Binary Data

Release 0.1

Dylan Hackers

December 15, 2018

The binary data library provides a domain-specific language for manipulation of binary data, or structured byte sequences, as they appear in everyday software such as networking tools or graphics file manipulation. The binary data domain-specific language uses the metaprogramming features of Dylan to a large extent.

The design goals are manifold: concise expressive syntax, efficient conversion between byte vectors and high level objects (in both directions, by using zerocopy and lazy parsing technology). The source code of this library is available under a MIT license at GitHub. A large body of implemented binary data formats using this library is also available at GitHub.

Inspiration for this library is taken among others from the defstorage system available as part of the Genera Common Lisp operating system and the swiss-army knife for interactive packet manipulation scapy.

For further information, you might want to read our published papers about a TCP/IP stack written entirely in Dylan:

- A domain-specific language for manipulation of binary data in Dylan (by Hannes Mehnert and Andreas Bogk at ILC 2007)
- Secure Networking (by Andreas Bogk and Hannes Mehnert in 2006)

CHAPTER

ONE

USAGE

Terminology

A vector of bytes that has an associated definition for its interpretation is a *frame*. These come in two variants: some cannot be broken down further structurally, we call those *leaf frames*. The others have a composite structure, those are *container frames*. They consist of a number of *fields*, which are the named components of that frame. Every field in a container frame is a frame in itself, leading to a recursive definition of frames. The description of the structure of a container frame in our domain-specific language `define binary-data` is referred to as a *binary data definition*.

Representation in Dylan

The binary-data library provides an extension to Dylan for manipulating frames, with a representation of frames as Dylan objects, and a set of functions on these objects to perform the manipulation. The representation used introduces a class hierarchy rooted at the abstract superclass `<frame>`, with the two disjoint abstract subclasses `<leaf-frame>` and `<container-frame>`. Every type of frame in the system is represented as a concrete subclass of either one, and actual frames are instances of these classes. A pair of generic functions, `parse-frame` and `assemble-frame`, convert a given byte vector into the appropriate high-level instance of `<frame>`, or vice versa.

Typical code that handles a frame then looks like this:

```
let frame = parse-frame(<ethernet-frame>, some-byte-vector);
format-out("This packet goes from %= to %=\n",
           frame.source-address,
           frame.destination-address);
```

The first line binds the variable frame to an instance of some subclass of <ethernet-frame>. This instance is created from the vector of bytes passed to the call of *parse-frame*. Then, the value of the source and destination address fields in the Ethernet frame are extracted and printed.

The class *<frame>* defines several generic functions:

- *parse-frame* instantiates a *<frame>* with the value taken from a given byte-vector
- *assemble-frame* encodes a *<frame>* instance into its byte-vector
- *frame-size* returns the size (in bit) of the given frame
- *summary* prints a human-readable summary of the given frame

Some properties are mixed in into our class hierarchy by introducing the direct subclasses of *<frame>*:

For efficiency reasons, there is a distinction between frames that have a static (compile-time) size (*<fixed-size-frame>*) and frames of dynamic size (*<variable-size-frame>*).

Another property is translation of the value into a Dylan object of the standard library. An example of such a *<translated-frame>* is the (fixed size) type <2byte-big-endian-unsigned-integer> which is translated into a Dylan <integer>. This is referred to as a *translated frame* while frames without a matching Dylan type are known as *untranslated frames* (*<untranslated-frame>*).

The appropriate classes and accessor functions are not written directly for container frames. Rather, they are created by invocation of the macro *define binary-data*. This serves two purposes: it allows a more compact representation, eliminating the need to write boilerplate code over and over again, and it hides implementation details from the user of the DSL.

Frame Types

Leaf Frames

A leaf frame can be fixed or variable size, and translated or untranslated. Examples are:

- *<raw-frame>* has a variable size and no translation
- *<null-frame>* has a zero size and no translation
- *<fixed-size-byte-vector-frame>* (e.g. an IPv4 address) has a fixed size and no translation
- <2byte-big-endian-unsigned-integer> has a fixed size of 16 bits, and its translation is a Dylan <integer>.

FIXME: *<externally-delimited-string>* is variable size and untranslated, though as in both directions with <string> is provided (should inherit from translated frame)

The generic function *read-frame* is used to convert a <string> into an instance of a <*leaf-frame*>.

FIXME: why is read-frame not defined on container-frame?

The running example in this guide will be an <ethernet-frame>, which contains the mac address of the source and a mac-address of the destination. A mac address is the unique address of each network interface, assigned by the IEEE. It consists of 6 bytes and is usually printed in hexadecimal, each byte separated by :.

The definition of the <mac-address> class in Dylan is:

```
define class <mac-address> (<fixed-size-byte-vector-frame>)
end;

define inline method field-size (type == <mac-address>)
 => (length :: <integer>)
  6 * 8
end;
```

```
define method mac-address (data :: <byte-vector>)
 => (res :: <mac-address>)
  parse-frame(<mac-address>, data)
end;

define method mac-address (data :: <string>)
 => (res :: <mac-address>)
  read-frame(<mac-address>, data)
end;

define method read-frame (type == <mac-address>, string :: <string>)
 => (res :: <mac-address>)
  let res = as-lowercase(string);
  if (any?(method(x) x = ':' end, res))
    //input: 00:de:ad:be:ef:00
    let fields = split(res, ':');
    unless(fields.size = 6)
     signal(make(<parse-error>))
    end;
    make(<mac-address>,
         data: map-as(<stretchy-vector-subsequence>,
                      rcurry(string-to-integer, base: 16),
                      fields))
  else
    //input: 00deadbeef00
    ...
  end;
end;

define method as (class == <string>, frame :: <mac-address>)
 => (string :: <string>)
  reduce1(method(a, b) concatenate(a, ":", b) end,
          map-as(<stretchy-vector>,
                 rcurry(integer-to-string, base: 16, size: 2),
                 frame.data))
end;
```

The data is stored in the `data` slot of the *<fixed-size-byte-vector-frame>*, the `field-size` method returns statically 48 bit, syntax sugar for constructing `<mac-address>` instances are provided, `read-frame` converts a `<string>`, whereas `as` converts a `<mac-address>` into human readable output.

