Encoding for Objects Matters

Dave Mason

Toronto Metropolitan University, Toronto, Canada

Abstract

In dynamically typed programming languages, variables and expressions don't have types associated with them; rather, values have types. This means that choosing the correct machine instructions to operate on the data requires run-time dispatch on the types associated with the values.

The simplest encoding is to simply store every object in a memory heap, where they are tagged with their type. This puts tremendous pressure on the memory system and the memory allocation system. The most complex encoding will put everything possible into immediate values to minimize this pressure.

In this paper we outline the relative costs and needs of the various models.

Keywords

Dynamically-Typed Languages, Runtime Systems, Value Encoding

1. Introduction

In statically-typed programming languages, the compiler knows at compile time the type of all of the variables and expressions, and can make sophisticated choices about the use of registers and the types of instructions to manipulate those values. However in dynamically typed languages like Smalltalk, Lisp, Python, Self, et cetera, variables and expressions have no pre-defined type; the current values in the variables determine the instructions to use. This forces more reference to the values, and the encoding of those values matters a great deal.

It's pretty difficult to wrap one's head around just how different latencies are for various parts of a computer system because we don't commonly deal in nanoseconds and microseconds. To help understand the scale it's helpful to express these times as multiples of instruction step times. This lets us step into the perspective of the CPU which can execute at least one instruction every step time unless it is waiting for data to be delivered by various types of storage, or for the result of a previous operation. We can see these different types of storage in Table 1.

Storage Type	Instruction Time Multiples	**Real Time Scale**
Single CPU Instruction (at 3 GHz)		0.3 nSec
Registers (storage for active instructions)	1x to 3x	0.3 to 1 nSec
Memory Caches	2x to 12x	0.7 to 4 nSec
Main System Memory (RAM)	30x to 60x	10 to 20 nSec
NVMe SSD	300,000x or more	100 to 200 uSec

Table 1
Relative time scale for memory access

Modern processors fetch instructions, decode them, and put them in the execution pipeline. This means that unconditional branches and calls to fixed addresses have almost no cost as long as the pipeline remains relatively full. Similarly, conditional branches are relatively inexpensive, because of branch prediction hardware and sometimes decoding along more than one potential execution path. What *are* expensive are indirect branches because only after the data is available can the fetch-decode

IWST'25: International Workshop on Smalltalk Technologies Gdansk, Poland July 1st to 4th, 2025 Co-located with ESUG 2025

dmason@torontomu.ca (D. Mason)
https://sarg.torontomu.ca/dmason/ (D. Mason)
0000-0002-2688-7856 (D. Mason)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

logic start adding to the pipeline. Pipelines can execute instructions out-of-order and sometimes in parallel, as long as dependencies are observed.

There are many options in terms of dynamically-typed language implementation, from byte interpreters to threaded execution to JIT'ed native code. However the encoding choices will affect all of the implementations, to varying degrees.

Paper outline. First, in section 2 we catalog the considerations that affect performance for a given encoding. Section 3 discusses 5 possible encodings from the perspective of those considerations. Section 4 describes an experiment to compare those encodings. Finally, section 5 draws some conclusions and identifies aspects that are beyond the scope of this paper.

2. Encoding Considerations

Encoding values, particulaly for modern architectures has 3 considerations:

Determining types. Since operations must be parameterized at run time by the types of the values, the first step is to access the types of the values. If the types are in memory, this can cause a pipeline stall until they can be loaded.

Accessing values. In order to perform operations, the values must be made available to the CPU. As seen in Table 1, accessing values in memory is an order of magnitude slower than accessing them in registers, and even if the values happen to be in cache they will be several times slower than in registers.

Supporting Memory Management. Dynamically-typed languages invariably have an automatic memory management system, whether reference counting or garbage collection. Objects need to be allocated, and removed when no longer needed. While memory management has improved greatly over the decades, there remains a significant cost to both actions, and the more objects allocated in memory, the greater the cost. We use the term **churn** to refer to the allocation and subsequent collecting of in-memory objects.

