A domain-specific language for manipulation of binary data in Dylan

Article · January 2007
DOI: 10.1145/1622123.1622148

2 authors:

Hannes Mehnert
University of Cambridge
10 PUBLICATIONS 38 CITATIONS

Andreas Bogk
HERE Technologies
6 PUBLICATIONS 7 CITATIONS

All content following this page was uploaded by Hannes Mehnert on 28 June 2015.
The user has requested enhancement of the downloaded file.
A Domain-Specific Language for manipulation of binary data in Dylan

Hannes Mehnert
hannes@mehnert.org
Dylan Hackers

Andreas Bogk
bogk@andreas.org
Dylan Hackers

ABSTRACT
We present a domain specific language for manipulation of binary data, or structured byte sequences, as they appear in everyday applications such as networking or graphics file manipulation. Our DSL is implemented as an extension of the Dylan language, making use of the macro facility. Dylan macros, unlike Common Lisp macros, are implemented as pattern matches and substitutions on the parse tree, and we will show the strengths and limits of this approach for the given problem.

1. INTRODUCTION
In the modern computer landscape, exchange of data as vectors of bytes is ubiquitous, and the concept of files or network packets has become second nature to every programmer. The concept appears to be simple, but closer inspection shows that those “bags of bytes” have an internal structure that the sender and receiver need to agree upon. A lot of the effort in designing the external interfaces for a given piece of software goes into writing code that reads or writes this structure according to some agreed-upon specification. This is time-consuming and error-prone.

We encountered this problem when writing a network analysis application in the Dylan programming language, where a large number of different network protocols needs to be supported. We decided to approach it by implementing a domain specific language for describing arbitrary binary formats, as well as the generalized abstractions for both parsing and generating byte vectors in these formats.

Our design goals were expressiveness, encapsulation, genericity and efficiency. By expressiveness we mean that definitions of the structure of a byte vector format should be compact and free of redundancy. Encapsulation refers to the goal of hiding and automating as much as possible from the user, such as platform endianness or computation of field offsets. Genericity means that our DSL shall be applicable to arbitrary binary formats. Efficiency is a demand to make usage of our DSL and framework as fast as it would have been with manually written manipulation code, ideally rivaling C code.

Section 2 shows the DSL we devised, and introduces the design of the framework behind it. In section 3, we have a look at related approaches. Section 4 finally gives an overview of what we have achieved and an outlook on future work.

2. SOLUTION
2.1 Dylan Macros
The Dylan macro system [7] provides a tool to extend the grammar of the core language by the user. Unlike the preprocessor macros of the C language, it doesn’t operate on the character stream input of the source file. Rather, it performs pattern matching and substitution on a grammatical level, by operating on the parse tree of the source input.

In a way, this can be seen as applying the idea of Lisp macros to a language with a more complex syntax. It is often said that Lisp is so powerful, because in it “code is data”. What’s meant by that phrase is that, by virtue of the simplicity of the s-expression grammar and its representation as nested lists, it is easy to manipulate source code on a structural level, and thus the power to do meta programming arises. A Lisp macro is expanded by finding its call site in the nested lists, which is equivalent to doing pattern matching on a nested list, and replacing the call by a nested list that’s produced from an expression in the macro definition. A Dylan macro is expanded by finding its call site, matching a Dylan grammar fragment expression on the parse tree, and replacing the call by another fragment expression with interpolated pattern variables.

Here is a simple example for a macro definition in Dylan:

```dylan
define macro inc!
  { inc!(?:expression) }
=>
  { ?expression := ?expression + 1 }
end
```

All macro definitions follow this general structure. The first line introduces the macro definition, and states the macro name. Then, the pattern and substitution fragments are given in curly braces, separated by a => token. Every token that does not have a leading question mark character is
If the name is omitted, it defaults to being the same as the name of grammar fragments. The name appears between the question mark and the colon, whereas a Lisp macro can evaluate standard are not as powerful as their Lisp counterpart. The right mark being replaced by the comma operator. However, Dylan macros as specified in the language standard are not as powerful as their Lisp counterpart. The right hand side of the Dylan macro can be thought of being the equivalent of a backquoted list, with the question mark being replaced by the comma operator.

When a macro is called in Dylan, the left hand side of the macro definition is matched with the call site, and pattern variables get bound to the fragment that’s written in their place. Our macro would match a call like inc!(foo) and would bind the pattern variable ?expression to the expression fragment foo. On the right hand side, the pattern variable is interpolated into the literal fragment, producing the desired substitution fragment. Our call would expand to foo := foo + 1.

