Université de Montréal

BinDat:
Bin there, done Dat

Stefan Monnier

monnier@iro.umontreal.ca

Université de Montréal

DIRO

Overview

What's BinDat anyway?

Problems encountered

- Lack of flexibility
- Repeated assaults on kitten

The new design

- Simpler and more flexible by *exposing code as code*
- Try and keep efficiency in mind

Examples

What's BinDat anyway

An Emacs Lisp library. From the horse's mouth:

> Packing and unpacking of (binary) data structures.
>
> The data formats used in binary files and network protocols are often structured data which can be described by a C-style structure such as the one shown below. Using the Bindat package, decoding and encoding binary data formats like these is made simple using a structure specification which closely resembles the C style structure declarations.
>
> Encoded (binary) data is stored in a unibyte string or vector, while the decoded data is stored in an alist with `(field . value)` pairs.

More concretely

A description language:

```
struct data {
  char opcode;
  char id[7];
  int  length;
  char data[];
};
```

```
(setq data-bindat-spec
  '((opcode    u8)
    (id        strz 7)
    (length    i32r)
    (data      vec (length))
    (align     4)))
```

With 3 functions to make use of those descriptions:

`(bindat-pack` *spec struct*`)`

`(bindat-length` *spec struct*`)`

`(bindat-unpack` *spec raw*`)`

Example use

```
[ 192 168 1 100 192 168 1 101 01 28 21 32 2 0 0 0
  2 3 5 0 ?A ?B ?C ?D ?E ?F 0 0 1 2 3 4 5 0 0 0 1
  4 7 0 ?B ?C ?D ?E ?F ?G 0 0 6 7 8 9 10 11 12 0 ]
```

```
((header (dest-ip   . [192 168 1 100])
         (src-ip    . [192 168 1 101])
         (dest-port . 284)
         (src-port  . 5408))
 (items . 2)
 (item ((data   . [1 2 3 4 5])   ((data   . [6 7 8 9 10 11 12])
        (id     . "ABCDEF")       (id     . "BCDEFG")
        (length . 5)              (length . 7)
        (opcode . 3)              (opcode . 4)
        (type   . 2))             (type . 1))))
```

Patient 0 of BinDat

Switch to `lexical-binding` broke `weechat.el`:

```
(defconst weechat--relay-message-spec
  '((length u32)
    (compression u8)
    (id struct weechat--relay-str-spec)
    (data vec
          (eval (- (bindat-get-field struct 'length)
                   4    ;length
                   1    ;compression
                   (+ 4 (length (bindat-get-field
                                 struct 'id 'val)))))))
```

The DSL's original syntax

```
SPEC ::= (ITEM... )

ITEM ::= ([FIELD] TYPE)
       |  ([FIELD] eval FORM)      -- Eval FORM for side
       |  ([FIELD] fill LEN)       -- Skip LEN bytes
       |  ([FIELD] align LEN)      -- Skip to next multi
       |  ([FIELD] struct SPEC_NAME)
       |  ([FIELD] union ARG (TAG SPEC)... [(t SPEC)])
       |  ([FIELD] repeat COUNT ITEM...)
```

The DSL's original syntax (cont'd)

```
TYPE ::= (eval EXPR)
       |  u8   | byte          -- Length 1
       |  u16  | word | short  -- Length 2, network byt
       |  u24                  -- 3-byte value
       |  u32  | dword | long  -- Length 4, network byt
       |  u16r | u24r | u32r   -- Little endian byte or
       |  str LEN              -- LEN byte string
       |  strz LEN             -- LEN byte (zero-termin
       |  vec LEN [TYPE]       -- Vector of LEN items o
       |  ip                   -- 4 byte vector
       |  bits LEN             -- List with bits set in
```

The DSL's original syntax (cont'd)

```
FIELD ::= NAME | (eval EXPR)

LEN   ::= ARG |  <omitted> | nil

TAG   ::= LISP_CONSTANT
        |  (eval EXPR)  -- Return non-nil if tag match
                           current ARG in 'union'.

ARG   ::= (eval EXPR)
        |  INTEGER_CONSTANT
        |  ([NAME | INTEGER]...)
           -- Field NAME or array index relative
              to current structure spec.
```

Problems I saw with BinDat

The spec imposes the use of `eval`: kitten hell

Var `struct` undocumented, tho indispensable!

