Endian Madness

Roy Ratcliffe[1,*]

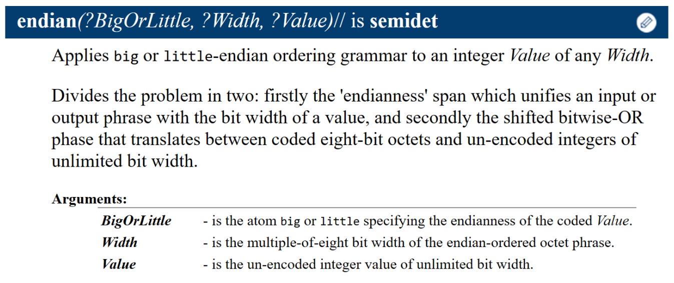
Abstract

Introduces definite-clause grammars (DCG predicates) for dealing with endian-oriented octet difference lists.

Introducing definite-clause grammars for dealing with endian-oriented octet list slices:

- `endian//3`
- `big_endian//2`
- `little_endian//2`

These grammar predicates unify octet $Codes|Code \in \mathbb{Z}, 0 \ldots 255$ with arbitrarily sized integer $Value$ terms or vice versa. See Figure 1.



endian*(?BigOrLittle, ?Width, ?Value)*// is **semidet**

Applies `big` or `little`-endian ordering grammar to an integer *Value* of any *Width*.

Divides the problem in two: firstly the 'endianness' span which unifies an input or output phrase with the bit width of a value, and secondly the shifted bitwise-OR phase that translates between coded eight-bit octets and un-encoded integers of unlimited bit width.

Arguments:

BigOrLittle	- is the atom `big` or `little` specifying the endianness of the coded *Value*.
Width	- is the multiple-of-eight bit width of the endian-ordered octet phrase.
Value	- is the un-encoded integer value of unlimited bit width.

Figure 1: Predicate `endian//3` manual entry screenshot

1. Two-Phase Approach

The implementation of the grammar divides the problem in two: firstly the ‘endianness’ span which unifies an input or output phrase with the bit width of a value, and secondly the shifted bitwise-OR phase that translates between coded eight-bit octets and un-encoded integers of unlimited bit width by accumulation.

1.1. Pure Endianness

First, unite ‘difference lists’ of octet codes with *zero* or more items by a width.

```
endianness(Width, Octets) -->
    { var(Width), !
    },
    remainder(Octets),
    { length(Octets, Len),
      Width is Len << 3
    }.
endianness(Width, Octets) -->
    { Width_ is Width /\ 2'111,
      Width_ == 0,
      Len is Width >> 3,
      length(Octets, Len)
    },
    Octets.
```

Multiple predicate argument modes exist. The $Width$ term can be either a variable or an integer. For unknown widths, the clauses span the remainder of the difference lists. The length of the outstanding list of codes determines the final width multiplied by eight.

The *Octets* may also have variable i.e. unbound items. The grammar does not examine the codes themselves; it only concerns the length and its relationship to width. The grammar fails if the width is *not* a multiple of eight.

1.2. Bitwise Shifting

Unifying some prescribed list of octets with its equivalent integer sum requires bit shifting and bitwise-OR operations over an accumulator. There is a catch, however. Merging an 8-bit byte from the list depends on two things: the appropriate order because big versus little determines the order, and the mode of the arguments because accumulating from a ‘ground’ value to a ‘variable’ list differs by computation from its reverse.

The two big-endian predicates follow, $endian_{big}(++,-)$ and $endian'_{big}(-,+)$ in predicate mode notation. The double plus indicates that the none-prime $(++,-)$ predicate requires a fully bound list of integers where for any octet code $\forall_x \bullet 0 \leq x \in \mathbb{Z} \leq 255$.

```
big_endian([], Value, Value).
big_endian([H|T], Value0, Value) :-
    0 =< H,
    H =< 255,
    Value_ is H \/ (Value0 << 8),
```

*Corresponding author
Email address: `roy@ratcliffe.me` (Roy Ratcliffe)
[1]See more hackery at GitHub.

```
    big_endian(T, Value_, Value).

big_endian_([], Value, Value).
big_endian_([H|T], Value0, Value) :-
    big_endian_(T, Value0, Value_),
    H is Value_ /\ 16'ff,
    Value is Value_ >> 8.
```

Notice that in $(-, +)$ mode the accumulator recurses *first* and then the residual $Value'$ merges with the accumulated $Value$ because the first octet code is the most-significant byte of the value for big-endian integer representations, rather than the least-significant. The $0 \leq H \leq 255$ guard conditions ensure failure for non-octet code items in the list.