The equivalent macro definition in Lisp would look like this:

```
(deffunc inc! (expression)
  '(setf ,expression (+ 1 ,expression)))
```

The structural similarities become apparent here. The left hand side of the Dylan macro definition is essentially equivalent to the macro name plus lambda list in the defmacro. Dylan uses a more powerful deconstruction mechanism that makes use of literal tokens, so macro arguments a.k.a. pattern variables need to be introduced with a question mark. The right hand side of the Dylan macro can be thought of being the equivalent of a backquoted list, with the question mark being replaced by the comma operator.

However, Dylan macros as specified in the language standard are not as powerful as their Lisp counterpart. The right hand side of macro definitions are always literal fragments with interpolated variables, whereas a Lisp macro can evaluate an arbitrary Lisp expression at compile time to produce a substitution. To keep the analogy: the macro expansion is always a backquoted list, and all unquoted elements are bindings introduced in the macro argument lambda list.

However, multiple patterns and substitutions can be given for a certain macro, and auxiliary rules can be specified for an even more fine-grained control over the syntax of the macro call. This allows to implement recursive macros, and makes Dylan macros powerful enough to write the equivalent of the Lisp loop macro, with its colorful syntax.

2.2 Terminology

We call a vector of bytes that has an associated definition for its interpretation 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 that follows the Composite pattern [5]. The description of the structure of a container frame in our DSL is referred to as a protocol definition.

2.3 Representation in Dylan

Since our goal is to provide an extension to Dylan for manipulating frames, we have to decide on a representation of frames as Dylan objects, and a set of functions on these objects to perform the manipulation. The obvious representation, which we have chosen, 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.

Access to the fields of a container frame is done via getter and setter functions. Since access to slots in a Dylan class is also done via getters and setters, this makes fields in a frame look like slots in a class to code manipulating these frames.

For convenience, Dylan provides syntactic sugar for calling getters and setters. The expression foo(bar) that calls the function foo with some bar as an argument can be written equivalently using the dot syntax: bar.foo. Likewise, foo.setter(23, bar) is the same as bar.foo := 23.

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 %e to %e\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 fields in the Ethernet frame are extracted and printed.

The appropriate classes and accessor functions are not written directly for container frames. Rather, they are created by invocation of the define protocol macro. 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.

2.4 A simple protocol definition

In the simplest case, a protocol definition lists the fields, giving a name and frame type to each. Here is an example.

```
define protocol foo (container-frame)
```
This macro usage defines a protocol named `foo`. It inherits from a `container-frame`. The protocol consists of two fields, named `source` and `destination`, with the same type `<address>`. This class is a subclass of `<leaf-frame>` and has a fixed size.

We will see the full expansion process later, but for now, we will show a simplified expansion that illustrates the general concept:

```dylan
define class <foo> (<container-frame>)
  class slot frame-fields = $frame-fields-<foo>;
  slot source :: <address>;
  slot destination :: <address>;
end;
define constant $frame-fields-<foo> = vector(make(<field>,
    name: "source",
    type: <address>),
    make(<field>,
    name: "destination",
    type: <address>));
```

This defines the class, `<foo>`, a subclass of `<container-frame>`. `<foo>` contains three slots, a class slot `frame-fields`, which refers to the constant `$frame-fields-<foo>`, a `source` and `destination` of type `<address>` each. The constant is defined as vector of `<field>` objects, one for each `field` statement in the macro call.

We're assuming here that `parse-frame` and assemble-frame just walk the list of fields, and fill in or read the appropriate slots in the class. The full version is more intricate and provides lazy parsing, we'll see this later.

The protocol-definer-macro does the expansion and is defined as following:

```dylan
define macro protocol-definer
  { define protocol ?:name (?superprotocol:name) ?fields:* end }
  => { define-class("<" ## ?name ## ">";
      "<" ## ?superprotocol ## ">";
      ?fields);
      frame-field("<" ## ?name ## ">"; ?fields) end;
```

This definition macro calls two function macros in the expansion, `define-class`, which defines a class with the given name, the superprotocol and the fields. The second macro call is `frame-field`, which defines the constant containing the fields of the protocol.

