *either* x and y have the same type, and x<y according to the rules stated in Types; or the types of x and y differ and the type of x is less than the type of y as shown above.

In order to ensure uniqueness of the canonical encoding for sets and maps:

- the elements of a set shall be encoded in ascending order according to the above ordering;

and

- the associations of a map shall be encoded in ascending order of their keys according to the above ordering.

This is sufficient since (a) set elements and association keys are atomic, and (b) two distinct associations within a map cannot have equal keys.

3.2.2. Encoding choices

The following rules specify how to select among the various encoding choices . Earlier rules take precedence over later rules.

1. The first octet of the encoding shall not be `240=1111 0000`$_2$ That is, padding octets shall not appear in a canonical encoding.
2. The first octet of the encoding shall be the numerically least first octet of any valid encoding of the value.
3. The encoding chosen shall be the shortest valid encoding of the value.

Some examples may help.

- *The canonical encoding of the integer* `0` *is* `00` *by rule 2.*
- *The canonical encoding of the integer* `65536` *is* `f2 00 00 01 00 00`*, even though* `f4 83 01 00 00` *is shorter, because* `f2 < f4` *and rule 2 takes precedence over rule 3.*

4. References

[Bra97] S. Bradner; 'Key words for use in RFCs to Indicate Requirement Levels'; RFC 2119 (Best Current Practice); March 1997; URL http://www.ietf.org/rfc/rfc2119.txt.

[Con07] Unicode Consortium; 'The Unicode Standard 5.0'; 2007; URL http://www.unicode.org/versions/Unicode5.0.0/.

[LMS05] P. Leach, M. Mealling, and R. Salz; 'A Universally Unique IDentifier (UUID) URN Namespace'; RFC 4122 (Proposed Standard); July 2005; URL http://www.ietf.org/rfc/rfc4122.txt.