A leaf frame on its own is not very useful, but it is the building block for the composed container frames.

Container Frame

The container frame class inherits from *<variable-size-frame>* and *<untranslated-frame>*.

A container frame consists of a sequence of fields. A field represents the static information about a protocol: the name of the field, the frame type, possibly a start and length offset, a length, a method for fixing the byte vector, ...

The list of fields for a given *<container-frame>* persists only once in memory, the dynamic values are represented by `<frame-field>` objects.

Methods defined on *<container-frame>*:

- *fields* returns the list of *<field>* instances
- `field-count` returns the size of the list
- *frame-name* returns a short identifier of the frame

The definer macro *define binary-data* translates the binary-data DSL into a class definition which is a subclass of *<container-frame>* (and other useful stuff).

The class *<header-frame>* is a direct subclass of *<container-frame>* which is used for container frames which consist of a header (addressing, etc) and some payload, which might also be a container-frame of variable type.

The running example is an <ethernet-frame>, which is shown as binary data definition.

```
define binary-data <ethernet-frame> (<header-frame>)
  summary "ETH %= -> %=", source-address, destination-address;
  field destination-address :: <mac-address>;
  field source-address :: <mac-address>;
  layering field type-code :: <2byte-big-endian-unsigned-integer>;
  variably-typed field payload, type-function: frame.payload-type;
end;
```

The first line specifies the name <ethernet-frame>, and its superclass, *<header-frame>*.

The second line specialises the method *summary* on an <ethernet-frame> to print ETH, the source address and the destination address.

The remaining lines represent each one field in the ethernet frame structure. The source-address and destination-address are each of type <mac-address>.

The type-code field is a 16 bit integer, and it is a layering field (*<layering-field>*). This means that its value is used to determine the type of its payload! Also, when assembling such a frame, the layering field will be filled out automatically depending on the payload type. There can be at most one layering field in a binary data definition.

The last field is the payload, whose type is variable and given by applying the function payload-type to the concrete frame instance. The default type-function of a *<variably-typed-field>* is *payload-type*.

A payload for an <ethernet-frame> might be a <vlan-tag>, if the type-code is #x8100 (the over keyword takes care of the hairy details).

```
define binary-data <vlan-tag> (<header-frame>)
  over <ethernet-frame> #x8100;
  summary "VLAN: %=", vlan-id;
  field priority :: <3bit-unsigned-integer> = 0;
  field canonical-format-indicator :: <1bit-unsigned-integer> = 0;
  field vlan-id :: <12bit-unsigned-integer>;
  layering field type-code :: <2byte-big-endian-unsigned-integer>;
  variably-typed field payload, type-function: frame.payload-type;
end;
```

Default values for fields can be provided, similar to Dylan class definitions, using the equals sign (=) after the field type.

A more detailed description of the binary data language can be found in its reference *define binary-data*.

Inheritance: Variably Typed Container Frames

A container frame can inherit from another container frame that already has some fields defined. The *<variably-typed-container-frame>* class is used in container frames which have the type information encoded in the frame. The layering field (*<layering-field>*) of such container frames must be parsed in order to determine the actual type.

Continuing with the <ethernet-frame> example, consider the options of an IPv4 packet. These share a common header (copy-flag and option-type), but a concrete option might have additional fields. The end of the

options list is determined by the `header-length` field of an IPv4 packet and by the `<end-option>` (whose `option-type` is 0).

```
define abstract binary-data <ip-option-frame> (<variably-typed-container-frame>)
  field copy-flag :: <1bit-unsigned-integer>;
  layering field option-type :: <7bit-unsigned-integer>;
end;

define binary-data <end-option> (<ip-option-frame>)
  over <ip-option-frame> 0;
end;

define binary-data <router-alert> (<ip-option-frame>)
  over <ip-option-frame> 20;
  field router-alert-length :: <unsigned-byte> = 4;
  field router-alert-value :: <2byte-big-endian-unsigned-integer>;
end;
```

This defines the `<end-option>` which has the `option-type` field in the ip-option frame set to `0`. An `<end-option>` does not contain any further fields, thus only has the two fields inherited from the `<ip-option-frame>`.

The `<router-alert>` specifies two more fields, which are appended to the inherited fields.

Fields

The domain-specific language *`define binary-data`* provides syntactic sugar to create *`<field>`* instances. A client should not need to instantiate these directly. A field contains the static information (such as type, length, default value) of a sequence of bits inside of a *`<container-frame>`*.

Binary data formats have some common patterns which are directly integrated into this library:

- *variably-typed* fields for payloads
- *layering* of protocols in the OSI network stack
- *enumeration* where the bit value has a direct correspondence to a `<symbol>`
- *repeating* occurences of a field, such as key-value pairs

Note: There might be more patterns, if you find any, please tell us!

Variably-typed

Most fields have the same type in all frame instances, i.e. they are statically typed. In some cases however, the type of a field can depend on the value of another field in the same *`<container-frame>`*. Such fields can be defined using *`<variably-typed-field>`* which does not have a static type, but an expression determining the field type for a concrete frame instance.

This example uses the `variably-typed field` syntax. The `type-function` keyword has `frame` bound to the concrete frame object.

```
field length-type :: <2bit-unsigned-integer>;
variably-typed field body-length,
  type-function: select (frame.length-type)
                   0 => <unsigned-byte>;
```

```
                    1 => <2byte-big-endian-unsigned-integer>;
                    2 => <4byte-big-endian-unsigned-integer>;
                    3 => <null-frame>;
                  end;
```

Note that whenever the actual type of a variably-typed field resolves to the <null-frame> type it means that the field is completely missing from the container frame.

