Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 20 July 2026

P. Thierry
Comonad Dev

16 January 2026

Binary Uniform Language Kit 1.0
draft-thierry-bulk-06

Abstract

This specification describes a uniform, decentrally extensible and efficient format for data serialization.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 July 2026.

Table of Contents

1. Introduction

1.1. Rationale

This specification aims at finding an original trade-off between uniformity, generality, extensibility, decentralization, compactness and processing speed for a data format. It is our opinion that every widely used existing format occupy a different position than this one in the solution space for formats, that none is better on all axes, and that this one is the current best on several axes, hence this new design. It is also our opinion that some of those existing formats constitute an optimal solution for their specific use case, either in a absolute sense, or at least at the time of their design. But the ever-changing field of IT now faces new challenges that call for a new approach.

In particular, whereas the previous trend for Internet and Web standards and programming tools has been to create human-readable syntaxes for data and protocols, the advent of technologies like protocol buffers [protobuf], Thrift [Thrift], the various binary serializations for JSON like Avro [Avro] or Smile [Smile], or the binary HTTP/2 [HTTP2] seem to indicate that the time is ripe for a generalized use of binary, reserved until now for the low-level protocols. The lessons about flexibility learnt in the previous switch from binary to plain text can now be applied to efficient binary syntaxes.

1.1.1. Definitions

By uniformity, we mean the property of a syntax that can be parsed even by an application that doesn’t understand the semantics of every part of the processed data. Of course, almost all syntaxes that feature uniformity contain a limited number of non uniform elements. Also, uniformity really only has value in the face of extension, as a fixed syntax doesn’t need uniformity (it only makes the implementation simpler).

Almost all extensible syntaxes have their extensible part uniform to a great degree. In this specification, uniformity is hence evaluated on two criteria: first, the number of non uniform elements (and, incidentally, their diversity), second, the fact that the uniformity of the extensible part is not a limitation to the users (i.e. that the temptation to extend the format in a non-uniform way is as absent as possible).

A good counter-example is found in most programming languages. Adding a new branching construct cannot be done in a terse way without modifying the underlying implementation. Such a construct either cannot be defined by user code (because of evaluation rules) or can in a terribly verbose and inconvenient way (with lots of

boilerplate code). Notable exceptions to this limitation of programming languages are Lisp, Haskell and stack programming languages.

On the other hand, a stack programming language is the canonical example of a non-uniform language. Each operator takes a number of operands from the stack. Not knowing the arity of an operator makes it impossible to continue parsing, even when its evaluation was optional to the final processing. In the design space, stack programming languages completely sacrifice uniformity to achieve one of the highest combination of extensibility, compactness and speed of processing.

By generality, we mean the ability of a syntax to lend itself to describe any kind of data with a reasonable (or better yet, high) level of compactness and simplicity. For example, although both arrays and linked lists could be considered very general as they are both able to store any kind of data, they actually are at the respective cost of complexity (arrays need the embedding of data structure in the data or in the processing logic) and size (in-memory linked lists can waste as much as half or two third of the space for the overhead of the data structure).

By decentralization, we mean the ability to extend the syntax in a way that avoid naming collisions without the use of a central registry. Note that the DNS, as we use it, is NOT decentralized in this sense, but distributed, as it cannot work without its root servers and prior knowledge of their location.

1.1.2. State of the art

Uniformity, generality and extensibility are usually highly-valued traits in formats design. Programming languages obviously feature them foremost, although their generality usually stops at what they are supposed to express: procedures. Most of them are ill-suited to represent arbitrary data, but notable exceptions include Lisp (where "code is data") and Javascript, from which a subset has been extracted to exchange data, JSON, which has seen a tremendous success for this purpose. JSON may lack in generality and compactness, but its design makes its parsing really straightforward and fast. All of them, though, lack decentralization. Some of them make it possible to extend them in a distributed way if some discipline is followed (for example, by naming modules after domain names), but the discipline is not mandatory (and even with domain names, a change of ownership makes it possible for name collisions).

The SGML/XML family of formats also feature uniformity, generality and extensibility and actually fare much better than programming languages on the three fronts. XML namespaces also make XML naming distributed and there have been attempts at making it compact (e.g. EXI from W3C, Fast Infoset from ISO/ITU or EBML).

All the previously cited formats clearly lack compactness, although just applying standard compression techniques would sacrifice only very little processing time to gain huge size reductions on most of their intended use cases, but compression may not address their ineffectiveness at storing arbitrary bytes.

So-called binary formats pretty much exhibit the opposite trade-offs. Most of them are not uniform to achieve better compactness. Some are specifically designed for a great generality, but many lack extensibility. When they are extensible, it’s never in a decentralized way, again for reasons that have to do with compactness. They are usually extremely fast to parse.

Actually, many binary formats are not so much formats as they are formats frameworks, and exclude extensibility by design. For each use case, an IDL compiler creates a brand new format that is essentially incompatible with all other formats created by the same compiler (EBML specifically cites this property among its own disadvantages). If the IDL compiler and framework are correctly designed, such a format usually represent an optimum in compactness and speed of processing, as the compiler can also automatically generate an ad-hoc optimized parser.

Where extensibility has been planned in existing formats, it often doesn’t get used that much or at all because of the complications around it. Many binary formats include reserved values meant to extend them to future uses, like the CM field in the ZIP format. A case like this one faces an chicken-and-egg problem: if you don’t write and get a specification officially adopted, implementations might not want to include your extension, but if your extension is purely theoretical and hasn’t been tested in the wild, you may face resistance to get it officially adopted. This is probably why even though most compression formats include the ability to later encode other compression methods, each new compression method usually comes with its own format.

When extensions are managed with any form of registry, another issue is that you usually need to reserve a large set of values for free experimentation, and once an extension gains any traction while in experimentation, its authors face the difficulty to switch all existing implementations to the definitive values they'll get. And how experimenters choose their temporary values makes them vulnerable to conflicts with others.

1.2. Format overview

A BULK stream is a stream of 8-bit bytes, in big-endian order. Parsing a BULK stream yields a sequence of expressions, which can be either atoms or forms, which are sequences of expressions. The syntax of forms is entirely uniform, without a single exception: a starting byte marker, a sequence of expressions and an ending byte marker. Among atoms, only nil (the null byte) and arrays have a special syntax, for efficiency purposes. Even booleans and floating-point numbers follow the uniform syntax that every other expression follows.

Non uniform atoms start with a marker byte, followed by a static or dynamic number of bytes, depending on the type.

Any other atom is a reference, which consists of a namespace marker (in almost all cases, a single byte) followed by an identifier within this namespace (a single byte). All in all, a very little sacrifice is made in compactness for the benefit of a very simple syntax: apart from nil and small integers, nothing is smaller than 2 bytes, and as most forms involve a reference followed by some content, a form is usually 4 bytes + its content.

