Ling instructions packing

January 11, 2012

Unpacked argument types

The unpacking of the code of a BEAM file results if a list of generic instructions. Types of arguments of generic instructions are given in the Table 1.

Tag	Type
N	unsigned integer
{a,A}	atom
{i,I}	integer
nil	empty list
{x,X}	register
{y,Y}	stack slot
{f,F}	label ref
{float,V}	floating-point number
{list,L}	list of pairs
{fr,R}	floating-point register
{alloc,S}	heap size spec
{literal,Q}	any term

Table 1 Simple generic BEAM instruction argument types

Untagged unsigned integer arguments sometimes require special treatment and they may represent an index into a table. Resolving of such arguments usually require changes to the instruction arity. The exhaustive list of such argument resolutions are given below:

- call_ext A N → call_ext {a,M} {a,F} A
- call_ext_last A N D → call_ext_last {a,M} {a,F} A D
- call_ext_only A N → call_ext_only {a,M} {a,F} A
- bif0 N u1 → bif0 {a,M} {a,F} 0 u1
- bif1 {f,F} N u1 u2 → bif1 {f,F} {a,M} {a,F} 1 u1 u2

- bif2 {f,F} N u1 u2 u3 → bif2 {f,F} {a,M} {a,F} 2 u1 u2 u3
- gc_bif1 {f,F} u0 N u1 u2 → gc_bif1 {f,F} u0 {a,M} {a,F} 1 u1 u2
- gc_bif2 {f,F} u0 N u1 u2 u3 → gc_bif2 {f,F} u0 {a,M} {a,F} 2 u1 u2 u3

u0-u3 are arguments of arbitrary type. Such argument resolution makes the import table redundant. The export and fun tables are kept.

The range of values for arguments of various types observed after unpacking 1661 BEAM files are given in the Table 2.

Tag	Range
N	0-33832
{x,X}	0-1023
{y,Y}	0-47
{i,I}	many-many digits
{f,F}	0-5861
{list,L}	(size) 2-990
{fr,R}	0-13

Table 2 Observed argument ranges

Analysis of BEAM file listings indicate that generic instructions allow multiple types for their arguments. Such combinations of argument types represent compound types. The compound types observed are given in the Table 3.

Tag	Simple types
s	a/float/i/literal/nil/x/y
d	x/y

Table 3 Compound argument types

Compound types not compatible with s or d should be resolved by making multiple variants of a generic instruction that uses them. The following instructions require such variants: allocate_heap, allocate_heap_zero, test_heap, bs_restore2, bs_save2, bs_init2, bs_init_bits, fmove.

Note that some simple types appear exclusively as a part of a compound type, such as {float,F}. An integer simple type appeared standalone only once in is_function2 instruction. The case is changed to a more appropriate unsigned type.

The argument types of all generic BEAM instructions can be looked up in the implementation of the `disasm:validate/1` function.

Argument specialisation

During argument specialisation phase some argument types change. {alloc,S} type goes away and is replaced with an unsigned integer as at this stage the number of heap words to store a floating-point value is known. The size of small integer is also known and {i,I} type is split into {smallint,I} and {bigint,I}. {x,255} results in error and {x,1023} is mapped to {x,255}. This limits the number of function arguments to 255. Such mapping is reasonable as the maximum number of "live" registers is 255.

All possible argument types after argument specialisations are summarized in the Table 4.

Tag	Type
N	unsigned integer
{a,A}	atom
{smallint,I}	small integer, $-268435456 \leqslant I \leqslant 268435455$
{bigint,I}	big integer, $I < -268435456, I > 268435455$
nil	empty list
{x,X}	register, $0 \leqslant X \leqslant 255$
{y,Y}	stack slot
{f,F}	label ref
{float,V}	floating-point number
{fr,R}	floating-point register
{literal,Q}	any term
{list,L}	list of pairs
{e,N}	import tab ref (appears after transforms)

Table 4 Specialised argument types

Instruction transformations

Generic instructions may require transformations for the following reasons:

1. To remove a redundant argument

During instruction transformation two new types {e,N} and {bif,N} are introduced. {e,N} represents an index into an import table. {bif,N} represents a reference to a C-function implementing the BIF. See Table 4. In addition a {list,L} argument, if present, are replaced a {u,N} argument containing the length of the list. The list items are set aside in a trailer table. Also, instructions, such as `fmove`, that may have arguments of incompatible types are splitted.