Layering

Binary data format stacking is omnipresent in network protocols. An ethernet frame can contain different types of payload, amongst others ARP frames, IPv4 frames. This library provides syntactic sugar `layering` to define which field in a frame determines the type of the payload. A binary data definition can also specify which value is used to be the payload of another binary data format.

A layering field (*<layering-field>*) provides the information that the value of this field controls the type of the payload, and establishes a registry for field values and matching payload types.

The registry can be extended with the `over` syntax of *define binary-data*, and it can be queried using the convinience function *payload-type*, or *lookup-layer* and *reverse-lookup-layer*.

Enumeration

An enumerated field (*<enum-field>*) provides a set of mappings from the binary value to a Dylan `<symbol>`. Note that the binary value must be a numerical type so that the mapping is from an integer to a symbol.

In this example, accessing the value of the field would return one of the symbols rather than the value of the *<unsigned-byte>*. For mappings not specified, the integer value is used:

```
enum field command :: <unsigned-byte> = 0,
    mappings: { 1 <=> #"connect",
                2 <=> #"bind",
                3 <=> #"udp associate" };
```

Repeating

Repeated fields (*<repeated-field>*) have a list of values of the field type, instead of just a single one. Currently two kinds of repeated fields are supported, *<self-delimited-repeated-field>* and *<count-repeated-field>*, they only differ in the way the the number of elements in the repeated field is determined.

A self-delimited field definition uses an expression to evaluate whether or not the end of the list of values has been reached, usually by checking for a magic value. This expression should return `#t` when the field is fully parsed. For example:

```
repeated field options :: <ip-option-frame>,
  reached-end?:
    instance?(frame, <end-option>);
```

A count field definition uses another field in the frame to determine how many elements are in the field. For example:

```
field number-methods :: <unsigned-byte>,
  fixup: frame.methods.size;
repeated field methods :: <unsigned-byte>,
  count: frame.number-methods;
```

Note the use of the `fixup` keyword on the `number-methods` field to calculate a value for use by *assemble-frame* if the value is not otherwise specified.

Adding a New Leaf Frame Type

Depending on the properties of the frame, there are different methods which should be specialized. In general, there need to be a specialization of the size, how to parse, and how to assemble the frame.

There are two generic functions which should be specialized by every *<leaf-frame>* subclass: *parse-frame* and *read-frame*.

Note: there should be a `print-frame` as well, rather than using `as(<string>, frame)`.

Fixed size frames must specialize *field-size*, variable sized ones *frame-size*.

Translated frames must specialize *high-level-type* and *assemble-frame-into-as*.

Untranslated frames must specialize *assemble-frame-into*.

There are already several classes and macros implemented where these methods are defined.

See also

- *<leaf-frame>*
- *<fixed-size-translated-leaf-frame>*
- *<variable-size-translated-leaf-frame>*
- *<fixed-size-untranslated-leaf-frame>*
- *<variable-size-untranslated-leaf-frame>*
- *define n-byte-vector*
- *define n-bit-unsigned-integer*
- *define n-byte-unsigned-integer*
- *<unsigned-integer-bit-frame>*
- *<variable-size-byte-vector>*
- *<externally-delimited-string>*
- *<fixed-size-byte-vector-frame>*
- *<big-endian-unsigned-integer-byte-frame>*
- *<little-endian-unsigned-integer-byte-frame>*

Efficiency Considerations

The design goal of this library is, as usual in object-centered programming, that the time and space overhead are minimal (the compiler should remove all the indirections!).

This library is carefully designed to achieve this goal, while not limiting the expressiveness, sacrificing the safety, or burdening the developer with inconvenient syntactic noise. A story about binary data is that there are often big chunks of data, and deeply nested pieces of data. The good news is that most applications do not need all binary data.

The binary data library was designed with lazy parsing in mind: if a byte vector is received, the high-level object does not parse the byte vector completely, but only the requested fields. To achieve this, we gather information about each field, specifically its start and end offset, and also its length, already at compile time, using a number system consisting of the type union between `<integer>` and `$unknown-at-compile-time`, for which basic arithmetics is defined.

For fixed sized fields, meaning single fields with a static and fixed size frame type, their length is propagated while the DSL iterates over the fields. All field offsets for the `<ethernet-frame>` are known at compile time. Accessing the `payload` is a subsequence operation (performing zerocopy) starting at bit 112 (or byte 15) of the binary vector.

While at the user level arithmetics is on the bit level, accesses at byte boundaries are done directly into the byte vector. This is encapsulated in the class *`<stretchy-byte-vector-subsequence>`*

FIXME: move <stretchy-byte-vector-subsequence> to a separate module

Each binary data macro call defines a container class with two direct subclasses, a high-level decoded class (`<decoded-container-frame>`) and a partially parsed one with an attached byte-vector (`<unparsed-container-frame>`). The decoded class has a list of `<frame-field>` instances, which contain the metadata (size, fixup function, reference to the field, etc.) of each field. The partially parsed class reuses this class in its `cache` slot, and keeps a reference to its byte vector in another slot.

CHAPTER

TWO

API REFERENCE

Overview

This describes the API available from binary-data. It is organized in the following sections, depending on different demands: using binary data in a tool, extending with a custom binary format, and the internal API.

Class hierarchy

The class hierarchy is rooted in `<frame>`. Several direct subclasses exist, some are `open` and may be subclassed. Only those combinations of direct subclasses which were needed until now are defined (there might be need for other combinations in the future).

`<frame>` Abstract Class

Discussion The abstract superclass of all frames, several generic functions are defined on this class.

Superclasses `<object>`

Operations

- *parse-frame*
- *assemble-frame*
- *summary*

`<leaf-frame>` Abstract Class

Discussion The abstract superclass of all frames without any further structure.