A namespace marker in a BULK stream is associated to a namespace identified by some identifier guaranteed to be unique without coordination (like a UUID or cryptographical hash), thus ensuring decentralized extensibility. The stream can be processed even if the application doesn't recognize the namespace. Parsing remains possible thanks to the uniform syntax.

Combination of BULK namespaces, BULK streams and even other formats doesn't need any content transformation to work. Here are some examples:

* The content of a BULK stream, enclosed in list starting and ending byte markers, constitute a valid BULK expression. Thus BULK streams can be packed or annotated within a BULK stream without modification. Annotation use cases include adding metadata or cryptographic signature.

* A BULK format could specify in its syntax the place for an expression holding metadata. Whether the specification provides its own metadata forms or not, an application could use a BULK serialization for MARC, TEI Header, XML or RDF for this metadata expression. The vocabulary selected would be univocally expressed by the namespace and every vocabulary would be parsed by the same mechanisms.

* Whenever a content must be stored as-is instead of serialized, or a highly-optimized ad hoc serialization exists for some data, anything can always be stored within an array. They can contain arbitray bytes and there is no limit to their size.

Furthermore, BULK expressions can be evaluated. Most expressions evaluate to themselves, but some evaluate by default to the result of a pure function call, making it possible to serialize data in an even more compact form, by eliminating boilerplate data and repeated patterns.

1.3. Conventions and Terminology

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 RFC 2119 [RFC2119].

Literal numerical values are provided in decimal or hexadecimal as appropriate. Hexadecimal literals are prefixed with 0x to distinguish them from decimal literals.

The text notation of the BULK stream uses mnemonics for some bytes sequences. Mnemonics are series of characters, excluding all capital letters and white space, like this-is-one-mnemonic or what-the-%則!?#-is-that?. They are always separated by white space. Outside the use of mnemonics, a sequence of bytes (of one or more bytes) can be represented by its hexadecimal value as an unsigned integer prefixed by 0x (e.g. 0x3F or 0x3A0B770F). Such a sequence of bytes can include dashes to make it more readable (e.g. 0xDDA37D36-85E6-4E6D-9B51-959E1CCE366C). Some types in this specification define a special syntax for their representation in the text notation.

In the grammar, a shape is a pattern of bytes, following the rules of the text notation for a BULK stream. Apart from mnemonics and fixed sequences of bytes, a shape can contain:

* an arbitrary sequence of a fixed number of bytes, represented by its size, i.e. a number of bytes in decimal immediately followed by a B uppercase letter (e.g. 4B)

* a typed sequence of bytes, represented by the name of its type, a capitalized word (e.g. Foo); this means a sequence of bytes whose specific yield (cf. Section 2.1) has this type

* a named sequence of bytes (of zero or more bytes), represented by a series of any character excluding '{}' between '{' and '}' (e.g. {quux}); a named sequence can be typed or sized, in which case it is immediately followed by ':' and a type or size (e.g. {quux}:Bar or {quux}:12B)

When an entire shape describes the byte sequence of an atom, it is the normative specification for parsing it, but shapes of forms are only normative with respect to their default evaluation. A reference defined with a form shape can be used in different shapes, albeit with different semantics and value and even when used in its default shape, a processing application MAY give it alternative semantics.

For example, this specification defines a way do specify a string encoding with forms of the shape (stringenc {enc}:Expr). But the shapes (stringenc {arg1}:Int {arg2}:Int) or ({arg1}:Int stringenc {arg2}:Int) are syntactly valid. They just have unspecified semantics, as far as this specification is concerned.

Some identifiers are expected to be verifiable against a byte sequence. This means that there must be an algorithm that, given the byte sequence as input, produces the identifier as output and, given a different byte sequence, would produce a different identifier. Because this verification has security implications, the algorithm used should have the same guarantees than a cryptographic hash function in terms of collisions.

2. BULK syntax

A BULK stream is a sequence of 8-bit bytes. Bits and bytes are in big-endian order. The result of parsing a BULK stream is a list of abstract data, called the abstract yield. BULK parsing is injective: a BULK stream has only one abstract yield, but different BULK streams can have the same abstract yield (if they associate namespaces to different markers, see namespaces (Section 3.1.3)).

A processing application is not expected to actually produce the abstract yield, but an adaptation of the abstract yield to its own implementation, called the concrete yield. Also, some expressions in a BULK stream may have the semantics of a transformation of the abstract yield. A processing application MAY thus not produce or retain the concrete yield but the result of its transformation. This specification deals mainly with the byte sequence and the abstract yield and occasionnally provide guidelines about the concrete yield.

Of course, a processing application MAY not produce any concrete yield at all but produce various data structures and side effects from parsing the BULK stream (for example, an event sourced application may read its event log from a BULK stream and build its application state by applying the events, discarding each of them as soon as it has been applied).

The abstract yield is a list of expressions. Expressions can be atoms or forms. Forms are lists of expressions. If a byte sequence is parsed as an expression, this byte sequence is said to encode this expression.

When a sequence of bytes is named in a shape, its name can be used in this specification to designate either the byte sequence, or the expression or list of expressions it encodes. When there could be ambiguity, this specification specifies which is designated.

2.1. Parsing algorithm

The parser operates with a context, which is a list of expressions. Each time an expression is parsed, it is appended at the end of the context. The initial context is the abstract yield.

At the beginning of a BULK stream and after having consumed the byte sequence encoding a complete expression, the parser is at the dispatch stage. At this stage, the next byte is a marker byte, which tells the parser what kind of expression comes next (the marker byte is the first byte of the sequence that encodes an expression). The expression appended to the context after reading a byte sequence is called the specific yield of the byte sequence.

The 0x01 and 0x02 marker bytes are special cases. When the parser reads 0x01, it immediately appends an empty list to the current context. This list becomes the new context. This new context has the previous context as parent. Then the parser returns to its dispatch stage. When the parser reads 0x02, it appends nothing to the context, but instead the parent of the current context becomes the new context and the parser returns to the dispatch stage. Thus it is a parsing error to read 0x02 when the context is the abstract yield.

Some forms have side-effects in their semantics. Those side-effects MUST not affect the parsing of any expression. They can affect evaluation, in which case they MUST only affect the evaluation of expressions in the scope of the form. The outer scope of an expression is the part of its context that follows the expression. Some forms MAY define an inner scope in their shape. The scope of an expression is the union of the outer and inner scopes. This makes BULK lexically scoped.

Whenever a parsing error is encountered, parsing of the BULK stream MUST stop.