3. Catalog of Encodings

In an Object-Oriented language, objects can be categorized as mutable or immutable. Mutable obects must be allocated in memory and have references passed so that when the object is mutated it is seen as such by all other references. These allocations are typically on a heap and have a header word. Without loss of generality, we will use 64-bit words and Zag encoding throughout this paper. For example the Zag memory object looks like:

```
|n|n|n|n|n|n|n|n|n|n|n|n|a|a|a|a|f|f|f|f|f|f|f|f|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|h|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
|                                                            fields                                                             |
|                                                              ...                                                              |
```

Here, n is the 12-bit number of words the object occupies; a is the 4-bit age of the object; f is the 8-bit format of the object; h is the 24-bit identity hash; and c is the 16-bit class of the object. It is followed by the fields of the object. The reference to the object is the address of the header word.

There are many possible ways to encode the range of immutable values that arise in normal calculations. In this section we discuss 6 possible encodings that encompass the range from all values being in memory, to treating as many things as immediate as possible. We examine each from the perspective of the three considerations outlined in section 2. Table 2 summarizes the important encoding considerations for the encodings discussed below.

Encoding	Determining type & Accessing Value	Memory Considerations
In Memory	memory access	every created immutable requires allocation
SmallInteger Cache	memory access	common integers avoid allocation
Tag SmallInteger	direct for integer	integers avoid allocation
Spur	direct for integer, most floats	integer and most float values avoid allocation
NaN Encoding	direct for most immutables	most immutables avoid allocation
Zag	direct for most immutables	more immutables avoid allocation

Table 2
Considerations of Encodings

3.1. Every Object in Memory

This is the simplest model. Every object - mutable or immutable - is encoded as a word-aligned address:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	0	0	0

Here the low 3 0-bits are part of the address, and the high 16 0-bits are hardware limitations.

Determining the value type requires a memory access, as does accessing the actual value. Interoperation with foreign functions is complicated. The result of every operation must be allocated to memory. This will create a huge amount of churn, although most of the values wil be very short-lived. Immutable values nil, $true$, $false$, and ASCII Characters are normally allocated at fixed addresses to gain some performance by allowing hardcoded memory address comparisons and we assume this convention.

3.2. Every Object in Memory with SmallInteger Cache

This is the same encoding as above, with a simple optimization of statically defining a set of small integers (say -7..100). This means that if a result is in this pre-allocated set, then no allocation or collection is required.

This reduces churn because the commonly created objects can be easily identified and no allocation or collection of them is required. It also may get some memory-cache advantage as these values will typically be in cache.

3.3. Tag $SmallInteger$

Early Lisp and Smalltalk implementers noticed that the vast majority of values that were created were for $SmallInteger$ objects. Most such systems tag small integers with a 1 bit tag (either in the low bit leaving all other addresses natural by aligning all objects on at least a word boundary, or (less commonly) using the sign bit). This encoding was so important that the SPARC architecture had basic arithmetic instructions that checked that the low bit of the parameters to verify they were flagged as integers.

Here $SmallInteger$ values are encoded as:

s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	1

and everything else remains encoded as the address:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	0	0	0

Note that the low bit distinguishes the integer value from the reference.

A small drawback of this encoding is that 1 bit of precision is lost, but most such systems move automatically to big integers with unbounded precision on overflow. Furthermore, on modern, 64-bit systems that one bit of precision is fairly irrelevant.

3.4. Tag *SmallInteger*, *Character*, *Float* (Spur Encoding)

The Spur[1] encoding cleverly extends the 1-bit tag to 3 bits and in addition to *SmallInteger* and general (memory) objects, supports encoding for Unicode characters and a subset of *Float*.

The default encoding for everything not described below remains encoded as the address:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	0	0	0

The encoding for *SmallInteger* is:

s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	0	0	1

The encoding for *Character* is:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	0	1	0

The encoding for *ImmediateFloat* is:

e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	s	1	0	0

Examining the low 3 bits of an object, a simple *and* instruction disambiguates the encodings. Only the pointer tag is a natural value - all the others require some decoding before use.

The *ImmediateFloat* value maintains the full mantissa from the 64-bit IEEE-754 floating point (see section 3.5), but reduces the exponent range, so it can represent all 64-bit values in the range of ± `5.87d-39` to ±`6.80d+38`. The exact algorithms for converting between *ImmediateFloat* and IEEE-754 are detailed in [2] but require 7-8 instructions. If a value is not encodable, it is stored in memory.