## is the splicing operator in Dylan macros. It is used here to add angle brackets to the protocol name, to generate a Dylan class name that follows the standard naming conventions.

```dylan
define macro define-class
  { define-class
      (?:name; ?superprotocol:name; ?fields:*) }
  => { define class ?name (?superprotocol)
    class slot frame-fields
      = "$frame-fields-" ## ?name;
    ?fields
    end }

  fields:
    { } => { }
    { ?field:* ; ... } => { ?field ; ... }

  field:
    { field ?:name :: ?type:name } => { slot ?name :: ?type, init-keyword: ?"name" }
end;
```

The `define-class` macro expands to a class definition with the corresponding name and superprotocol. The colon-seperated list of fields is matched by the auxiliary macro `fields`. This has two possible expansions: empty to terminate the recursion, or `?field:*` followed by a colon. The ellipsis ... means exactly the same thing as the pattern variable that is rewritten by this auxiliary rule set. So, a `?field:*` can be followed by another `?field:*`, providing recursive matches. The auxiliary rule set `field` converts a field with a name and a type to a slot with the same name, the specified type and an optional init-keyword. ?"name" treats the macro variable as a symbol.

```dylan
define macro frame-fields
  { frame-fields(?:name; ?fields:*) }
  => { define constant "$frame-fields-" ## ?name =
    vector(?fields); }

  fields:
    { } => { }
    { ?field:* ; ... } => { ?field, ... }

  field:
    { field ?:name :: ?type:name }
    => { make(<field>, name: ?"name", type: ?type) }
end;
```

The `frame-fields` expands to a constant, `$frame-fields-<foo>` in our example. It consists of a vector of fields. We call the function `vector` on the macro expansion of the `?fields` macro variable. The auxiliary rule set `fields` is similar to that of the previous macro. The only difference is the comma instead of the colon on the right hand side. This is needed because the `vector` function expects comma-separated objects. The `field` auxiliary rule set creates a `<field>` instance with its name as string and its type.
2.5 Ethernet frames, an illustrated example

A first real world example is Ethernet v2. It is the most common link-layer protocol used in IP networks. Our protocol definition of an Ethernet frame is shown:

```dylan
define protocol ethernet-frame (header-frame) 
  summary "ETH %d -> %d",
  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: payload-type(frame);
end;
```

Here, `<ethernet-frame>` inherits from `<header-frame>` which is an abstract subclass of `<container-frame>` and indicates that the frame consists of a header and a payload.

The second line starts with the `summary` keyword. This is used to provide a printing method on the defined protocol.

The destination and source fields are straightforward from the simple case, its type is a `<mac-address>`.

The fifth line contains the attribute `layering` in front of the field declaration. This provides the information that the value of this field controls the type of the payload, and introduces a registry for field values and matching payload types. We'll use this in a later example to declare how an IP frame is embedded in an Ethernet frame.

The type of `type-code` is `<2byte-big-endian-unsigned-integer>`. Unlike with a `<mac-address>` field, reading the field does not return an instance of the leaf frame type. Instead, translation into a Dylan `<integer>` is done. We call frames that have this property `translated` frames, and those without a matching Dylan type `untranslated` frames.

The sixth line uses another attribute, `variably-typed`, of a field. Most fields have the same type in all frame instances, these are statically typed. Some fields depend on the value of another field of the same protocol, these are variably typed. To figure out the type, a type function has to be provided for the variably typed field using the `type-function:`.

The first three fields are of fixed size, a `<mac-address>` has 6 bytes, a `<2byte-big-endian-unsigned-integer>` obviously 2 bytes. The payload consumes the rest of the byte vector.

We make an effort to compute the start offsets of each field in a frame at compile time, this is only possible if all preceding fields have a fixed length. We're implicitly assuming here that fields are packed next to each other. It's possible to override this behaviour, as we will see later.

The macro which matches the above protocol definition and generates several classes and methods is shown in detail.

The first new feature the `summary` keyword, it is matched by an additional rule of the protocol-definer-macro:

```dylan
{ define protocol ?:name (?superprotocol:name) 
  summary ?:summary:*
  ?:fields:*
end }
```

The helper macro `summary-generator` is called with the class name and the rest of the summary line. Also define protocol is called without the summary line, which matches another macro rule. The `summary-generator` macro is shown next:

```dylan
define macro summary-generator 
{ summary-generator(?type:name; 
  ?:summary-string:expression, 
  ?:summary-getters:*) } 
=> { define method summary (frame :: ?type) 
  => (result :: <string>);
  apply(format-to-string,
    ?summary-string,
    map(rcurry(apply, frame),
      list(?summary-getters)));
end; }
```

This macro defines a method `summary` on an instance of the specific frame. The method calls `format-to-string` with the `summary-string`, which matches the first expression. In our example it is "ETH %d => %d". The arguments of the format string are computed by applying each provided getter on the actual instance of the frame.

The generation of classes and accessors is a little more involved in our real code than in the simple example above. To provide efficiency, we generated two Dylan classes for each protocol definition: decoded frames, which store only the high-level objects in a slot for each field, and untranslated frames, which store a byte vector and cache in the form of a decoded frame. Parsing is done lazily on untranslated frames, only when a field value is accessed, it is parsed. This reduces the amount of work done for each received frame.