2.1.1. Summary of marker bytes

marker	shape
00	nil (Section 2.3.1)
01	((Section 2.2.1)
02	) (Section 2.2.2)
03	# Nat {content} (Section 2.3.2.1)
040F	reserved (Section 2.3.3)
107F	references (Section 2.3.4)
80BF	w6[value] (Section 2.3.2.3)
C0FF	#[size] {content} (Section 2.3.2.2)

Table 1

2.1.2. Evaluation

A processing application MAY implement evaluation of BULK expressions and streams. When evaluating a BULK stream, when the parser gets to the dispatch stage and the context is the abstract yield, the last expression in the context is replaced by what it evaluates to. (of course, this description is supposed to provide the semantics of BULK evaluation, but a processing application MAY implement evaluation with a different algorithm as long as it provides the same semantics)

The default evaluation rule is that an expression evaluates to itself. A name within a namespace can have a value, which is what a reference associated to this name evaluates to. A reference whose marker value is associated to no namespace or whose name has no value evaluates to itself. How self-evaluating BULK expressions are represented in the concrete yield is application-dependent, but future specifications MAY define a standard API to access it, similar to the Document Object Model for XML.

The evaluation of a form obeys a special rule, though: if the first expression of the form has type Function, that function is called with an argument list and the form evaluates to the return value if it’s an atom or the evaluation of the return value if it is a form. If the function has type LazyFunction, the argument list is the rest of the form. If the function has type EagerFunction, the argument list is the rest of the form, where each expression is replaced by what it evaluates to. Any expression that has type LazyFunction or EagerFunction also has type Function.

A form whose first expression doesn’t have type Function evaluates to itself.

When an application evaluates a BULK expression, it MUST verify that evaluation will terminate in a finite number of evaluation steps. An application MAY verify finite termination statically or dynamically. For example, an application MAY stop evaluation in error after a predetermined number of steps.

2.2. Forms

2.2.1. starting marker byte

marker 0x01

mnemonic (

2.2.2. ending marker byte

marker 0x02

mnemonic)

2.2.3. Difference between sequence and form

There is a difference between a byte sequence encoding several expressions among the current context and a byte sequence encoding a form (i.e. a single expression that is a list of expressions). As an example, let’s examine several forms of the shape (foo {bar}).

* In the form (foo nil nil nil), {bar} encodes 3 expressions, and they are three atoms in the yield.

* In the form (foo nil), {bar} is a single expression in the yield, and that expression is an atom.

* In the form (foo (nil nil nil)), {bar} is also a single expression in the yield, and that expression is a form, a list in the yield.

In a shape, when a byte sequence must yield a single expression, it has the type Expr. So the last two examples fit the shape (foo {seq}:Expr) but not the first. When a byte sequence must yield a form, it has type Form. Thus the shape (foo {bar}:Form) is equivalent to (foo ({bar})). Either one MAY be used.

2.3. Atoms

2.3.1. nil

marker 0x00 (mnemonic: nil)

shape nil

Apart from being a possible short marker value, the fact that the 0x00 byte represents a valid atom means that a series of null bytes is a valid part of a BULK stream, thus making the format less fragile. In a network communication, nil atoms can be sent to keep the channel open. They can also be used as padding at the end of a form or between forms.

2.3.2. Arrays

Arrays can be used to store arbitrary bytes.

An array can be interpreted either as a bits sequence or as an unsigned integer in binary notation. The choice depends on the context and the application. Actually, many processing applications may not need make any choice, as most programming language implementations actually also confuse unsigned integers and bits sequences to some extent. Expressions that are unsigned integers (that is, natural numbers) have type Nat (whether they are encoded as an array or not).

Big arrays typically store the content of a file or a binary message of another format. They can also be used to store a vector or matrix of fixed-size elements.

In any case, the semantics of the content must be inferred by the processing application; where ambiguity can appear, an application SHOULD enclose the array in a form that makes the semantics explicit (e.g. string (Section 3.1.5.4), string* (Section 3.1.5.5), blob (Section 3.1.5.6), or unsigned-int (Section 3.1.6.1)).

Because BULK arrays have no end markers, the payload of a BULK array can constitute the end of the stream.

The start and end of an array are known without reading its content, which means that its content can be skipped in constant time and mapped in memory (or read lazily by any other means).

Because BULK can use integers with arbitrary size to store the size of an array, BULK arrays have no limit in size.

2.3.2.1. Generic array

marker 0x03 (mnemonic: #)

shape # Nat {content}

Arrays have a special parsing rule. After consuming the marker byte, the parser returns to the dispatch stage. It is a parser error if the parsed expression is not of type Nat or if its value cannot be recognized. This integer is not added to any context, but the parser consumes as many bytes as this integer and they constitute the content of this array.

In the text notation, a quoted string is the notation for a generic array containing the encoding of that string in the current encoding (Section 3.1.5.1), except if the size of the encoding is below 64 bytes, cf. small arrays (Section 2.3.2.2).

Types: Bytes, Nat

In a shape, the type String is synonymous with Bytes, but means that the content of the array is supposed to be taken as a string in the current encoding.

2.3.2.2. Small array

marker 0xC00xFF (mnemonic: #[size])

shape #[size] {content}

Small arrays have a special parsing rule. The 6 least significant bits of the marker byte are treated as un unsigned integer. This integer is not added to any context, but the parser consumes as many bytes as this integer and they constitute the content of this array.

In the text notation, the notation of the marker byte of a small array of size X is #[X]. For example, #[2] 0x1234 is a notation for the bytes 0xC2-1234.

In the text notation, a quoted string is the notation for a small array containing the encoding of that string in the current encoding if the size of the encoding is below 64 bytes.

Types: Bytes, Nat

2.3.2.3. Small unsigned integers

marker 0x800xBF (mnemonic: w6[value])

shape w6[value]

Small unsigned integers have a special parsing rule. The 6 least significant bits of the marker byte are the value encoded by this byte (as bits or as an unsigned integer in binary notation).

In the text notation, the notation of the marker byte of a small unsigned integer of value X is w6[X]. For example, w6[11] is a notation for the byte 0x8B (as is 11, cf. Section 3.1.6).

Types: Bytes, Nat

2.3.2.4. Reprensenting natural numbers

When the syntax of a BULK form mandates that an expression can only be a Nat, an application SHOULD encode it as the smallest possible array using one of the following sizes: 6, 8, 16, 32, or any multiple of 64 bits.

2.3.3. Reserved marker bytes

Marker bytes 0x04−0x0F are reserved for future major versions of BULK. It is a parser error if a BULK stream with major version 1 contains such a marker byte.

2.3.4. References

marker 0x10−0x7F

shape {ns}:1B {name}:1B

0x7F {ns’} {name}:1B

The {ns} byte is a value associated with a namespace, called the namespace marker. Values 0x10−0x13 are reserved for namespaces defined by BULK specifications. Greater values can be associated with namespaces identified by a unique identifier.

