Encoding for Objects Matters

Dave Mason

Toronto Metropolitan University, Toronto, Canada

Keywords
Dynamic Languages, Runtime Systems, Value Encoding

In dynamic 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 everything in memory (where it is tagged with a 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.

Requirements

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 milliseconds. To help understand the scale it's helpful to step through a thought experiment where we stretch time by a factor of 3 billion. In this expanded time scale, a 3GHz CPU would complete a single instruction every second. This lets us step into the perspective of the CPU which is waiting for data to be delivered by various types of storage. We can see these different types of storage listed below:

Storage Type	Slowed Time Scale	**Real Time Scale**
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec
Main System Memory (RAM)	30 to 60 seconds	10 to 20 nSec
NVMe SSD	3 to 11 days	100 to 200 uSec

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 causes 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 the table, accessing values in memory is an order of magnitude slower than accessing them in registers, and several factors slower even if the values happen to be in cache

Supporting Memory Management. Dynamic 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.

Catalog of Encodings

Every Object in Memory. This is the simplest model. Accessing the value type requires a memory access, as does accessing the actual value. Interoperation with foreign functions is complicated.

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)

The result of every operation must be allocated to memory. One optimization is for a few immutable values such as nil, $true$, $false$, ASCII Characters, and a set of small integers (say -2..256) to be pre-allocated to save allocation and collection.

Every Object in Memory with Tagged Pointers. On a 64-bit architecture, memory addresses are typically 48 bits. This could allow the use of the top 16 bits to code the type of the values, so that the type would be simply a shift away. This would allow dispatching to the appropriate machine code without memory delays, although an extra and operation would be required before the pointer could be followed.

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). One of the drawbacks of this 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.

Tag $SmallInteger$, $Character$, $Float$. 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 $Float$. Examining the low 3 bits of an object, a coding of 0 is a pointer to a memory object, 1 is a $SmallInteger$, 2 is a $Character$, and 4 is a $Float$. Only the pointer tag is a natural value - all the others require some decoding before use. As with tagging reducing precision on integers, the Spur encoding limits the range of floating point values.

NaN Encoding. NaN encoding utilizes the large number of code points in the IEEE-7544 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 Spridermonkey[2], and was the originally planned encoding for Zag Smalltalk[3]. This encoding supports, within 64 bits, all $Float$ values naturally, as well as 51-bit $SmallInteger$ values, pointers to memory objects, booleans, symbols, characters, as well as several common $BlockClosure$ values.

Tag Most Possible Values. Zag Smalltalk[4] now uses a modified-Spur encoding. It uses the bottom 3 bits of an object: 0 to naturally encode pointers to memory objects; 1 to encode 31 immediate immutable classes; 2-7 to encode a broader range of $Float$ values than encoded by Spur. The immediate classes include 13 kinds of special block closures, including some non-local returns, that are common in existing code.

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

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

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