We generate custom getter methods for untranslated classes which store the value in the cache object. Each getter needs access to its corresponding field to compute the start offset of the field in the byte vector. We also need to keep track of start and end offsets we already computed for a certain field, which we do in a class `<frame-field>`.

To provide quick access to the `<frame-field>` for a given field, we give an index number to every field. The index is generated by the `frame-field-generator` macro which is called by the `protocol-definer-macro` as shown:

```dylan
frame-field-generator
(""## ?name "")
  field-count
```
It has three parameters, the class name, the start index, which is the result of field-count on the superclass, and the field specifications.

The frame-field-generator calls itself recursively in the first rule, incrementing count by one. Once ?rest is not present, the second macro rule matches which defines the method field-count. We depend on constant folding in the compiler to fold the ?count expression to an integer in order to make field access efficient.

The first rule in the macro calls unparsed-frame-field-generator, which is again a macro:

define macro unparsed-frame-field-generator
{ unparsed-frame-field-generator
  (?name, ?frame-type:name, 
   ?field-index:expression) }
=> { define inline method ?name 
     (mframe :: ?frame-type) 
     if (mframe.cache.?name) 
       mframe.cache.?name := value; 
     else 
       let frame-field 
         = get-frame-field(?field-index, mframe); 
       let (value, parsed-end) 
         = parse-frame-field(frame-field, mframe); 
       mframe.cache.?name := value; 
     end; 
   end; }
end;

This generates a getter which does a lookup in the cache object and if the slot has a value returns this. If it does not have a value, thus is not yet parsed, the get-frame-field with the index and the actual frame instance is called. This returns an object with offsets for the specific field in the actual frame, inheriting the information from the <field> object. After that, parse-frame-field is called which returns the value of the field in the frame. This value is stored in the cache object and returned.

We also support in-place modification of frames. A field of an unparsed frame can be set on high-level, and the corresponding bits of the byte vector will automatically be changed, as long as no other field offsets change during this operation. This provides efficiency when a lot of similar frames should be generated. The setter method for a field is also defined in the unparsed-frame-field-generator macro:

    { define inline method ?name "-setter" 
      (value, mframe :: ?frame-type) => (res) 
      let frame-field 
        = get-frame-field(?field-index, mframe); 
      assemble-field-into 
        (frame-field.field, 
         mframe, 
         subsequence(mframe.packet, 
           start: frame-field.start-offset)); 
      value; 
      end; 
    end; 

First, the setter in the cache object is called. Then the corresponding metadata object is looked up in the vector. Finally assemble-field-into with the field, the frame and the proper subsequence of the byte vector is called. This method converts the value to binary representation and modify the corresponding bits in the byte vector.

To get the Dylan class of a field, with translated and untranslated frames in place, we defined a generic function, high-level-type(frame-type :: subclass(<frame>)) => (res :: <type>). This is used in the class definition of a decoded class.

There are fields which can be present multiple times, like IP options in an IPv4 frame. We introduced the attribute repeated for these fields, and translate their type to a <collection> to store multiple occurrences. The decoded-class-definer is similar to the define-class macro in the simple protocol definition:

define macro decoded-class-definer
{ decoded-class-definer 
  (?name, ?superclasses:*; ?fields:*) 
  => { define class ?name (?superclasses) 
    ?fields 
  end } 
fields:
  { } => { }
  { ?field:*; ... } => { ?field ; ... }
Each field is translated to a slot of the decoded class, by default initialized to #f.

The frame-field macro which defines the vector of fields of the container frame is only shown partially, see the full source for all details. The field auxiliary macro looks like this:

```small
field:

{ } => { }

{ variably-typed field ?:name, ?rest:* }

=> { slot ?name :: false-or(<frame>) = #f,
      init-keyword: ?#"name" }

{ repeated field ?:name ?rest:* }

=> { slot ?name :: false-or(<collection>) = #f,
      init-keyword: ?#"name" }

{ ?attrs:* field ?:name ::?

  ?field-type:name ?rest:* }

=> { slot ?name :: false-or(high-level-type(?field-type)) = #f,
      init-keyword: ?#"name" }

end;
```

There are two different cases of variably typed fields shown: with and without a default initialization value. There are four different cases for other fields, with and without default initialization value (only the latter is shown here) as well as with and without optional arguments.

