UPPSALA UNIVERSITET

Bit-level Binaries and Generalized Comprehensions in Erlang

Per Gustafsson and Kostis Sagonas
Dept of Information Technology
Uppsala University

Binaries as we know them

- Introduced in 1992 as a container for object code
- Used in applications that do I/O, networking or protocol programming
- A proposal for a binary datatype and a syntax was made in 1999 and a revised version was adopted in 2000
- Since then, binaries have been used extensively, often providing innovative solutions to common telecom programming tasks

UPPSALA UNIVERSITET

Binaries are not so flexible

Some limitations:

- Binaries are byte streams, not bit streams
- Segment sizes cannot be arbitrary arithmetic expressions

Both undermine the use of the binary syntax for writing high level specifications

This work:

We show how to lift these limitations while maintaining backward compatibility

UPPSALA UNIVERSITET

Make binaries as flexible as lists

- In lists:
  - deconstructing a list always yields valid terms
  - can be constructed using list comprehensions
- In binaries:
  - deconstructing a binary sometimes yields terms which cannot be represented as Erlang binaries
  - no binary comprehensions are available

- **This work:**
  - allows binaries to represent bit streams
  - introduces binary comprehensions
  - introduces extended comprehensions to make conversions between lists and binaries simpler

UPPSALA UNIVERSITET

Flexible bit-level binaries

- The multiple-of-eight size restriction is lifted
- The size field of a segment can contain an arbitrary arithmetic expression
- No type specifier is needed in binary construction

Pros and cons of bit-level binaries

+ Allows natural representation of bit fields
  - `<<BitSize:8, BitField:BitSize/binary, ...`
+ Helps avoid padding calculations
  - `Pad = (8 - ((X + Y) rem 8)) rem 8,`
+ Makes binary matching as easy for bit streams as it was for byte streams

– Introduces a speed trade-off

Pattern Matching
- byte streams vs bit streams

```
keep_0XX(<<0:8,X:16,Rest/binary>>) ->
   <<0:8,X:16,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:24,Rest/binary>>) ->
   keep_0XX(Rest);
keep_0XX(<<>>) ->
   <<>>.
```

This function only keeps the byte triples whose first byte is 0.

But what if me want to keep the bit triples whose first bit is 0?

UPPSALA UNIVERSITET

Pattern Matching

- byte streams vs bit streams

```
keep_0XX(<<0:1,X:2,Rest/binary>>) ->
   <<0:1,X:2,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:3,Rest/binary>>) ->
   keep_0XX(Rest);
keep_0XX(<<>>) ->
   <<>>.
```

This is how it ought to look!

Pattern Matching
- byte streams vs bit streams

```
keep_0XX(Bin) -> keep_0XX(Bin, 0, 0, <<>>).

keep_0XX(Bin, N1, N2, Acc) ->
  Pad1 = (8 - ((N1+3) rem 8)) rem 8,
  Pad2 = (8 - ((N2+3) rem 8)) rem 8,
  case Bin of
    <<_:N1, 0:1, X:2, _:Pad1, _/binary>> ->
      NewAcc =
      <<Acc:N2/binary-unit:1, 0:1, X:2, 0:Pad2>>,
      keep_0XX(Bin, N1+3, N2+3, NewAcc);
    <<_:N1, _:3, _:Pad1, _/binary>> ->
      keep_0XX(Bin, N1+3, N2, Acc);
    <<_:N1>> -> Acc
  end.
```

This is how you have to write it today!