Superclasses *<frame>*

Operations

- *read-frame*

`<fixed-size-frame>` Abstract Class

Discussion The abstract superclass of all frames with a static length. The specialization of *frame-size* calls *field-size* on the object class of the given instance.

Superclasses *<frame>*

`<variable-size-frame>` Abstract Class

Discussion The abstract superclass of all frames with a variable length.

Superclasses *<frame>*

<translated-frame> Abstract Class

Discussion The abstract superclass of all frames with a conversion into a native Dylan type.

Superclasses *<frame>*

<untranslated-frame> Abstract Class

Discussion Abstract superclass of all frames with a custom class instance.

Superclasses *<frame>*

<fixed-size-untranslated-frame> Abstract Class

Discussion Abstract superclass for fixed sized frames without a translation

Superclasses *<fixed-size-frame>*, *<untranslated-frame>*

<variable-size-untranslated-frame> Abstract Class

Discussion Abstract superclass for variable sized frames without a translation. This is the direct superclass of *<container-frame>*.

Superclasses *<variable-size-frame>*, *<untranslated-frame>*

<fixed-size-translated-leaf-frame> Open Abstract Class

Discussion Superclass of all fixed size leaf frames with a translation, mainly used for bit vectors represented as Dylan <integer>

Superclasses *<leaf-frame>*, *<fixed-size-frame>*, *<translated-frame>*

<variable-size-translated-leaf-frame> Open Abstract Class

Discussion Superclass of all variable size leaf frames with a translation (currently unused)

Superclasses *<leaf-frame>*, *<variable-size-frame>*, *<translated-frame>*

<fixed-size-untranslated-leaf-frame> Open Abstract Class

Discussion Superclass of all fixed size leaf frames without a translation, mainly used for byte vectors (IP addresses, MAC address, ...), see its subclass *<fixed-size-byte-vector-frame>*.

Superclasses *<leaf-frame>*, *<fixed-size-untranslated-frame>*

<variable-size-untranslated-leaf-frame> Open Abstract Class

Discussion Superclass of all variable size leaf frames without a translation (for example class *<raw-frame>* and class *<externally-delimited-string>*)

Superclasses *<leaf-frame>*, *<variable-size-untranslated-frame>*

<null-frame> Class

Discussion A concrete zero size leaf frame without a translation. This frame type can be used as one of the types of a variably-typed field to make the field optional. A field with a type <null-frame> is considered to be missing from the container frame. Conversion of a <null-frame> to string or vice versa is not supported (because it wouldn't make much sense).

Superclasses *<fixed-size-untranslated-leaf-frame>*

<container-frame> Open Abstract Class

Superclass of all binary data definitions using the *define binary-data* macro.

Superclasses *<variable-size-untranslated-frame>*

Operations

- *frame-name*

- *fields*
- field-count
- *packet*

<header-frame> Open Abstract Class
Superclass of all binary data definitions which support layering, thus have a header and payload.

Superclasses *<container-frame>*

Discussion The method payload projects the payload of the header frame. The method payload-setter is also defined. The specialized method *fixup!* calls *fixup!* on the payload of the header frame instance.

Operations

- payload
- payload-setter
- *fixup!*

<variably-typed-container-frame> Open Abstract Class
Superclass of all binary data definitions which have an abstract header followed by more fields. In the header a specific *<layering-field>* determines which subclass to instantiate.

Superclasses *<container-frame>*

Tool API

Parsing Frames

parse-frame Open Generic function
Parses the given binary packet as frame-type, resulting in an instance of the frame-type and the number of consumed bits.

Signature parse-frame *frame-type packet* #rest *rest* #key #all-keys => *result consumed-bits*

Parameters

- **frame-type** – Any subclass of <frame>.
- **packet** – The byte vector as <sequence>.
- **rest** (*#rest*) – An instance of <object>.

Values

- **result** – An instance of the given frame-type.
- **consumed-bits** – The number of bits consumed as <integer>

read-frame Open Generic function
Converts a given string to an instance of the given leaf frame type.

Signature read-frame *frame-type string* => *frame*

Parameters

- **frame-type** – An instance of subclass(<leaf-frame>).
- **string** – An instance of <string>.

Values

- **frame** – An instance of <object>.

Assembling Frames

assemble-frame Generic function

Produces a binary vector representing this frame. All field fixup functions are called.

Signature assemble-frame *frame* => *packet*

Parameters

- **frame** – An instance of *<frame>*.

Values

- **packet** – An instance of <object>.

Information about Frames

frame-size Open Generic function

Returns the length in bits for the given frame.

Signature frame-size *frame* => *length*

Parameters

- **frame** – An instance of <frame>.

Values

- **length** – The size in bits, an instance of <integer>.

summary Open Generic function

Returns a human-readable customizable (in binary-data-definer) string, which summarizes the frame.

Signature summary *frame* => *summary*

Parameters

- **frame** – An instance of *<frame>*.

Values

- **summary** – An instance of <string>.

packet Open Generic function

Underlying byte vector of the given *<container-frame>*.

Signature packet *frame* => *byte-vector*

Parameters

- **frame** – An instance of *<container-frame>*.

Values

- **byte-vector** – An instance of <byte-sequence>.

parent Sealed Generic function

If the frame is a payload of another layer, returns the frame of the upper layer, false otherwise.

Signature parent *frame* => *parent-frame*

Parameters

- **frame** – An instance of *<container-frame>* or <variable-size-byte-vector-frame>

Values

- **parent-frame** – Either the *<container-frame>* of the upper layer or #f

Information about Frame Types

fields Open Generic function

Returns a vector of *<field>* for the given *<container-frame>*

Signature fields *frame-type* => *fields*

Parameters

- **frame-type** – Any subclass of *<container-frame>*.