The {name} byte is the name within the namespace. Vocabularies with more than 256 names thus need to be spread accross several namespaces.

The specification of a namespace SHOULD include a mnemonic for the namespace and for each defined name. When descriptions use several namespaces, the mnemonic of a reference SHOULD be the concatenation of the namespace mnemonic, ":" and the name mnemonic if there can be an ambiguity. For example, the fp name in namespace math becomes math:fp.

Type: Ref

2.3.4.1. Special case

References have a special parsing rule. In case a BULK stream needs an important number of namespaces, if the marker byte is 0x7F, the parser continues to read bytes until it finds a byte different than 0xFF. The sum of each of those bytes taken as unsigned integers is the namespace marker. For example, the reference encoded by the bytes 0x7F 0xFF 0x8C 0x1A is the name 26 in the namespace associated with 522.

3. Standard namespaces

Standard namespaces have a fixed marker value and are not identified by a unique identifier.

3.1. BULK core namespace

marker 0x10 (mnemonic: bulk)

3.1.1. Version

name 0x00 (mnemonic: version)

shape (version {major}:Nat {minor}:Nat)

When parsing a BULK stream, a processing application MUST determine explicitely the major and minor version of the BULK specification that the stream obeys. This information MAY be exchanged out-of-band, if BULK is used to exchange a number a very small messages, where repeated headers of 6 bytes might become too big an overhead. A processing application MUST NOT assume a default version.

If the version is expressed within a BULK stream, this form MUST be the first in the stream. In any other place, this form has no semantics attached to it. This specification defines BULK 1.0. When writing a BULK stream, an application MUST encode {major} and {minor} by the smallest byte sequence as described in Section 2.3.2.4.

An application writing a BULK stream to long-term storage (e.g. in a file or a database record) SHOULD include a version form.

Two BULK versions with the same major version MUST share the same parsing rules and the same definitions of marker bytes. Changing the syntax or semantics of existing marker bytes and using marker bytes in the reserved interval warrants a new major version. Changing the syntax or semantics of existing names in standard namespaces also.

Adding standard namespaces or adding names in existing standard namespaces warrants a new minor version.

3.1.2. Booleans

3.1.2.1. true

name 0x01 (mnemonic: true)

shape true

Type: Boolean.

3.1.2.2. false

name 0x02 (mnemonic: false)

shape false

Type: Boolean.

3.1.3. Namespaces

3.1.3.1. New namespace

name 0x03 (mnemonic: ns)

shape (ns {marker}:Nat {id}:Expr)

This associates the namespace identified by {id} to the namespace marker {marker}, within the scope of this expression.

3.1.3.2. Package

name 0x04 (mnemonic: package)

shape (package {id}:Expr {namespaces})

This creates a package identified by {id}. Packages are immutable, {id} MUST be verifiable against the byte sequence {namespaces}. {namespaces} MUST be a series of expressions each identifying a BULK namespace.

3.1.3.3. Import

name 0x05 (mnemonic: import)

shape (import {base}:Nat {count}:Nat {id}:Expr)

This associates the first {count} namespaces in the package identified by {id} with a continuous range of marker bytes starting at {base} within the scope of this expression.

Example: (import 28 3 0x0123456789ABCDEF) associates the first 3 namespaces of the package identified by 0x0123456789ABCDEF to the markers 28, 29 and 30.

3.1.4. Definitions

To define a reference is to change the the value of its name in its namespace (as identified by its unique identifier, not the marker value) within a certain scope.

If a BULK stream is not evaluated, the semantics of a definition are entirely application-dependent.

When a BULK stream containing definitions for a namespace comes from a trusted source (i.e. in configuration files of the application, or in the communication with an agent that has been granted the relevant

authority), an application MAY give those definitions long-lasting semantics (i.e. keep the values of the names at the end of parsing). This is the preferred mechanism for bulk namespace definition when the semantics of the defined expressions can be expressed completely by BULK forms.

3.1.4.1. Simple definition

name 0x06 (mnemonic: define)

shape (define {ref}:Ref {value}:Expr)

(define nil {value}:Expr)

This defines the reference {ref} to the yield of {value} in the outer scope of this form.

In any context where there is a default namespace where definitions are made, e.g. verifiable-ns (Section 3.1.4.4), the second shape defines the smallest name that is not yet defined to {value}.

3.1.4.2. Named definition

name 0x07 (mnemonic: mnemonic/def)

shape (mnemonic/def {ref}:Ref {mnemonic}:String {doc}:Expr {value})

(mnemonic/def nil {mnemonic}:String {doc}:Expr {value})

This suggest {mnemonic} as the mnemonic of the name designated by {ref} in its namespace. If {value} is of type Expr, this defines the reference {ref} to {value} in the scope of this form.

{doc} is any expression that provides a documentation for this reference. If it has type Bytes, it MUST be a string. It could be any kind of metadata or document type.

In any context where there is a default namespace where definitions are made, e.g. verifiable-ns (Section 3.1.4.4), the second shape defines the smallest name that is not yet defined to {value}.

3.1.4.3. Namespace description

name 0x08 (mnemonic: ns-mnemonic)

shape (ns-mnemonic {ns}:Expr {mnemonic}:String {doc})

This suggest {mnemonic} as the mnemonic of the namespace designated by {ns} (which can be the integer to which this namespace is associated, a reference in this namespace or the unique identifier of this namespace).

3.1.4.4. Verifiable namespace definition

name 0x09 (mnemonic: verifiable-ns)

shape (verifiable-ns {id}:Expr {marker}:Nat {data}:Expr {mnemonic}:Expr {doc}:Expr {definitions})

inner scope {id} {data} {mnemonic} {doc} {definitions}

This associates the namespace identified by {id} to the namespace marker {marker}, within the scope of this form. Verifiable namespaces are immutable, {id} MUST be verifiable against the byte sequence {marker} {data} {mnemonic} {doc} {definitions}. The semantics of this form is to define in its scope any definition made in the designated namespace within {definitions}.

If {mnemonic} is of type String, then this suggests it as the mnemonic of the namespace. Else it MUST be nil.

If more data than {id} is needed to verify {id} against {definitions} (like the salt of a hash function, or the namespace of a UUID), this data should be provided by {data}. Else {data} MUST be nil.

A verifiable namespace wouldn’t really be immutable if it used definitions from other namespaces that aren’t immutable. To that effect, an application SHOULD stop processing this form with an error when {definitions} contain references from namespaces that cannot be determined to be immutable themselves. The goal is to prevent a user or system to be unwittingly vulnerable, so an application MAY provide an option to accept a specific verifiable namespace, but an application MUST NOT provide an option to accept any vulnerable verifiable namespace. That is, an option like --accept-ns 8f82849556d74466 is acceptable but --disable-immutability-check is not.