Allowing arithmetic expressions in the size field

Consider this classic example of the bit syntax:

```
case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13, TTL:8, Proto:8,
    SrcIP:32, DestIP:32,
    RestDgrm/binary>> ->
      OptsLen = Hlen – 5,
      <<Opts:OptsLen/binary-unit:32,
        Data/binary>> = RestDgrm,
      ...
end
```

UPPSALA
UNIVERSITET

Allowing arithmetic expressions in the size field

Using flexible binaries it could be written in the following manner:

```
case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13, TTL:8,
    Proto:8, SrcIP:32, DestIP:32,
    Opts:((Hlen - 5)*32)/binary,
    Data/binary>> ->  ...
end,
```

UPPSALA UNIVERSITET

No need for a type-specifier in binary construction

Consider the following code:

```
X = <<1, 2, 3>>,
B = <<X, 4, 5>>
```

It causes a runtime exception. To avoid this you must explicitly specify the type

```
X = <<1, 2, 3>>,
B = <<X/binary, 4, 5>>
```

We want to lift this restriction, the type should default to the type of the variable.

Binary Comprehensions

Analogous to List Comprehensions

- List Comprehensions represent a combination of map and filter

Comprehensions require a notion of an element

- For binary comprehensions the user must specify what they consider as an element

UPPSALA UNIVERSITET

Binary Comprehensions:

Introductory Example, invert

Using list comprehension:

```
invert(ListOfBits) ->
  [bnot(X) || X <- ListOfBits]
```

Using binary comprehension:

```
invert(Binary) ->
  <<bnot(X):1 || X:1 <- Binary>>
```

If your binary is byte-sized:

```
invert(Binary) ->
  <<bnot(X):8 || X:8 <- Binary>>
```

UPPSALA UNIVERSITET

Binary Comprehensions:

UU-decode

Using a binary comprehension UU-decode basically becomes a one-liner in Erlang

```
uudecode(UUBin) ->
  <<(X-32):6 || X:8 <- UUBin, 32=<X, X=<95 >>
```

Note the filter expressions which make sure that inserted characters such as line-breaks are dropped

UPPSALA UNIVERSITET

Extended comprehensions

Can we use list generators in binary comprehensions?

```
convert_to_binary(ListofWords) ->
  <<X:32 || X <- ListofWords>>.
```

YES !

Extended comprehensions

Can we use binary generators in list comprehensions?

```
convert_to_listofwords(Binary) ->
  [X || X:32 <- Binary].
```

YES !

UPPSALA UNIVERSITET

Generators

Note that we need to be able to separate list generators from binary generators.

List generators:

```
P <- E_L
```

Binary generators:

```
S_1 ... S_n <= E_B
```

P	–	a pattern
E_L	–	an Erlang expression which evaluates to a list
S_i	–	a binary segment
E_L	–	an Erlang expression which evaluates to a binary

UPPSALA UNIVERSITET

Implementation of extended binary comprehensions

- We present a simple translation of extended comprehensions into Erlang in the form of rewrite rules in the paper

- Using these simple rules the cost of building the resulting binary is quadratic in the number of segments

- We present another set of rewrite rules which gives linear complexity, but the rules are slightly less straight-forward

Implementation of extended binary comprehensions

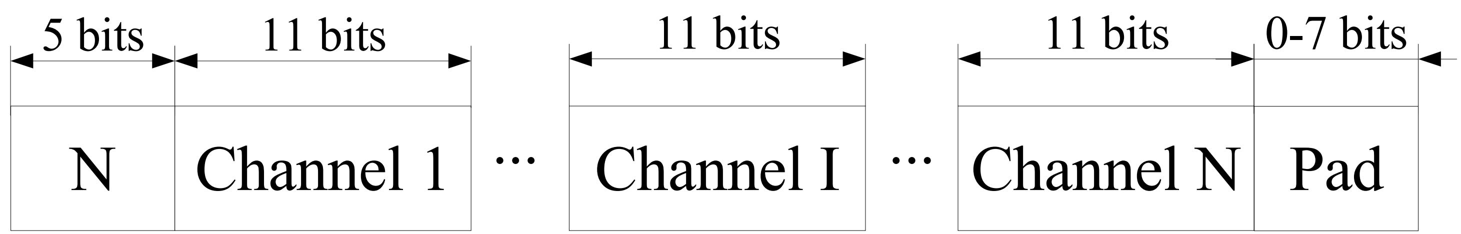
When the size of the resulting binary can be calculated as a function of a generator binary, the translation can be very efficient

```
Res = << X:16 || X:8 <= Bin>>.
              =>
bitsize(Res)  == (bitsize(Bin) / 8) * 16
```