Little-endian accumulators perform the same logical unification in reverse. The only difference between big and little: recurse first or recurse last. Apart from that subtle but essential difference, the inner computation behaves identically. Indeed, the inner shift and bitwise-OR deserve some refactor work to share the underlying behaviour. They could factor out as $acc(+, +, -)$ and $acc'(-, +, +)$ predicates[2].

```
little_endian([], Value, Value).
little_endian([H|T], Value0, Value) :-
    little_endian(T, Value0, Value_),
    0 =< H,
    H =< 255,
    Value is H \/ (Value_ << 8).

little_endian_([], Value, Value).
little_endian_([H|T], Value0, Value) :-
    H is Value0 /\ 16'ff,
    Value_ is Value0 >> 8,
    little_endian_(T, Value_, Value).
```

Put this all together with high-level grammar `endian//3`. It applies *big* or *little*-endian ordering grammar to an integer $Value$ of any $Width$.

```
endian(big, Width, Value) -->
    big_endian(Width, Value).
endian(little, Width, Value) -->
    little_endian(Width, Value).

big_endian(Width, Value) -->
    { var(Value), !
    },
    endianness(Width, Octets),
    { big_endian(Octets, 0, Value)
    }.
big_endian(Width, Value) -->
    endianness(Width, Octets),
    { big_endian_(Octets, Value, _)
    }.

little_endian(Width, Value) -->
    { var(Value), !
    },
    endianness(Width, Octets),
    { little_endian(Octets, 0, Value)
    }.
little_endian(Width, Value) -->
    endianness(Width, Octets),
    { little_endian_(Octets, Value, _)
    }.
```

[2]This happens in the final refactor.

2. Results

How does it work? This far, it all seems a little bit academic. Who cares about finding integers of different widths and endianness in arbitrary lists of octets?

Suppose you have a frame of octets. Starting as simply as possible, take $[1, 2, 3, 4]$. Say you want to extract two 16-bit integers, the first has big-endian order, the second little-endian. The first value extracted *should* be evaluated as $A = 1 \times 2^8 + 2 = 258$ and the second $B = 4 \times 2^8 + 3 = 1024 + 3 = 1027$. Applying the `endian//3` grammar to this same trivial exemplar gives us the correct result.

```
?- phrase((endian(big, 16, A),
   endian(little, 16, B)), [1, 2, 3, 4]).
A = 258,
B = 1027.
```

Of course, this works in reverse as a generator of octets.

```
?- phrase((endian(big, 16, 258),
   endian(little, 16, 1027)), A).
A = [1, 2, 3, 4].
```

Correct again. Naturally, the bit width of 16 appears here just to demonstrate but can be any multiple of eight. String together multiple `endian//3` elements to extract, and simultaneously inject by two-way inference, any integer values from any sequences of octets.

3. Usefulness

Why would these grammars be useful? They might find useful applications in a protocol stack, for example. Datagrams would need encoding and decoding. Typically such frames encode multiple fixed-length fields of prescribed endian order. Communication layers often work at packing and unpacking between values and frames. Not limited to integers, floats of different sizes also transcode in alternative endian orders. Integers capture bits and a floating-point grammar might re-use the endian grammar as an intermediate form for standard 'float' codings such as IEEE-754.

In Prolog, integers have arbitrary width without limit, even up to the maximum size of the available memory. These grammars quietly discard any overflow and fill up any underflow with zeros as an application might expect. Higher levels in the protocol stack can thereby concern itself with the content of the bits rather than their coded widths and orders. The concerns for endianness do not propagate upwards where such details interfere and multiply any existing complexities—separation of concerns.