3.1.4.5. Array concatenation

name 0x0A (mnemonic: concat)

shape (concat {array1}:Bytes {array2}:Bytes)

Name’s type EagerFunction

Form’s type Bytes

Form’s value the concatenation of {array1} and {array2}.

The value of this form is an array that contains the bytes in array1 followed by the bytes in array2.

3.1.4.6. Substituton

3.1.4.6.1. Substitution function

name 0x0B (mnemonic: subst)

shape (subst {code})

Name’s type LazyFunction

Form’s type EagerFunction

Form’s value A substitution function whose return value is the value of {code}. Within {code}’s specific yield, the names arg and rest are defined:

3.1.4.6.2. Argument

name 0x0C (mnemonic: arg)

shape (arg {n}:Nat)

Name’s type EagerFunction

Form’s type Expr

Form’s value the element number {n} (starting at zero) of the substitution function’s arguments list

3.1.4.6.3. Rest of arguments list

name 0x0D (mnemonic: rest)

shape (rest {n}:Nat)

Name’s type EagerFunction

Form’s type Expr

Form’s value the substitution function’s arguments list without its first {n} elements.

3.1.4.6.4. Examples

Here is a definition of the inverse followed by the numbers 1/2, 1/3 and 1/4:

(define inverse (subst (frac 1 (arg 0)))) (inverse 2) (inverse 3) (inverse 4)

Substitution will splice multiple expressions in place:

The evaluation of ((subst 1 (rest 0) 2) 3 4) must yield the same as (1 3 4 2)

3.1.5. Strings and other typed byte arrays

3.1.5.1. Current encoding

name 0x10 (mnemonic: stringenc)

shape (stringenc {enc}:Encoding)

This tells the processing application that, in the scope of this expression, all expressions that are understood by the application as character strings will be encoded with the encoding designated by {enc}.

As the abstract yield doesn’t contain strings but expressions that will be used as strings by the application, it is not a parsing error if the application doesn’t recognize {enc}. In this situation, it is a parsing error when the application actually needs to decode a byte sequence as a string. It is not a parsing error when a processing application only transmits a byte sequence encoding a string, if it can accurately convey the encoding to the receiving application.

3.1.5.2. IANA registered character set

name 0x11 (mnemonic: iana-charset)

shape (iana-charset {id}:Nat)

This designates the string encoding registered among the IANA Character Sets [IANA-Charsets] whose MIBenum is {id}.

Type: Encoding.

3.1.5.3. Windows code page

name 0x12 (mnemonic: code-page)

shape (code-page {id}:Nat)

This designates the string encoding among Windows code pages whose identifier is {id}.

Type: Encoding.

3.1.5.4. String

name 0x13 (mnemonic: string)

shape (string {string:}Bytes)

This form indicates that the bytes encoded by {string} are meant to be interpreted as a string encoded with the current string encoding.

3.1.5.5. String with explicit encoding

name 0x14 (mnemonic: string*)

shape (string* {enc}:Encoding {string:}Bytes)

This form indicates that the bytes encoded by {string} are meant to be interpreted as a string encoded with the encoding designated by the expression {enc}.

3.1.5.6. Blob

name 0x15 (mnemonic: blob)

shape (blob {blob:}Bytes)

This form indicates that the bytes encoded by {blob} are meant be interpreted as just a raw sequence of bytes, not to be decoded.

3.1.5.7. Nested BULK stream

name 0x16 (mnemonic: nested-bulk)

shape (nested-bulk {embedded}:Boolean {bulk}:Bytes)

This form indicates that the bytes encoded by {bulk} are meant to be interpreted as a BULK stream. If the stream doesn't start with a version form, the stream MUST be assumed to have the same version as the parent stream.

This form can be useful to let the application reading a BULK stream skip parsing a large section.

If {embedded} is true, the default semantics of this form is the same as the semantics of the BULK stream in {bulk}, with the following exception. For example, these two forms have the same semantics:

* (4 5)

* (nested-bulk true #[2] 4 5)

It could be a security risk if a single BULK stream could be parsed into two different abstract yields by two conformant applications, so the semantics of the whole stream cannot change whether {bulk} is decoded or not. For that reason, any effects in the nested stream that affect how BULK expressions are parsed or evaluated (like namespace associations or definitions) MUST be isolated within that form.

For the same security reason, there isn't a (bulk-with-size Nat Expr) form because it would open up the same risk when the size given is not the size of the enclosed expression, accidently or maliciously.

3.1.5.8. Indexed data

When writing a stream containing a big number of expressions where an application may want to access one of those expression without parsing all expressions before, one could imagine as a solution to use pointer-like references that each use the offset of some expression in the stream. This solution creates a security risk, because if reading according to the pointers doesn't produce the same result as parsing the stream without using them, an attacker might use this inconsistency to their advantage, when they can expect one application to use pointers and another application to use normal parsing, especially when the stream is big enough that verifying the consistency of the pointers might be costly enough that it might not be done or not in time to prevent the attack.

Because of that risk, whenever a stream includes indexed BULK expressions, that is, expressions that are meant to be accessed by their position, indexed reading SHOULD be the only way used to access them. To that end, indexed data SHOULD be stored in arrays.

3.1.5.8.1. Indexable content

name 0x17 (mnemonic: indexable)

shape (indexable {container}:Nat {content}:Bytes)

This form contains in {content} the data that can be indexed somewhere else in the stream, with indexed-bulk (Section 3.1.5.8.2) or indexed-array (Section 3.1.5.8.3). {container} is the identifier of this container and MUST be unique across the stream.

3.1.5.8.2. Indexed BULK expression

name 0x18 (mnemonic: indexed-bulk)

shape (indexed-bulk {container}:Nat {start}:Nat)

The semantics of this form is the same as if it was replaced by the BULK expression in the container whose identifier is {container} (by comparing the value of the unsigned integers, not how they are encoded, but an application SHOULD encode them identically), starting at offset {start}.

3.1.5.8.3. Indexed array

name 0x19 (mnemonic: indexed-array)

shape (indexed-array {container}:Nat {start}:Nat {size}:Nat)

The semantics of this form is an array whose content are {size} bytes, starting at offset {start} in the container whose identifier is {container} (by comparing the value of the unsigned integers, not how they are encoded, but an application SHOULD encode them identically).

Compared to indexed-bulk, which can reference an array, indexed-array is useful when several different but overlapping sections of the same byte sequence are needed as arrays.