3.5. NaN Encoding

NaN encoding utilizes the large number of code points in the IEEE-754 floating point encoding that do not represent valid floating-point numbers, by using those bit patterns to represent values of other types, including pointers and *SmallInteger*. This encoding has been used by Spidermonkey[3], and was the originally planned encoding for Zag Smalltalk[4]. The IEEE-754 64-bit floating point number is formatted as:

s	e	e	e	e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m

When all the exponent bits are 1 and the mantissa is non-zero, it is considered not-a-number or NaN. This gives a huge number of potential bit patterns that can be used to encode other values. Our overall encoding looks like:

1	1	1	1	1	1	1	1	1	1	1	1	t	t	t	t	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

where the `t` bits are a 4-bit tag where 0 encodes general references; the next 10 values encode various kinds of immediate *BlockClosure* values; the next tag (`0xB`) is encoded as:

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

where the `c` is the 16-bit class and encodes *Symbol*, *True*, *False*, *Character*, *UndefinedObject* with the `x` bits being the extra information for the class. Finally, *SmallInteger* is encoded as:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

where the `s` is the 50-bit integer value.

This encoding recognizes all *Float* values with a single comparison and they require no conversion before use. All the other values, including references, require more manipulation to extract the class, and access the value. It can encode all immutable values with up to 32 bits of extra data. It can also encode some of the same *BlockClosure* values as described in Section 3.6, but with much-restricted parameters.

3.6. Tag Most Possible Values (Zag Encoding)

Zag Smalltalk[5] now uses a modified-Spur encoding. It is modified in 3 ways:

1. it uses all 8 possible values of the low 3 bits;
2. all non-float, non-reference objects are encoded with an auxiliary tag;

3. it supports a larger range of float values, with fewer instructions to decode/encode.

The default encoding for everything not described below (and the *nil* object) remains encoded as the address:

```
0000000000000000aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa000
```

The encoding for 56-bit *SmallInteger* is:

```
ssssssssssssssssssssssssssssssssssssssssssssssssssssssss10000001
```

Which is a special case of the encoding for immediate classes (including *Symbol*, *True*, *False*, *UndefinedObject*, and *Character*, is:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxccccc001
```

where the `c` encodes up to 32 special classes. There are several classes that encode special *BlockClosure* subclasses that are common in code; for example the immediate closure that does a non-local return of an instance variable looks like:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaiiiiiiii00010001
```

where a is the address of the *Context* that we are to return from; `i` is the 8-bit field number in the *self* object for that *Context*. The encoding for *ImmediateFloat* is similar to Spur, except there are 6 used values for the low 3 bits:

```
eeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmsEEE
```

where `e` are the low 8 bits of the exponent and `m` are the bits of the mantissa (like Spur), but the `E` are the high 3 bits of the exponent offset by 2. This increases the range of the representable floating point values to all 64-bit floats smaller than ± `2.68d154`, including 0 and the small fractional values. Examining the low 3 bits of an object disambiguates the encodings. Only the pointer tag is a natural value - all the others require some decoding before use.

4. Experimental Design

Zag Smalltalk was originally going to use NaN-encoding, and even after the decision was made to use the Zag-encoding described in Section 3.6, the code for both was carried in parallel, although bit-rot set in for the NaN encoding as additional features were implemented. However the question of the best encoding has always remained.

There is a single enumeration in the Zag configuration module that determines which encoding will be used, so in principal, Zag can be compiled with any one of these encodings. Since the decision was made to run this experiment, much work has been done to factor out the part of the code that is aware of the underlining encoding into a single module. This has been a more significant undertaking than originally understood as encoding assumptions appeared in many parts of the codebase. The refactoring is nearing completion, but was not available for this paper.

The experiment is to run the fibonacci algorithms shown in Listing 1 and Listing 2.

Listing 1: fibonacci1

```
fibonacci1

        self < 2 ifTrue: [ ^ self ].
        ^ (self - 1) fibonacci1 + (self - 2) fibonacci1
```

Listing 2: fibonacci2

```
fibonacci2

        ^ self < 2 ifTrue:  [ self ]
                   ifFalse: [ (self - 1) fibonacci2 +
                              (self - 2) fibonacci2 ]
```

Each will be run 4 times with the combinations of with/without any inlining and $SmallInteger$ and $Float$ as receiver. The plan is to evaluate $30 fibonacci$ and $30.0 fibonacci$, however this may not be tractable with the memory-allocating encodings. This will exercise the integer and floating-point paths, and the version with no inlining will exercise the $BlockClosure$ creation in both listings and the non-local return in Listing 1.