Grammar is more complex than necessary

- 9 distinct nonterminals, with ambiguity
- Vector length can be `INTEGER` or `(LABEL)` or `(eval EXP)`
- Lots of places allow an `(eval EXP)`, but not all!

No Edebugging

Not conveniently extensible

Problem sample 1

```
(defun weechat--message-available-p ()
  (and (> (buffer-size) 5)
       (>= (buffer-size)
           (bindat-get-field
            (bindat-unpack '((len u32))
                           (buffer-string))
            'len))))
```

Useless construction of an alist

Problem sample 2

```
(bindat-get-field
 (bindat-unpack
  `((:val
     ,(cond ((= n 1) 'u8)
            ((= n 2) 'u16)
            ((= n 4) 'u32)
            (t (error "websocket-get-bytes: ..."))))))
  s)
 :val)
```

Fixed integer sizes (no `(eval EXP)` available there)

BinDat only used to pack&unpack integers!

The new design

Make it all code!

Expose the field's values as local variables

```
(defconst weechat--relay-message-spec
  (bindat-type
   (length uint 32)
   (compression uint 8)
   (id type weechat--relay-str-spec)
   (data vec (- length
                4   ;length
                1   ;compression
                (+ 4 (length (bindat-get-field
                              id 'val)))))))
```

New grammar

```
TYPE ::= uint BITLEN          - Big-endian unsigned in
         uintr BITLEN         - Little-endian unsigned
         str LEN              - Byte string
         strz [LEN]           - Zero-terminated byte-s
         bits LEN             - Bit vector (LEN is cou
         fill LEN             - Just a filler
         align LEN            - Fill up to the next mu
         vec COUNT TYPE       - COUNT repetitions of T
         type EXP             - Indirection; EXP shoul
         unit EXP             - 0-width type holding t
         struct FIELDS...     - A composite type

FIELDS... ::= (LABEL TYPE)...
```

Advantages

Support for Edebug

`struct`, `repeat`, `align` not special cased any more

No need for `union`: can use `cond` or `pcase` instead!

No need to use a single-field struct for simple types

Flymake and the compiler can pester you

Kitten can safely play with your mouse!

Everything is code! Even `TYPE`!

```
(bindat-type vec (sqrt -1)
                 if (reino-del-revés)
                    (uintr 32) (uint 32))
```

New features

Control over the representation of unpacked `structs`:

- `:unpack-val EXP` tells how to construct the unpacked value from its fields's unpacked values.
- `:pack-val EXP` tells how to extract the unpacked value of a field from the unpacked value of the containing object.
- `:pack-var VAR` gives a name to the unpacked value of an object, so `:pack-val` can refer to it.

Define new types with `bindat-defmacro`!

Example, rookie

Timestamp values in OSC:

```
(defconst osc-timetag
 (let ((hz (ash 1 32)))
  (bindat-type :pack-var time
   (ticks uint 64
          :pack-val (+ (car (time-convert time hz))
                       (* osc-ntp-offset hz)))
   :unpack-val (cons (- ticks (* osc-ntp-offset hz))
                     hz))))
```

`bindat-pack/unpack` convert to/from proper time values!

Admittedly, we're back to single-field structs :-(

Example, journeyman

```
(defconst bindat-test--LEB128
 (bindat-type
  letrec
  ((loop
    (struct :pack-var n
            (head u8
                  :pack-val (+ (logand n 127)
                               (if (> n 127) 128 0)))
            (tail if (< head 128) (unit 0) loop
                  :pack-val (ash n -7))
            :unpack-val (+ (logand head 127)
                           (ash tail 7)))))
  loop))
```

Conclusion

Simpler, more flexible, more powerful, better behaved.

Significantly faster as well, by the way.

Distinction between `bindat-defmacro` and `bindat-type`?

No bit-level control

Not anywhere near GNU Poke's power and flexibility

Not used very much