3.1.6. Arithmetic

A processing application must recognize the type of all expressions defined in this specification that have the type Number, but an application MAY consider a number as having an unknown value if it has no adequate data type to store it.

In the text notation of a BULK stream, a decimal integer is the notation for the smallest byte sequence that yields this integer as described in Section 2.3.2.4. For example, (31 256) is a notation for the bytes 0x01 0x9F 0xC2-0100 0x02.

3.1.6.1. Unsigned integer

name 0x20 (mnemonic: unsigned-int)

shape (unsigned-int {bits}:Bytes)

The bits contained in {bits} is the value of this integer in binary notation. This form exists in case disambiguation of the semantics is necessary.

Type: Number, Int, Nat.

3.1.6.2. Signed integer

name 0x21 (mnemonic: signed-int)

shape (signed-int {bits}:Bytes)

The bits contained in {bits} is the value of this integer in two’s-complement notation.

Type: Number, Int.

3.1.6.3. Fraction

name 0x22 (mnemonic: frac)

shape (frac {num}:Int {div}:Int)

This is the number {num}/{div}.

Type: Number.

3.1.6.4. Binary floating-point number

name 0x23 (mnemonic: binary-float)

shape (binary-float {bits}:Bytes)

This is a floating-point number expressed in IEEE 754-2008 binary interchange format. {bits} can be of size 16, 32, 64, 128 or any bigger multiple of 32 bits, as per IEEE 754-2008 rules.

Types: Number, Float.

3.1.6.5. Decimal floating-point number

name 0x24 (mnemonic: decimal-float)

shape (decimal-float {bits}:Bytes)

This is a floating-point number expressed in IEEE 754-2008 decimal interchange format. {bits} can be of size 32, 64, 128 or any bigger multiple of 32 bits, as per IEEE 754-2008 rules.

Types: Number, Float.

3.1.6.6. Binary fixed point number

name 0x25 (mnemonic: binary-fixed)

shape (binary-fixed {point}:Nat {bits}:Bytes)

This is a fixed point binary number. {bits} contains an integer in two’s complement. That integer divided by 2^point is the value of this form. For example, (binary-fixed 2 15) has value 3.75_10 (11.11_2).

Types: Number, Float.

3.1.6.7. Decimal fixed point number

name 0x26 (mnemonic: decimal-fixed)

shape (decimal-fixed {point}:Nat {bits}:Bytes)

This is a fixed point decimal number. {bits} contains an integer in two’s complement. That integer divided by 10^point is the value of this form. For example, (decimal-fixed 2 123) has value 1.23.

Types: Number, Float.

3.1.6.8. Decimal fixed point number with 2 decimal places

name 0x27 (mnemonic: decimal2)

value (subst (decimal-fixed 2 (arg 0)))

3.1.7. Compact formats

This specification and other specifications in the official BULK suite take the option to use as their basic building block a form with a distinguishing reference as first element (basically, they are a binary representation of an abstract syntax tree). As noted previously, this means that most representations weigh 4 bytes plus their actual content, which will in turn have some overhead because of one or several marker bytes.

But when there is a special need for compactness, BULK makes it possible to design protocols and formats with different trade-offs, while retaining its property of being parseable by processing applications not knowing the protocol in its entirety.

On one end of the spectrum, a format might choose to use an array to encapsulate an ad hoc binary format. An extreme use of this scheme would be to use BULK just to make explicit the binary format used. With a known profile (for example with a file extension and/or media type for such explicitly typed BLOBs), such a BULK stream can consist solely of the version form, a reference that describes the binary format and an array, which would amount to an overhead between 11 bytes and 20 bytes depending on the size of the content (11, 13, 14, 16 and 20 bytes for contents of no more than 63B, 255B, 65kB, 4GB and 18EB respectively). Without a profile, with the namespaces associations, the overhead is between 28 and 37 bytes (the difference is a single import form to import two namespaces: the one providing a form used to identify namespaces, and the one for the format used in the stream).

Still, even this extreme in the design space retains the ability to insert expressions in the BULK stream, whatever their type. Thus metadata can be added about data that is represented in a format that doesn’t allow for metadata or for limited metadata.

In-between these two extremes, several options are available to produce a format that leverages the BULK parser a lot more while being more compact than a basic BULK format. The following forms provide a standard way to create such formats.

A flat list of operators and operands is called a BULK bytecode. Prefix bytecodes are those where operators come before operands, postfix bytecodes are those where operators come after operands. In the following forms, operators MUST be references.

The default semantics of a bytecode form is to transform it to an abstract syntax tree of its content and then evaluate the resulting expression with the normal BULK evaluation rules. When evaluating a

bytecode form that doesn’t provide arities, a processing application MUST abort this transformation as soon as it encounters a reference for which it cannot determine if it is an operator or its arity. When evaluating a bytecode form that provides arities, any reference that is not known to be an operator MUST be determined to be an operand.

To transform a prefix bytecode form, a processing application creates an alternate context. If the first expression of the bytecode can be determined to be an operand, it is removed from the beginning of the bytecode and appended at the end of the alternate context. If the first expression of the bytecode is a reference that can be determined to be an operator, it is removed from the beginning of the bytecode and a list is created with the operator as the first expression, then as many next expressions as its arity are removed from the beginning of the bytecode and appended at the end of this list. Then that resulting list is appended at the end of the alternate context. The transformation continues until the bytecode is empty, in which case the alternate context replaces the bytecode form and the transformation is complete. The resulting form can then be evaluated in turn.

Example: the default semantics of

```
( prefix* ( ( 2 go:black ) ) go:game go:black 1 2 go:black 3 4
go:black 5 6 )
```

is that it’s transformed into

```
( go:game ( go:black 1 2 ) ( go:black 3 4 ) ( go:black 5 6 ) )
```

To transform a postfix bytecode form, a processing application creates a data stack. If the first expression of the bytecode can be determined to be an operand, it is removed from the beginning of the bytecode and pushed on top of the stack. If the first expression of the bytecode can be determined to be an operator, it is removed from the beginning of the bytecode and a list is created with the operator as the first expression, then as many next expressions as its arity are popped from the stack and appended at the end of this list (with the top of the stack as the last element). Then that resulting list is pushed on top of the stack. The transformation continues until the bytecode is empty, in which case the list of the elements on the stack (with the top of the stack as the last element) replaces the bytecode form and the transformation is complete. The resulting form can then be evaluated in turn.