4.1. Hypothesis

Zag design features other than the choice of encoding will influence some of the runtimes. In particular, Zag has a three-tier memory system: stack, nursery, and global heap, and references are only allowed to go to the right. None of the test runs should hit the global heap, and the nursery is a copying collector, so should be quite efficient.

Inlining is a fundamental part of the Zag code generator, and applies to the threaded-execution and the eventual JIT (the threaded code version drives the JIT). However, inlining is not done on a per-selector basis, so "no-inlining" does not include inlining that most Smalltalk systems do (`ifTrue:`, etc.), so that version will create the nonlocal-return closure in the first listing and both closures in the second version and will send the `ifTrue:` and `ifTrue:ifFalse:` messages to the boolean result of the comparison.

The inlined version, on the other hand will inline everything except the recursive calls to $fibonacci$. This also means that no context will be created on any path that doesn't perform a send.

For some runtime systems the $BlockClosure$ objects would have to be allocated on the heap, but Zag actually allocates them on the stack. However, this still is a moderate amount of work, so the encodings that produce immediate values will be faster. The non-local return block is potentially particularly tricky, because it references the context from which it must return, so if it were allocated in the nursery it would force the context to also spill to the nursery. This would lead to significant churn in the first 4 cases. The last 2 encodings create immediate values for some of these blocks, so do not force any context spilling, and may, in fact, create no nursery allocations at all.

4.1.1. Every Object in Memory

The result of every addition will allocate in the nursery. This will create a great deal of churn for the numbers.

4.1.2. Every Object in Memory with SmallInteger Cache

The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster. The floating-point version will have no speedup.

4.1.3. Tag $SmallInteger$

Now all integers (for the test) have immediate values, so the integer tests should speed up significantly. The floating-point version will have no speedup.

4.1.4. Tag $SmallInteger$, $Character$, $Float$ (Spur Encoding)

Because Spur encodes immediate values for all of the generated floating-point values, the $Float$ version of this should run significantly faster. The integer version should experience no speedup.

4.1.5. NaN Encoding

NaN encoding has the floating-point already in the correct format, therefore this should have the fastest floating point version. The integer versions may be somewhat slower because more manipulation is required to retrieve the integral value. Because the blocks are immediate values, the “no-inlining” version should speed up noticeably.

4.1.6. Tag Most Possible Values (Zag Encoding)

The Zag native format should be slightly slower than Spur for inlined integer. It should also be a bit faster than Spur for floating point, because the decoding is a couple of instructions faster. The floating-point runs should be a bit slower than for NaN-encoding.

5. Conclusion

Some of the encodings described in this paper have been common lore for decades, and highly applicable to 32-bit machines. With the advent of 64-bit architectures, the potential design space has broadened significantly. Other encodings are much more recent. The value of this paper is to describe the spectrum of encodings, and to explore how the encoding interacts with the computer architecture.

Future Work

It is unfortunate that the actual experimental data is not yet available.

The proposed benchmark, although useful to demonstrate some of the macro properties of various encodings, does not fully exercise the potential of the Zag encoding. It would be good to run a more realistic benchmark such as Delta Blue or Richards.

A complex issue is how much jitted code can capture the flow of type information. The Zag approach is to handle this as much as possible at the inlining stage, inferring type information, but it would be interesting to look at how dynamic information can be made to flow.

References

[1] C. Béra, E. Miranda, A bytecode set for adaptive optimizations, in: International Workshop on Smalltalk Technologies, Cambridge, United Kingdom, 2014. URL: https://inria.hal.science/hal-01088801.

[2] C. Béra, 64 bits Immediate Floats, 2018. URL: https://clementbera.wordpress.com/2018/11/09/64-bits-immediate-floats/, [Online; accessed 2025-08-10].

[3] Mozilla, Spidermonkey javascript engine, 2025. URL: https://spidermonkey.dev/, [Online; accessed 2025-04-08].

[4] D. Mason, Design principles for a high-performance smalltalk, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST ’22, 2022.

[5] Zag smalltalk, 2025. URL: https://github.com/Zag-Research/Zag-Smalltalk, [Online; accessed 2025-04-08].