Values

- **fields** – A <simple-vector> containing all fields.

Note: Current API also allows instances of <container-frame>, should be revised

frame-name Open Generic function

Returns the name of the frame type.

Signature frame-name *frame-type* => *name*

Parameters

- **frame-type** – Any subclass of *<container-frame>*.

Values

- **name** – A <string> with the human-readable frame name.

Note: Current API also allows instances of <container-frame>, should be revised

Fields

Syntactic sugar in the *define binary-data* domain-specific language instantiates these fields.

<field> Abstract Class

The abstract superclass of all fields.

Superclasses <object>

Init-Keywords

- **name** – The name of this field.
- **fixup** – A unary Dylan function computing the value of this field, used if no default is supplied and none provided by the client, defaults to #f.
- **init-value** – The default value if the client did not provide any, default $unsupplied.

- **static-end** – A Dylan expression determining the end, defaults to `$unknown-at-compile-time`.
- **static-length** – A Dylan expression determining the length, defaults to `$unknown-at-compile-time`.
- **static-start** – A Dylan expression determining the start, defaults to `$unknown-at-compile-time`.
- **dynamic-end** – A unary Dylan function computing the end, defaults to `#f`.
- **dynamic-length** – A unary Dylan function computing the length, defaults to `#f`.
- **dynamic-start** – A unary Dylan function computing the start, defaults to `#f`.
- **getter** – The getter method to extract this fields value out of a concrete frame.
- **setter** – The setter method to set this fields to a concrete value in a concrete frame.
- **index** – An *<integer>* which is an index of this field in its *<container-frame>*.

Discussion All keyword arguments correspond to a slot, which can be accessed.

Operations

- `field-name(<field>)`
- `fixup-function(<field>)`
- `init-value(<field>)`
- `static-start(<field>)`
- `static-length(<field>)`
- `static-end(<field>)`
- `getter(<field>)`
- `setter(<field>)`

See also

- *define binary-data*
- *fields*

<variably-typed-field> Class

The class for fields of dynamic type.

Superclasses *<field>*

Init-Keywords

- **type-function** – A unary Dylan function computing the type of the field, defaults to *payload-type*.

See also

- *payload-type*
- *lookup-layer*
- *reverse-lookup-layer*

<statically-typed-field> Abstract Class

The abstract superclass of all statically typed fields.

Superclasses *<field>*

Init-Keywords

- **type** – The static type, a subclass of *<frame>*.

Operations

- `type(<statically-typed-field>)`

Note: restrict type in source code!

<single-field> Class
The common field. Nothing interesting going on here.

Superclasses *<statically-typed-field>*

<enum-field> Class
An enumeration field to map `<integer>` to `<symbol>`.

Superclasses *<single-field>*

Init-Keywords

- **mapping** – A mapping from keys to values as `<collection>`.

<layering-field> Class
The layering field is used in *<header-frame>* and *<variably-typed-container-frame>* to determine the concrete type of the payload or which subclass to use.

Superclasses *<single-field>*

Discussion

The `fixup-function` slot is bound to use the available layering information. No need to specify a fixup.

<repeated-field> Abstract Class
Abstract superclass of repeated fields. The `init-value` slot is bound to `#()`.

Superclasses *<statically-typed-field>*

<count-repeated-field> Class
A repeated field whose number of repetitions is determined externally.

Superclasses *<repeated-field>*

Init-Keywords

- **count** – A unary function returning the number of occurences.

<self-delimited-repeated-field> Class
A repeated field whose end is determined internally.

Superclasses *<repeated-field>*

Init-Keywords

- **reached-end?** – A unary function returning a `<boolean>`.

Layering of frames

payload-type Function
The type of the payload, It is just a wrapper around *lookup-layer*, which returns *<raw-frame>* if `lookup-layer` returned false.

Signature payload-type *frame* => *payload-type*

Parameters

- **frame** – An instance of *<container-frame>*.

Values

- **payload-type** – An instance of `<type>`.

`lookup-layer` Open Generic function

Given a *frame-type* and a *key*, returns the type of the payload.

Signature lookup-layer *frame-type key* => *payload-type*

Parameters

- **frame-type** – Any subclass of *<frame>*.
- **key** – Any `<integer>`.

Values

- **payload-type** – The resulting type, an instance of `false-or(<class>)`.

`reverse-lookup-layer` Open Generic function

Given a frame type and a payload, returns the value for the layering field.

Signature reverse-lookup-layer *frame-type payload* => *layering-value*

Parameters

- **frame-type** – Any subclass of *<frame>*.
- **payload** – Any *<frame>* instance.

Values

- **value** – The returned layering field value, an `<integer>`.

Note: Check whether it can work with other types than integers

---

Database of Binary Data Formats

Note: Rename to `$binary-data-registry` or similar. Also, narrow types for the functions in this section.

`$protocols` Constant

A hash table with all defined binary formats. Insertion is done by a call of *define binary-data*.

Type `<table>`

Value Mapping of `<symbol>` to subclasses of *<container-frame>*.

`find-protocol` Function

Looks for the given name in the hashtable *$protocols*. Signals an error if no protocol with the given name can be found.

Signature find-protocol *frame-name* => *frame-type frame-name*

Parameters

- **frame-name** – An instance of `<string>`.

Values

- **frame-type** – The frame type for the requested frame name, an instance of <class>.
- **frame-name** – The name under which the frame is known in the registry, an instance of <string>.

find-protocol-field Function

Queries a field by name in a given binary data format. Errors if no such field is known in the binary data format.

Signature find-protocol-field *frame-type field-name* => *field*

Parameters

- **frame-type** – The type of a frame, an instance of <class>.
- **field-name** – The name of a field, an instance of <string>.

Values

- **field** – An instance of *<field>*.

Utilities

hexdump Generic function

Prints the given *data* in hexadecimal on the given *stream*.