Example: the default semantics of

```
   ( bulk:postfix*
     ( ( 2 go:black go:white go:comment go:alternative ) )
     go:game
     1 2 go:black
     "white tried an unorthodox opening" 3 4 go:white go:comment
     "a more classical opening would be" 8 9 go:white go:comment
     go:alternative
     2 3 go:black
     4 5 go:white )
```

is that it's transformed into

```
( go:game
  ( go:black 1 2 )
  ( go:alternative
    ( go:comment "white tried an unorthodox opening" ( go:white 3 4 ) )
    ( go:comment "a more classical opening would be" ( go:white 8 9 ) ) )
  ( go:black 2 3 )
  ( go:white 4 5 ) )
```

The obivous advantage of postfix bytecode is that it makes it possible to compact nested forms when they have a known arity. When a reference in a vocabulary can be used in a form containing a variable number of expressions, if some arity is used frequently enough, an application can define a specific form for it. The trade-offs for this are explained in Appendix B

If the overhead of several marker bytes in the operands of some operators is too much, even more compactness can be achieved by packing together small operands. For example, instead of an operator with two integers as its operands, one could specify an operator to take a single word as operand and extract the integers from it (while still retaining the ability to operate on many sizes of integers, because it can still deduce the size of the integers by dividing the size of the word by two).

For example, a BULK format representing player moves with a pair of coordinates on a large board might represent a single move with the following shapes:

basic (8 bytes) (game:black/2 #[1] 0x41 #[1] 0x5A)

packed basic (7 bytes) (game:black/1 #[2] 0x41 0x5A)

bytecode (6 bytes) game:black/2 #[1] 0x41 #[1] 0x5A

packed bytecode (5 bytes) game:black/1 #[2] 0x41 0x5A

The transformation defined for the bytecode forms makes it possible to mix literal expressions and operations represented by a sequence of operators and operands. In the previous scenario, for example, one might represent each alternating move by the two players as two integers, lowering the weight of each move to 2 bytes as coordinates are below 64:

```
( bulk:postfix*
  ( ( 2 go:white go:comment go:alternative ) )
  go:game
  1 2
  "white tried an unorthodox opening" 3 4 go:white go:comment
  "a more classical opening would be" 8 9 go:white go:comment
  go:alternative
  2 3
  4 5 )
```

The difference between all these schemes and an array is that you keep the ability to insert other forms, for example here to represent comments on the game or variants.

The cost of the bytecode format is that if it contains operators whose arity is unknown to a processing application, the whole list after the first occurrence of them is unreadable to that processing application, whereas in the basic format, the processing application can still process all the forms it understands, and that requires no anticipation by the application creating the BULK stream.

3.1.7.1. Prefix bytecode

name 0x30 (mnemonic: prefix)

shape (prefix {bytecode})

This is a prefix bytecode form that doesn’t provide arities.

3.1.7.2. Prefix bytecode with arities

name 0x31 (mnemonic: prefix*)

shape (prefix* {arities}:Expr {bytecode})

This is a prefix bytecode form that provides arities.

{arities} MUST be a list of shapes ({arity}:Nat {refs}). {refs} MUST be a series of references. It indicates that all references in this series are operators of arity {arity}. {arities} can be a form or a reference defined to a list.

Within the prefix bytecode of this form, if there is a prefix form, the arities declared in the outside form still apply.

3.1.7.3. Postfix bytecode

name 0x32 (mnemonic: postfix)

shape (postfix {bytecode})

This is a postfix bytecode form that doesn’t provide arities.

3.1.7.4. Postfix bytecode with arities

name 0x33 (mnemonic: postfix*)

shape (postfix* {arities}:Expr {bytecode})

This is a postfix bytecode form that provides arities.

{arities} MUST be a list of shapes ({arity}:Nat {refs}). {refs} MUST be a series of references. It indicates that all references in this series are operators of arity {arity}. {arities} can be a form or a reference defined to a list.

Within the postfix bytecode of this form, if there is a postfix form, the arities declared in the outside form still apply.

3.1.7.5. Arity declaration

name 0x34 (mnemonic: arity)

shape (arity {arity}:Nat {refs})

{refs} MUST be a series of references. It indicates that all references in this series are operators of arity {arity}.

Whenever arities have been provided by this form for some references in a namespace, all references in that namespace whose arities aren’t provided MUST be determined to be operands by a processing application.

4. Extension namespaces

Extension namespaces are defined with a unique identifier, to be associated to a marker value.

By is decentralized nature, as far as a processing application is concerned, apart from standard namespaces, there is no difference between a namespace defined as part of the official BULK suite and a user-defined one.

5. Profiles

A profile is a byte sequence parsed by a processing application just after the version form or before the first expression if there is no version form. Thus a parser SHOULD look ahead at the beginning of a stream to see if the first three bytes are (bulk:version. With respect to the BULK stream, the profile is an out-of-band information, usually implicit.

A processing application doesn’t need to include the profile in the concrete yield, as long as the semantics of the abstract yield are maintained.

The same BULK stream might be processed with different profiles.

A processing application MUST NOT deduce the profile from the content of a BULK stream.

5.1. Profile redundancy

A processing application SHOULD only rely on the use of a profile when it is a safe assumption that the profile is known, for example within a communication where the protocol dictates the profile.

In particular, long-term storage of a BULK stream SHOULD preserve profile information, for example with a media type that dictates the profile.

Otherwise, an application writing a BULK stream in a long-term storage SHOULD include the profile after the version form. For this reason, the expressions in a profile SHOULD have idempotent semantics.

5.2. Standard profile

This specification defines the default profile that a processing application MUST use when it is not using a specific profile:

```
( bulk:stringenc ( bulk:iana-charset 106 ) )
```

This means that the default string encoding in a BULK stream is UTF-8.

6. Security Considerations

6.1. Parsing

Parsing a BULK stream is designed to be free of side-effects for the processing application, apart from storing the parsed results.

Arrays in BULK carry their size, so as for the application to know in advance the size of the data to read and store, thus making it easier to build robust code. A malicious software, however, may announce an array with a size choosen to get an application to exhaust its available memory. When a BULK stream has been completely received, an array bigger than the remaining data SHOULD trigger an error. When a BULK stream's size is not known in advance, the application SHOULD use a growable data structure.

Evaluation opens up some known attacks that appear whenever a format provides a way to express abstraction, like the billion laughs attack. As it is explained in Evaluation, an implementation MAY stop evaluation after a predefined number of evaluation steps. As this has been demonstrated not to be sufficient to prevent attacks based on expansion, an implementation SHOULD also put predefined limits on the space that the abstract yield can take on disk or in memory.

Applications MAY use out-of-band information to select size limits (like HTTP attributes), or a BULK namespace MAY provide hints. An implementation SHOULD emit warnings when the size of the abstract yield would exceed the size limits set by such out-of-band or in-band information.

6.2. Forwarding

When a processing application forwards all or part of the data in a BULK stream to another application, care must be taken if part of the forwarded data was not entirely recognized, as it could be used by an attacker to benefit from the authority the forwarding application has on the recipient of the data.