Ling instruction table

All Ling instructions are listed in `iops.tab`. Argument types in the file is a subset of specialised types and two aggregate types as indicated in the Table 5.

Each Ling instruction is represented by a single tuple in the file. Note that all simple types comprising `s` type can be stored a tagged term value.

Selector	Tag(s)
u	N
a	{a,A} (func_info only)
x	{x,X}
y	{y,Y}
f	{f,F}
fr	{fr,R}
e	{e,N}
b	{bif,N}
s	{a,A}, {float,F}, {smallint,I}, {bigint,I}, {literal,L}, nil, {x,X}, {y,Y}
d	{x,X}, {y,Y}

Table 5 Argument types of the Ling instruction set

Ling instruction variants

Each Ling instruction may have many variants depending on prevalent argument values and their packing. All variants of a given instruction have exactly the same semantics. For instance, the statistics show that in 50% of cases the first argument of an instruction is {x,0}. Thus it may be reasonable to create an instruction variant that assumes that the first argument is {x,0}. Such variant may run faster and take less memory space. Instruction variants are inferenced automatically using the instruction set in `iops.tab` and instruction statistics generated by `make_iopstats`.

Each instruction variant has a particular representation of its arguments when the code is being executed. Upon each instruction dispatch arguments are unpacked from this representation. Possible representations of arguments of a given type are given in the Table 6. The packing code should be enough to generate the argument access code. Note that some arguments can be both read and written. All items of the trailer table have either t or p representation to provide for indexed access to the table.

The atom type {a,A} cannot be packed into smaller size because non-standard atom indices are known at runtime only.

Automatic inference of instruction variants

Automatic inference of the optimal set of instruction variants is based on the analysis of disassembly listings of a large body of BEAM files. The frequency of certain values of arguments and all possible packings of them are taken into account. The criterion for adding a new instruction variant is that the size of the code implementing the instuction is smaller than the associated savings in the size of loaded modules. The criterion is based on the assumption that the smaller code is usually faster.

The automatic inference procedure is implemented by iopvars_gen script.

Tag	Packing	Size
N	u8	8
	u32	32
{a,A}	t	32
{smallint,I}	i8	8
	t	32
{bigint,I}	t	32
nil	(implict)	0
	t	32
{x,X}	x8	8
	t	32
{y,Y}	y8	8
	t	32
{f,F}	f	32
{float,V}	t	32
{fr,R}	fr	8
{literal,Q}	t	32
{e,MFA}	e	32
{bif,MFA}	b	32
{fu,N}	fu	32
{str,N}	str	32
{catch,F}	ca	32

Table 6 Argument type packing

The script takes many minutes to run and thus it is not invoked by make directly while building the kernel. Use 'make iopvars' to rebuild instruction variants

Loader-specific code representation

The loader-specific code representation or "specs" is used for static and dynamic code loading. Specs has all labels resolved and code offsets substituted throughout instead of label numbers. Each "spec" corresponds to a single word of the runnable code. A spec may encode multiple arguments. All possible specs are presented in the Table .

Binary module format

The code is stored and transferred over the network using the Ling binary module format. The Ling binary module format resembles the format of BEAM files. The major difference is that the Ling binary module format is much closer to the runnable code.

Pattern	Meaning
N	unsigned 32-bit integer may contain multiple arguments
{tag_int,I}	immediate integer term
{reg_as_term,X}	immediate term containing a register reference
{slot_as_term,Y}	immediate term containing a slot reference
nil	immediate nil term
{f,none}	undefined label reference ({f,0})
{f,Off}	a reference to the code location
{export,N}	a reference to export_t N is the index into the import table
{bif,N}	a bif_func_t reference N is the index into the import table
{fu,N}	a reference to fun_entry_t N is the index into the lambda table
{str,Off}	a reference into a string table of the module
{catch,N}	immediate term containing a catch context N is the local catch index
{literal,N}	literal pointer term N is the index into the literal table
{atom,N}	immediate atom term N is the index into the atom table

Similarly to the BEAM file format, the Ling binary module format is based on Interchange File Format (IFF). The module is represented as a sequence of chunks each identified with a unique four character name.

TODO