The attributes auxiliary macro specifies which field class is instantiated based on the given attributes:

```small
attributes:

{ } => { <single-field> }

{ layering } => { <layering-field> }

{ repeated } => { <repeated-field> }
```

The args auxiliary macro supports several keywords for frames:

```small
args:

{ static-start: ?start:expression, ... }

=> { static-start: ?start, ... }

{ type-function: ?type:expression, ... }

=> { type-function: method(=?frame :: <frame>)
     ?type
     end, ... }
```

Only two of the possible arguments are shown here, again see the full source for details. First comes the static-start: keyword, which is followed by an expression and expands to the static-start: init keyword with the expression as its value. The type-function: keyword is used in variably typed fields, already seen in the ethernet frame definition. This expands to an anonymous method which introduces an unhygienic lexical binding to a frame with ?=frame. This is how frame in payload-type(frame) is bound.

### 2.6 IPv4 Frame - more features of our DSL

Another real world protocol is IPv4, our frame definition is shown:

```small
define protocol ipv4-frame (header-frame)
summary "IP SRC %= DST %=",
source-address, destination-address;
over <ethernet-frame> #x800;
field version :: <4bit-unsigned-integer> = 4;
field header-length :: <2byte-big-endian-unsigned-integer>,
fixup: ceiling/
(reduce
(\+, 20,
map(compose(byte-offset, frame-size),
frame.options)),
4);
field type-of-service :: <unsigned-byte> = 0;
field total-length
  :: <2byte-big-endian-unsigned-integer>,
fixup: frame.header-length * 4 +
byte-offset(frame-size(frame.payload));
field identification
  :: <2byte-big-endian-unsigned-integer> = 23;
field evil :: <1bit-unsigned-integer> = 0;
field dont-fragment :: <1bit-unsigned-integer> = 0;
field more-fragments :: <1bit-unsigned-integer> = 0;
field fragment-offset
  :: <13bit-unsigned-integer> = 0;
field time-to-live :: <unsigned-byte> = 64;
layering field protocol :: <unsigned-byte>;
field header-checksum
```

There are two different cases of variably typed fields shown: with and without a default initialization value. There are four different cases for other fields, with and without default initialization value (only the latter is shown here) as well as with and without optional arguments.
The next interesting field is the repeated field if otherwise unspecified. Let’s take a closer look at the type declaration on their static counterparts, static-start:

type-function:

during assembly to field of an Ethernet frame with an IPv4 frame as payload type-code frame. The protocol stacking code also sets the assemble-frame is called in necessarily equivalent to the end offset of the previous field.

The payload field of the Ethernet frame definition is the first field we encounter that has a start offset that’s not necessarily equivalent to the end offset of the previous field. Instead, it is computed by the value of the header-length field, which is given in 32 bit words. We supply an expression to calculate the start offset via the start: keyword. Other keywords we support here are end: length:; and their static counterparts, static-start: static-end: and static-length:.

Let’s take a closer look at the type declaration on the ip-option field, which is <ip-option-frame>. This is an abstract protocol inheriting from <variably-typed-container-frame>.

define abstract protocol ip-option-frame
  (variably-typed-container-frame)
hand side of a macro pattern, which evaluate to a phrase grammar fragment. A literal phrase fragment is also added. For instance, under this proposal, our initial example macro could be written like this:

```dylan
define macro inc!
  { inc!(?expression) } =>
  inc!-generator(?expression)
end;

define function inc!-generator(expression)
  #{ ?expression := ?expression + 1 }
end;
```

Note the missing curly brackets on the right hand side of a macro, instead, a function is called. The function implementation uses the newly introduced hashmark-curly syntax to indicate a literal grammar fragment, much like a backquoted list in Lisp. The returned fragment is used as the expansion.

Having this feature would be tremendously useful, and would increase performance and elegance of our code. Unfortunately, it is unimplemented, mainly because compile-time evaluation semantics of Dylan isn’t nailed down anywhere. However, the Open Dylan compiler (ex Harlequin Dylan) uses a facility like this internally, making use of bootstrapping tricks to bridge the gap between compile time and runtime. We intend to make this functionality usable for our code.

Finally, there is a wart in the elegance of protocol definitions in our language. For example, the start of the field `payload` depends on the value of another field `header-length`. This dependency must be annotated on both fields: once to determine the start of `payload` from the value of `header-length` during parsing, and once to calculate the value of `header-length` from the start of `payload` during assembly. Both expressions are related, but we lack the kind of symbol manipulation capability required to translate one into the other.

Our plans for future work include further refinement of our DSL to match other binary formats as they are encountered. Also, we plan to use the procedural macro system found in the Open Dylan compiler core to improve performance of getter methods.

5. REFERENCES