6.3. Definitions

The architecture of a processing application SHOULD ensure that a malicious agent cannot abuse authority given to it to define a namespace in order to modify associations in other namespaces. Depending on the use of data structures storing BULK expressions, this could amount to giving an attacker a way to manipulate the application's state. See Appendix A for an example of architecture that is resistant to that kind of attack.

7. IANA Considerations

This specification defines a new media type, application/bulk. Here are the informations for its registration to IANA:

Type name application

Subtype name bulk

Required parameters none

Optional parameters none

Encoding considerations none, content is self-describing

Security considerations cf. Section 6

Interoperability considerations the constraint to start any BULK file with a version form has the side-effect that classes of BULK streams can be identified by a sequence of bytes acting as "magic number", at offset 0:

0x011000 any BULK stream

0x01100081 a BULK stream of major version 1

0x011000818002 a BULK stream of version 1.0

Published specification this document

Applications that use this media type none so far

Fragment identifier considerations this specification defines no semantics for addressing the data with a fragment identifier; a future specification MAY define fragment identifier syntaxes to address the content by byte offset or the parsed results by their position in the yielded list

Additional information a future specification MAY define a naming convention for media types based on bulk with a +bulk suffix, as for XML with +xml

8. Acknowledgements

The original author of this specification read Erik Naggum's famous rant about XML (http://www.schnada.de/grapt/eriknaggum-xmlrant.html) several years before, and while forgotten as such for a time, it clearly was the seed that slowly bloomed into the design of BULK. This format is dedicated to Erik.

9. References

9.1. Normative References

[IANA-Charsets] "IANA Charset Registry (archived at):", <http://www.iana.org/assignments/character-sets>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

9.2. Informative references

[Avro] Cutting, D., "Apache Avro 1.7.4 Specification", February 2013, <http://avro.apache.org/docs/1.7.4/spec.html>.

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol version 2 (HTTP/2)", RFC 7540, May 2015, <https://www.rfc-editor.org/rfc/rfc7540>.

[protobuf] "Protocol Buffers", July 2008, <https://developers.google.com/protocol-buffers/>.

[Smile] Saloranta, T., "Smile Data Format", September 2010, <http://wiki.fasterxml.com/SmileFormat>.

[Thrift] Slee, M., Agarwal, A., and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation", April 2007, <http://thrift.apache.org/static/files/thrift-20070401.pdf>.

Appendix A. Robust namespace definition

This constitutes a suggestion of architecture for a BULK processing application. It has the advantage that an agent cannot modify the values of names to which it has not specifically been given authority. This architecture doesn't ensure this property by checking the validity of definitions but by adhering to the Principle Of Least Authority, thus ensuring no false positives or TOCTOU race

conditions.

For each new context (including the abstract yield when parsing starts), the parser creates a new copy of each known namespace. These copies are available in this context to retrieve and define values. It implements the lexical scoping of definitions on top of providing the robustness properties discussed here.

By default, all namespaces created in a context are discarded at the end of this context.

Of course, an implementation of the architecture presented here can be optimized compared to the abstract algorithm, for example by using copy-on-demand.

Any namespace that is not a copy for its context but the object retained by the application afterwards, gives authority to make long-lasting definitions. Such a namespace is called lasting here.

A.1. Selective authority

A number of lasting namespaces are included for the abstract yield. Their unique identifiers are agreed out-of-band. The disadvantage of this solution is that it needs prior agreement on the definable namespaces.

A.2. Open authority

Any ns form for a unique identifier unknown to the processing application triggers the creation of a lasting namespace.

The disadvantage of this solution is that it opens a denial of service vulnerability. If Bob is a processing application and Carol and Dave are agents communicating with Bob with an open authority, Dave can prevent Carol from defining a namespace if it manages to know the unique identifier and starting a communication with Bob before Carol.

If an agent uses a secure way to create unique identifiers, this solution is both flexible and safe (the burden is not on the BULK processing application).

Appendix B. Arity-carrying forms

Sometimes a vocabulary will include forms that can contain an arbitrary number of expressions. When such a form is used in postfix bytecode, the simplest solution is just to use a nested postfix form:

```
  ( bulk:postfix*
    ( ( 2 go:black go:white go:comment ) )
    go:game
    1 2 go:black
    ( bulk:postfix go:alternative
      "white tried an unorthodox opening" 3 4 go:white go:comment
      "a more classical opening would be" 8 9 go:white go:comment )
    2 3 go:black
    ( bulk:postfix go:alternative
      "white played a bad move" 4 5 go:white go:comment
      "white could have played a decent move" 5 6 go:white go:comment
      "white could have played a great move" 5 7 go:white go:comment ) )
```

The nested postfix form costs 4 bytes, compared to an equivalent postfix bytecode.

If those 4 bytes add up to too much space through repetition, an application could define a form for the sole purpose of assigning it an arity, while the evaluation of the arity-carrying form would just replace it with the original one. For example, after evaluating the postfix bytecode transformation and the resulting form of the last expression of

```
( bulk:ns-mnemonic 0x1400 "go2" )
( bulk:mnemonic/def 0x1401 "alt/2" nil ( bulk:subst ( alternative ( rest 0 ) ) ) )
( bulk:mnemonic/def 0x1401 "alt/3" nil ( bulk:subst ( alternative ( rest 0 ) ) ) )

( bulk:postfix*
  ( ( 2 go:black go:white go:comment go2:alt/2 ) ( 3 go2:alt/3 ) )
  go:game
  1 2 go:black
  "white tried an unorthodox opening" 3 4 go:white go:comment
  "a more classical opening would be" 8 9 go:white go:comment
  go2:alt/2
  2 3 go:black
  "white played a bad move" 4 5 go:white go:comment
  "white could have played a decent move" 5 6 go:white go:comment
  "white could have played a great move" 5 7 go:white go:comment
  go2:alt/3
  )
```

it would be transformed into

```
( go:game
  ( go:black 1 2 )
  ( go:alternative
    ( go:comment "white tried an unorthodox opening" ( go:white 3 4 ) )
    ( go:comment "a more classical opening would be" ( go:white 8 9 ) ) )
  ( go:black 2 3 )
  ( go:alternative
    ( go:comment "white played a bad move" ( go:white 4 5 ) )
    ( go:comment "white could have played a decent move" ( go:white 5 6 ) )
    ( go:comment "white could have played a great move" ( go:white 5 7 ) ) )
  ( go:white 4 5 ) )decent
```

Without the mnemonics, such an arity-carrying form basically costs 24 or 27 bytes to be usable. Which means that compared to the nested postfix form, it pays for itself if it is used at least 7 times.

Author’s Address

Pierre Thierry
Comonad Dev
Email: pierre@comonad.dev