Signature hexdump *stream data* => ()

Parameters

- **stream** – An instance of <stream>.
- **data** – An instance of <sequence>.

Discussion

Prints 8 bytes separated by a whitespace in hexadecimal, followed by two whitespaces, and another 8 bytes.

If the given *data* has more than 16 elements, it prints multiple lines, and prefix each with a line number (as 16 bit hexadecimal).

byte-offset Function

Computes the number of bytes for a given number of bits. A synonym for `rcurry(ash, 3)`.

Signature byte-offset *bits* => *bytes*

Parameters

- **bits** – An <integer>.

Values

- **bytes** – An <integer>.

bit-offset Function

Computes the number of bits which do not fit into a byte for a given number of bits. A synonym for `curry(logand, 7)`.

Signature bit-offset *bits* => *bits-not-in-byte*

Parameters

- **bits** – An <integer>.

Values

- **bits-not-in-byte** – An <integer> between 0 and 7.

byte-aligned Function

Checks that the given number of bits can be represented in full bytes, otherwise signals an <alignment-error>.

Signature byte-aligned *bits*

Parameters

- **bits** – An instance of <integer>.

data Generic function

Returns the underlying byte vector of a wrapper object, used for several untranslated leaf frames.

Signature data (object) => (#rest results)

Parameters

- **object** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

Note: should be removed from the API, or become internal

Errors

<out-of-bound-error> Class

Superclasses <error>

<out-of-range-error> Class

Superclasses <error>

<malformed-data-error> Class

Superclasses <error>

<parse-error> Class

Superclasses <error>

<inline-layering-error> Class

Superclasses <error>

<missing-inline-layering-error> Class

Superclasses <error>

Extension API

Extending Binary Data Formats

This domain-specific language defines a subclass of *<container-frame>*, and lots of boilerplate.

define binary-data Defining Macro

Macro Call

```
define [abstract] binary-data *binary-format-name* ([*super-binary-format*])
  [summary *summary*] [;]
  [over *over-spec* *] [;]
  [length *length-expression*] [;]
  [*field-spec*] [;]
end
```

Parameters

- **binary-format-name** – A standard Dylan class name.
- **super-binary-format** – A standard Dylan name, used superclass.
- **summary** – A Dylan expression consisting of a format-string and a list of arguments.
- **over-spec** – A pair of binary format and value.
- **length-expression** – A Dylan expression computing the length of a frame instance.
- **field-spec** – A list of fields for this binary format.

Discussion

Defines the binary data class *binary-data-name*, which is a subclass of *super-binary-format*. In the body some syntactic sugar for specializing the pretty printer (*summary* specializes `summary`), providing a custom length implementation (*length* specializes `container-frame-size`), and provide binary format layering information via *over-spec* (`<layering-field>`). The remaining body is a list of *field-spec*. Each *field-spec* line corresponds to a slot in the defined class. Additionally, each *field-spec* instantiates an object of `<field>` to store the static metadata. The vector of fields is available via the method `fields`.

```
summary: *format-string* *format-arguments*
```

This generates a method implementation for `summary`. Each *format-arguments* is applied to the frame instance.

```
over-spec: *over-binary-format* *layering-value*
```

The *over-binary-format* should be a subclass of `<header-frame>` or `<variably-typed-container-frame>`. The *layering-value* will be registered for the specified *over-binary-format*.

```
field-spec: [*field-attribute*] field *field-name* [:: *field-type*] [= *default-value*]

field-attribute: variably-typed | layering | repeated | enum

mapping: { *key* <=> *value* }
```

- *field-name*: Each field has a unique *field-name*, which is used as name for the getter and setter methods
- *field-type*: The *field-type* can be any subclass of `<frame>`, required unless `variably-typed` attribute provided.
- *default-value*: The *default-value* should be an instance of the given *field-type*.
- *field-attribute*: Syntactic sugar for some common patterns is available via attributes.
  - `variably-typed` instantiates a `<variably-typed-field>`.

  - `layering` instantiates a *<layering-field>*.
  - `repeated` instantiates a *<repeated-field>*.
  - `enum` instantiates a *<enum-field>*.
- *keyword-arguments*: Depending on the field type, various keywords are supported. Lots of values are standard Dylan expressions, where the current frame object is implicitly bound to `frame`, indicated by *frame-expression*.
  - fixup: A *frame-expression* computing the field value if no default was supplied, and the client didn't provide one (handy for length fields).
  - start: A *frame-expression* computing the start bit of the field in the frame.
  - end: A *frame-expression* computing the end bit of the field in the frame.
  - length: A *frame-expression* computing the length of the field.
  - static-start: A Dylan *expression* stating the start of the field in the frame.
  - static-end: A Dylan *expression* stating the end of the field in the frame.
  - static-length: A Dylan *expression* stating the length of the field.
  - type-function: A *frame-expression* computing the type of this *<variably-typed-field>*.
  - count: A *frame-expression* computing the amount of repetitions of this *<count-repeated-field>*.
  - reached-end?: A *frame-expression* returning a `<boolean>` whether this *<self-delimited-repeated-field>* has reached its end.
  - mappings: A *mapping* for *<enum-field>* between values and `<symbol>`

The list of fields is instantiated once for each binary data definition. If a static start offset, length, and end offset can be trivially computed (using constant folding), this is done during macro processing.

Several generic functions can be specialized on the *binary-format-name* for custom behaviour:

- *fixup!*
- *summary*
- *parse-frame*

Note: rename start, end, length to dynamic-start, dynamic-end, dynamic-length

---

Note: Check whether those field attributes compose in some way

---

`fixup!` Open Generic function

Fixes data in an assembled container frame.

Signature fixup! *frame* => ()

Parameters

- **`frame`** – A union of *<container-frame>* and *<raw-frame>*. Usually specialized on a subclass of `<unparsed-container-frame>`.

