BEAM generic operations encoding

January 9, 2012

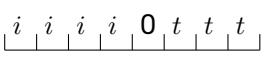
The encoding of the assembler output into a 'Code' chunk of a BEAM file is performed by functions in `beam_asm` module. `encode_op(Name, Args, Dict)` encodes a single generic operation into a binary. A contatenation of such binaries is the data that follows a 16-byte header of the chunk. `Name` is one of:

- label
- func_info
- int_code_end
- call
- call_last
- call_only
- call_ext
- call_ext_last
- bif0
- bif1
- bif2
- allocate
- allocate_heap
- allocate_zero
- allocate_heap_zero
- test_heap
- init
- deallocate
- return
- send
- remove_message
- timeout
- loop_rec
- loop_rec_end
- wait
- wait_timeout
- is_lt
- is_ge
- is_eq
- is_ne
- is_eq_exact
- is_ne_exact
- is_integer
- is_float
- is_number
- is_atom
- is_pid
- is_reference
- is_port
- is_nil
- is_binary
- is_list
- is_nonempty_list
- is_tuple
- test_arity
- select_val
- select_tuple_arity
- jump
- 'catch'
- catch_end
- move
- get_list
- get_tuple_element
- set_tuple_element

- put_list
- put_tuple
- put
- badmatch
- if_end
- case_end
- call_fun
- is_function
- call_ext_only
- bs_put_integer
- bs_put_binary
- bs_put_float
- bs_put_string
- fclearerror
- fcheckerror
- fmove
- fconv
- fadd
- fsub
- fmul
- fdiv
- fnegate
- make_fun2
- 'try'
- try_end
- try_case
- try_case_end
- raise
- bs_init2
- bs_add
- apply
- apply_last
- is_boolean
- is_function2
- bs_start_match2
- bs_get_integer2
- bs_get_float2
- bs_get_binary2
- bs_skip_bits2
- bs_test_tail2
- bs_save2
- bs_restore2
- gc_bif1
- gc_bif2
- is_bitstr
- bs_context_to_binary
- bs_test_unit
- bs_match_string
- bs_init_writable
- bs_append
- bs_private_append
- trim
- bs_init_bits
- bs_get_utf8
- bs_skip_utf8
- bs_get_utf16
- bs_skip_utf16
- bs_get_utf32
- bs_skip_utf32
- bs_utf8_size
- bs_put_utf8
- bs_utf16_size
- bs_put_utf16
- bs_put_utf32
- on_load
- recv_mark
- recv_set
- gc_bif3
- line

`Args` is the list arguments of the operation. The length of the list is the arity of the operation. The arity together with the `Name` dictate the opcode. The opcode is the first byte of the binary. The opcode is followed by individually encoded arguments. Arguments are encoded as (a sequence of) tagged values.

Integers in the 0..15 range are encoded in a single byte as



where $iiii$ is the encoded value, ttt is the tag.

Integers in the 16..2047 range are encoded as two bytes:

$\underbrace{i\ i\ i\ 0\ 1\ t\ t\ t}$ $\underbrace{j\ j\ j\ j\ j\ j\ j\ j}$ where $iiijjjjjjj$ is the encoded value.

Integers larger than 2047 (and any negative integers) are first converted into a sequence of bytes the most significant first. In general, for large integers the length of the sequence is encoded first followed by the bytes themselves. If the length is within the 2..8 range then it is encoded as

$\underbrace{l\ l\ l\ 1\ 1\ t\ t\ t}$ where $lll+2$ is the length of the byte sequence that follows.

The encoding of the length of a longer byte sequence starts with

$\underbrace{1\ 1\ 1\ 1\ 1\ t\ t\ t}$ followed by an encoding of the length as a separate value tagged with the tag u. The actual bytes go next.

Using the above scheme arguments are encoded using the following rules:

1. `{x,X}` and `{y,Y}` use tags x and y and encode the register.
2. `{atom,A}` uses tag a to encode the atom index.
3. `{integer,I}` uses tag i.
4. `nil` is represented as a value 0 encoded with the tag a.
5. `{f,L}` uses tag f to encode the label index.
6. `{string,S}` uses tag u to encode the offset into the string table.
7. `{extfunc,M,F,A}` encodes the index in the import table using u tag.
8. `Uint` is encoded using u tag.
9. `{field_flags,Flags}` uses u tag to encode bit-masked flags.

The rest of argument types use extended scheme that start with the subtype encoded using tag z:

1. `{float,F}` uses subtype 0 followed by 8-byte floating-point value.
2. `{list,List}` uses subtype 1 followed by the length of the list encoded with tag u followed by individually encoded list items.
3. `{fr,R}` uses subtype 2 followed by the floating-point register encoded with tag u.
4. `{alloc,List}` uses subtype 3 followed by two pairs of values encoded with tag u. If the first tagged value of the pair is 0 then the second value is the number of words. If the first value is 1 then the second is the number of floats.

5. `{literal,Lit}` uses subtype 4 followed by the index into the literal table encoded with tag u.