This allows us to preallocate the memory that is needed for the resulting binary

Example: IS-683 PRL

Data Structure



Task:
Create a list of Channels

First ”Padding” Solution:

UPPSALA UNIVERSITET

```
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
   decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
   Acc;
decode(Bin, NumChans, N, Acc) ->
   SkipBefore = (N - 1) * 11,
   SkipAfter = (NumChans - N) * 11,
   Pad = 8 - ((NumChans * 11 + 5) rem 8),
   <<_:5, _:SkipBefore, Chan:11,
     _:SkipAfter,_:Pad>> = Bin,
   decode(Bin, NumChans, N - 1, [Chan | Acc]).
```

Buggy calculation of padding

UPPSALA UNIVERSITET

Correct ”Padding” Solution:

```
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
   decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
   Acc;
decode(Bin, NumChans, N, Acc) ->
   SkipBefore = (N - 1) * 11,
   SkipAfter = (NumChans - N) * 11,
   Pad = (8 - ((NumChans * 11 + 5) rem 8)) rem 8,
   <<_:5, _:SkipBefore, Chan:11,
     _:SkipAfter,_:Pad>> = Bin,
   decode(Bin, NumChans, N - 1, [Chan | Acc]).
```

Expanded solution:

```
decode(Channels) ->
  case Channels of
    <<0:5,_:3>>->
      [];
    <<1:5,X1:11,_:0>>->
      [X1];
    <<2:5,X1:11,X2:11,_:5>>->
      [X1,X2];
    <<3:5,X1:11,X2:11,X3:11,_:2>>->
      [X1,X2,X3];
    <<4:5,X1:11,X2:11,X3:11,X4:11,_:7>>->
      [X1,X2,X3,X4];
    <<5:5,X1:11,X2:11,X3:11,X4:11,X5:11,_:4>>->
      [X1,X2,X3,X4,X5];
    <<6:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,_:1>>->
      [X1,X2,X3,X4,X5,X6];
    <<7:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,_:6>>->
      [X1,X2,X3,X4,X5,X6,X7];
    <<8:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,_:3>>->
      [X1,X2,X3,X4,X5,X6,X7,X8];
    <<9:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,_:0>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9];
    <<10:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,_:5>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10];
    <<11:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,_:2>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11];
    <<12:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,_:7>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12];
    <<13:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,_:4>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13];
    <<14:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,_:1>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14];
    <<15:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,_:6>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15];
    <<16:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,_:3>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16];
    <<17:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,_:0>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17];
    <<18:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,_:5>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18];
    <<19:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,_:2>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19];
    <<20:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,_:7>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20];
    <<21:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,_:4>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21];
    <<22:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,_:1>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22];
    <<23:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,_:6>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23];
    <<24:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,_:3>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24];
    <<25:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,_:0>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25];
    <<26:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,_:5>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26];
    <<27:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,_:2>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27];
    <<28:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,_:7>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28];
    <<29:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,_:4>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29];
    <<30:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,X30:11,_:1>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29,X30];
    <<31:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,X30:11,X31:11,_:6>>->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29,X30,X31]
  end.
```

```
<<3:5,X1:11,X2:11,X3:11,_:2>> ->
     [X1,X2,X3];
```

Smart, but inefficient solution

UPPSALA UNIVERSITET

```
decode(<<N_channels:5, Alignment_bits:3, Tail/binary>>) ->
  decode2(N_channels, <<Alignment_bits:3, Tail/binary, 0:5>>).

decode2(0, _) ->
  [];
decode2(N, <<C:11, A:5, T/binary>>) ->
  [C|decode2(N-1, <<A:5, T/binary, 0:3>>)].
```