Discussion Used for post-assembly of certain fields, such as checksum calculations in IPv4, ICMP, TCP frames, compression of domain names in DNS fragments.

Defining a Custom Leaf Frame

A common structure in binary data formats are subsequent ranges of bits or bytes, each with a different meaning. There are some macros available to define frame types of common patterns.

`field-size` Open Generic function

Returns the static size of a given frame type. Should be specialized for custom fixed sized frames.

Signature field-size *frame* => *length*

Parameters

- **`frame`** – Any subclass of *<frame>*.

Values

- **`length`** – The bit size of the frame type <number>.

`high-level-type` Open Generic function

For translated frames, return the native Dylan type. Otherwise identity.

Signature high-level-type *frame-type* => *type*

Parameters

- **`frame-type`** – An instance of `subclass(<frame>)`.

Values

- **`type`** – An instance of <type>.

`assemble-frame-into` Open Generic function

Shuffle the bits in the given *packet* so that the *frame* is encoded correctly.

Signature assemble-frame-into *frame packet* => *length*

Parameters

- **`frame`** – An instance of *<frame>*.
- **`packet`** – An instance of *<stretchy-vector-subsequence>*.

Values

- **`length`** – An instance of <integer>.

`assemble-frame-into-as` Open Generic function

Shuffle the bits in the given *packet* so that the *frame* is encoded correctly as the given *frame-type*.

Signature assemble-frame-into-as *frame-type frame packet* => *length*

Parameters

- **`frame-type`** – A subclass of *<translated-frame>*.
- **`frame`** – An instance of <object>.
- **`packet`** – An instance of *<stretchy-vector-subsequence>*.

Values

- **`length`** – An instance of <integer>.

`define n-bit-unsigned-integer` Defining Macro

Describes an `<integer>` represented by a bit vector of arbitrary size.

Macro Call

```
define n-bit-unsigned-integer (*class-name* ; *bits* )
end
```

Parameters

- **`class-name`** – A Dylan class name which is defined by this macro.
- **`bits`** – The number of bits represented by this frame.

Discussion

Defines the class *class-name* with *`<unsigned-integer-bit-frame>`* as its superclass.

There are several predefined classes of the form `<Kbit-unsigned-integer>` with *K* between 1 and 15, and 20.

Operations

- *`high-level-type`* returns `limited(<integer>, min: 0, max: 2 ^ bits -1)`.
- *`field-size`* returns *bits*.

`define n-byte-unsigned-integer` Defining Macro

Describes an `<integer>` represented by a byte vector of arbitrary size and encoding (little or big endian).

Macro Call

```
define n-byte-unsigned-integer (*class-name-prefix* ; *bytes*)
end
```

Parameters

- **`class-name-prefix`** – A prefix for the class name which is defined by this macro.
- **`bytes`** – The number of bytes represented by this frame.

Discussion

Defines the classes *class-name-prefix* `-big-endian-unsigned-integer>` (superclass *`<big-endian-unsigned-integer-byte-frame>`* and *class-name-prefix* `-little-endian-unsigned-integer>` (superclass *`<little-endian-unsigned-integer-byte-frame>`*.

The following classes are predefined: `<2byte-big-endian-unsigned-integer>`, `<2byte-little-endian-unsigned-integer>`, `<3byte-big-endian-unsigned-integer>`, and `<3byte-little-endian-unsigned-integer>`.

Operations

- *`high-level-type`* returns `limited(<integer>, min: 0, max: 2 ^ (8 * *bytes*) - 1`.
- *`field-size`* returns *bytes* * 8.

`define n-byte-vector` Defining Macro

Defines a class with an underlying fixed size byte vector.

Macro Call

```
define n-byte-vector (*class-name* , *bytes*)
end
```

Parameters

- **class-name** – A standard Dylan class name.
- **bytes** – The number of bytes represented by this frame.

Discussion Defines the class *class-name*, as a subclass of *<fixed-size-byte-vector-frame>*. Calls *define leaf-frame-constructor* with the given *class-name* (without surrounding angle brackets).

Operations

- *field-size* returns *bytes* * 8.

define leaf-frame-constructor Defining Macro

Defines constructors for a given name.

Macro Call

```
define leaf-frame-constructor (*constructor-name*)
end
```

Parameters

- **constructor-name** – name of the constructor.

Discussion Defines the generic function *constructor-name* and three specializations:

Operations

- *constructor-name* <byte-vector> calls *parse-frame*
- *constructor-name* <collection>, converts the <collection> into a <byte-vector> and calls *constructor-name*.
- *constructor-name* <string>, which calls *read-frame*.

Predefined Leaf Frames

<unsigned-integer-bit-frame> Abstract Class

The superclass of all bit frames, concrete classes are defined with the *define n-bit-unsigned-integer*.

Superclasses *<fixed-size-translated-leaf-frame>*

Operations

- as <string>
- *assemble-frame*
- *parse-frame*
- *read-frame*

See also

- *define n-bit-unsigned-integer*

<boolean-bit> Class
A single bit, at the Dylan level a <boolean>.

The *high-level-type* returns <boolean>. The *field-size* returns 1.

Superclasses *<fixed-size-translated-leaf-frame>*

<unsigned-byte> Class
A single byte, represented as a <byte>.

Operations

- *high-level-type* returns <byte>.
- *field-size* returns 8.

Superclasses *<fixed-size-translated-leaf-frame>*

<variable-size-byte-vector> Abstract Class
A byte vector of arbitrary size, provided externally.

Superclasses *<variable-size-untranslated-leaf-frame>*

<externally-delimited-string> Class
A <string> of a certain length, externally delimited. The conversion method as is specialised on <string> and <externally-delimited-string>.

Superclasses *<variable-size-byte-vector>*

Note: should be a variable-size translated leaf frame, if that is possible.