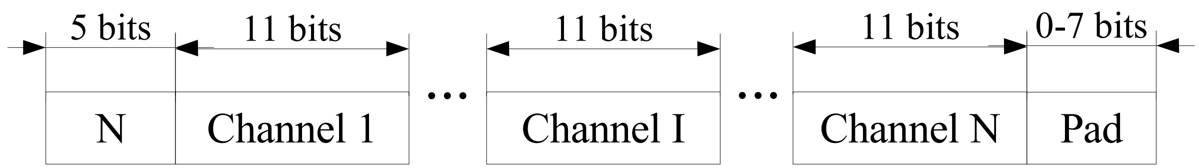
Avoids complicated padding calculations,
at the cost of recreating the binary in each iteration.

Using Flexible binaries

Since flexible binaries can represent bit streams properly and leads to a natural solution

```
decode(<<N:5, Channels:(11*N)/binary,_/binary>>) ->
  decode2(Channels).

decode2(<<C:11, T/binary>>) ->
  [C|decode2(T)];
decode2(<<>>) ->
  [].
```

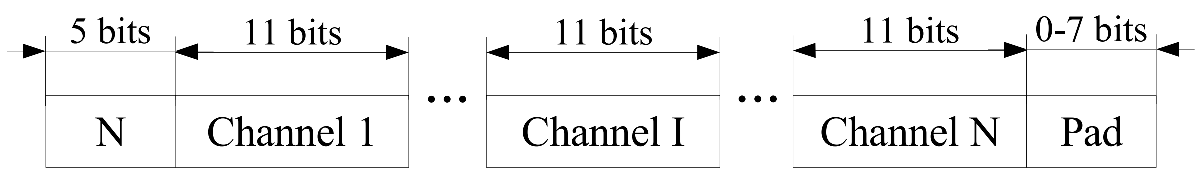


UPPSALA
UNIVERSITET

Extended comprehensions

Using extended comprehensions and flexible binaries we can solve the problem in two lines:

```
decode(<<N:5, Channels:(11*N)/binary,_/binary>>) ->
  [X || X:11 <= Channels].
```



UPPSALA UNIVERSITET

Succintness of flexible binaries

- as measured in line counts

Program in	C	Java	Erlang (R10B)	Erlang (this)
keep 0XX	51	33	14	2
μ-law encode	30	25	25	13
UU-decode	19	14	10	2

μ-law encode - Compresses sound
keep 0XX - Keeps bit-triples that start with 0
UU-decode - Decodes UU-encoded binaries

UPPSALA UNIVERSITET

Conclusion

- Introducing bit-level binaries makes it easy to represent bit streams as binaries
- This makes it possible to write high level specifications of operations on bit streams
- Extended comprehensions allow for powerful manipulation of binaries
- Together these extensions make binaries as easy to use as other datatypes in Erlang such as tuples and lists
- The extensions we propose are backwards compatible
- They will probably be included in the R11 release of Erlang/OTP

UPPSALA UNIVERSITET

Future Work

- A standard library for dealing with binaries
- A better representation of binaries to avoid quadratic complexity when appending binaries
- New compilation techniques which allow for in-place updates of binaries

Adapting BIF:s to bit-level binaries

`size(Bin)`

- should return the minimal number of bytes needed to represent the binary.

`bitsize(Bin)`

- new bif which returns the size in bits

`binary_to_list(Bin)`

> the following should hold:

```
Bin == list_to_binary(binary_to_list(Bin))
```

UPPSALA UNIVERSITET

binary_to_list(Bin)

Desired property:

```
Bin == list_to_binary(binary_to_list(Bin))
```

```
binary_to_list(<<X:8,Rest/binary>>) ->
  [X|binary_to_list(Rest)];
binary_to_list(<<>>) ->
  [];
binary_to_list(Bin) when is_binary(Bin) ->
  [Bin].
```

gives:

```
[0,0,<<0:4>>] == binary_to_list(<<0:20>>)
```