<raw-frame> Class
The bottom of the type hierarchy: if nothing is known, a <raw-frame> is all you can have. *hexdump* can be used to inspect the frame contents.

Superclasses *<variable-size-byte-vector>*

<fixed-size-byte-vector-frame> Open Abstract Class
A vector of any amount of bytes with a custom representation. Used amongst others for IP addresses, MAC addresses

Superclasses *<fixed-size-untranslated-leaf-frame>*

Init-Keywords

- **data** – The underlying byte vector.

Operations

- as <string>
- *assemble-frame*
- *parse-frame*
- *read-frame*

See also

- *define n-byte-vector*

<big-endian-unsigned-integer-byte-frame> Abstract Class
A frame representing an <integer> of a certain size, depending on the size of the underlyaing byte vector.

The macro define n-byte-unsigned-integer-definer defines subclasses with a certain size.

Superclasses *<fixed-size-translated-leaf-frame>*

Operations

- as <string>
- *assemble-frame*
- *parse-frame*
- *read-frame*

See also

- *define n-byte-unsigned-integer*
- *<little-endian-unsigned-integer-byte-frame>*

<little-endian-unsigned-integer-byte-frame> Abstract Class

A frame representing an <integer> of a certain size, depending on the size of the underlying byte vector.

The macro `define n-byte-unsigned-integer-definer` defines subclasses with a certain size.

Superclasses *<fixed-size-translated-leaf-frame>*

Operations

- as <string>
- *assemble-frame*
- *parse-frame*
- *read-frame*

See also

- *define n-byte-unsigned-integer*
- *<big-endian-unsigned-integer-byte-frame>*

32 Bit Frames

The <integer> type in Dylan is represented by only 30 bits, thus 32 bit frames which should be represented as a <number> require a workaround. The workaround consists of using *<fixed-size-byte-vector-frame>* and converting to <double-float> values.

Note: This hack is awful and should be replaced by native 32 bit integers, or machine words.

<big-endian-unsigned-integer-4byte> Class

Superclasses *<fixed-size-byte-vector-frame>*

<little-endian-unsigned-integer-4byte> Class

Superclasses *<fixed-size-byte-vector-frame>*

big-endian-unsigned-integer-4byte Generic function

Signature big-endian-unsigned-integer-4byte (data) => (#rest results)

Parameters

- **data** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

little-endian-unsigned-integer-4byte Generic function

Signature little-endian-unsigned-integer-4byte (data) => (#rest results)

Parameters

- **data** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

byte-vector-to-float-be Function

Signature byte-vector-to-float-be (bv) => (res)

Parameters

- **bv** – An instance of <stretchy-byte-vector-subsequence>.

Values

- **res** – An instance of <float>.

byte-vector-to-float-le Function

Signature byte-vector-to-float-le (bv) => (res)

Parameters

- **bv** – An instance of <stretchy-byte-vector-subsequence>.

Values

- **res** – An instance of <float>.

float-to-byte-vector-be Function

Signature float-to-byte-vector-be (float) => (res)

Parameters

- **float** – An instance of <float>.

Values

- **res** – An instance of <byte-vector>.

float-to-byte-vector-le Function

Signature float-to-byte-vector-le (float) => (res)

Parameters

- **float** – An instance of <float>.

Values

- **res** – An instance of <byte-vector>.

Stretchy Vector Subsequences

The underlying byte vector which is used in binary data is a *<stretchy-byte-vector>*. To allow zerocopy while parsing, and providing each frame parser only with a byte vector of the required size for the type, there is a *<stretchy-vector-subsequence>* which tracks the byte-vector together with a start and end index.

Note: Should live in a separate module and types can be narrowed a bit further.

<stretchy-byte-vector> Constant

Type <type>

Value limited(<stretchy-vector>, of: <byte>)

<stretchy-vector-subsequence> Abstract Class

Superclasses <vector>

Init-Keywords

- **data** –
- **end** –
- **start** –

subsequence Generic function

Signature subsequence (seq) => (#rest results)

Parameters

- **seq** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

<stretchy-byte-vector-subsequence> Class

Superclasses *<stretchy-vector-subsequence>*

decode-integer Generic function

Signature decode-integer (seq count) => (#rest results)

Parameters

- **seq** – An instance of <object>.
- **count** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

encode-integer Generic function

Signature encode-integer (value seq count) => (#rest results)

Parameters

- **value** – An instance of <object>.
- **seq** – An instance of <object>.
- **count** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

CHAPTER

THREE

INTERNALS

Note: just a collection of random things.. not very useful at the moment

Internal API

sorted-frame-fields, get-frame-field, .. generic-function:: parent-setter field-count fields-initializer unparsed-class decoded-class fixup-protocol-magic

`layer-magic` Open Generic function

`container-frame-size` Open Generic function

Signature container-frame-size (frame) => (length)

Parameters

- **`frame`** – An instance of `<container-frame>`.

Values

- **`length`** – An instance of `false-or(<integer>)`.

`copy-frame` Generic function

Signature copy-frame (frame) => (#rest results)

Parameters

- **`frame`** – An instance of `<object>`.

Values

- **`#rest results`** – An instance of `<object>`.

`assemble-frame!` Generic function

Signature assemble-frame! (frame) => (#rest results)

Parameters

- **`frame`** – An instance of *`<frame>`*.

Values

- **`#rest results`** – An instance of `<object>`.

Container Frame Internals

Due to the two disjoint activities: parse a byte vector into a high-level frame, and assemble a high-level frame into a byte vector, there are two direct subclasses, a `<decoded-container-frame>`, which only has the high-level objects, and a `<unparsed-container-frame>` which keeps an underlying byte vector and an instance of `<decoded-container-frame>`.

Parsing strategy and length information (which can be contradictionary).

N

O

P

R

S

T

U

V

Symbols

A

B

C

D

E

F

H

L

P

R

S