NaN-boxing for a Lisp implementation

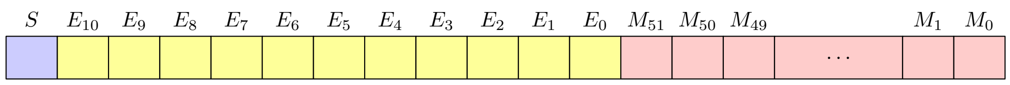
Kjell Post
Datasektionen, Zimmermanska skolan
`post.kjell@gmail.com`

Abstract

NaN-boxing is a clever way of hiding information in the unused bits of the IEEE 754 representation for NaN ("Not a Number"). In this paper we discuss how the unused bits can be used to squirrel away pointers, integers, booleans, etc.

1 Introduction

The IEEE 754 defines how floating point numbers are represented. We will look specifically at the representation of a `double` which is stored in 8 bytes, or 64 bits:

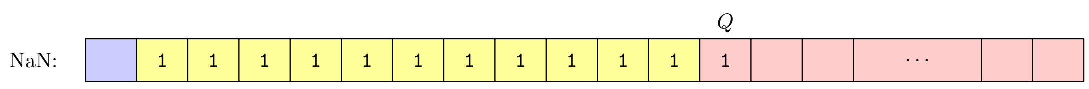

S
E_{10} E_9 E_8 E_7 E_6 E_5 E_4 E_3 E_2 E_1 E_0
M_{51} M_{50} M_{49}
$\cdots$
M_1 M_0

Here, S is the *sign bit* ($= 1$ if the number is negative, 0 otherwise); $E_{10} \ldots E_0$ is the 11-bit *exponent* and $M_{51} \ldots M_0$ is the 52-bit *mantissa*. We won't concern ourselves with how double floating point numbers like `3.14` are actually represented, except for one particular number — or rather "not a number" — *NaN*: when the following C-program is executed it prints `nan`.

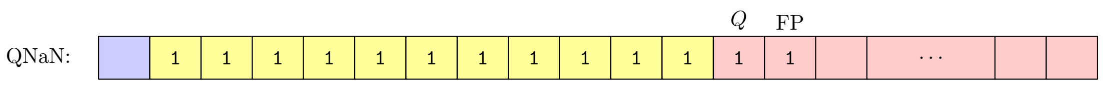
```
#include <stdio.h>

int main() {
    double x = 0.0/0.0;
    printf("%f\n", x);
    return 0;
}
```

Internally, NaN (the value of `x`) is represented as all exponent bits $E_i = 1$ and the first bit in the mantissa $M_{51} = 1$:



All other bits are disregarded. The first bit of the mantissa (henceforth Q) actually distinguishes between two types of NaN: if $Q = 0$ it is a so called "signaling NaN" and should generate an interrupt, although in practice I don't know how or when they are generated. The more common is $Q = 1$ which is called "quiet NaN" and is produced by, e.g., `0.0/0.0`. We will stick to quiet NaNs, and make sure $Q = 1$. So we still have 52 unused bits (1 sign bit and 51 mantissa bits) at our disposal, right? Almost, but not quite. Thanks to Intel's "QNaN Floating-Point Indefinite"which is a reserved value we can't use M_{50} either: we avoid this by setting M_{50} (henceforth FP) $= 1$. So we will stick to the following quiet NaN with $1 + 50$ bits for us to use.



2 Lisp datatypes

What datatypes do we have in a typical Lisp interpreter?

1. S-expressions
2. Symbols
3. Primitives (e.g., `car`, `cdr`, ...)
4. Integers, e.g., `42` or `-17`
5. Strings, e.g., `"foo"`
6. Boolean, i.e., `#t` and `#f`.
7. Arrays
8. "Broken heart", a special symbol used by the garbage collector.

What about doubles? As they are non-NaNs (not not a number) we get them for free simply by not using NaN. And if a NaN would occur in our language, e.g., when computing `(/ 0 0)` we will make sure that *that* NaN is different from our NaN by virtue of bit M_{50} (FP). Two functions in `math.h` are useful here:

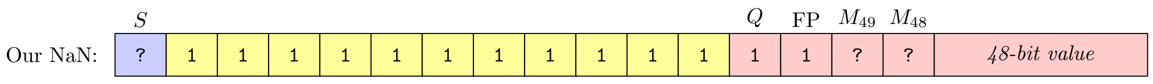
- `int isfinite(x)`

  returns 1 for "normal" numbers, e.g., `2.718`, and 0 for NaN (including our NaNs).
- `int isnan(x)`

  returns 1 for a NaN value (including ours) and 0 otherwise.

How can we distinguish between the system's NaN and our NaN? We would have to examine bits Q and FP to see if they are both set (defined in the next section).

We can then use the three bits $\{S, M_{49}, M_{48}\}$ to encode type information and still have $M_{47} \dots M_0$ for data: it turns out that 48 bits is enough to store an address because even on a 64-bit architecture, no more than 48 bits are actually used[1].



Datatype	S	M_{49}	M_{48}
Symbol	0	0	0
Primitive	0	0	1
Integer	0	1	0
String	0	1	1
S-expression	1	0	0
Boolean	1	0	1
Array	1	1	0
Broken heart	1	1	1

3 Join the union

We are going to have to interpret the double floating point numbers as 64-bit integers in order to do some bit fiddling. A convenient way to look at the 64 bits is to introduce a union:

```
typedef union {
  double as_double;
  uint64_t as_int;
} VALUE;
```

[1]48 bits is enough to address 256 terabytes.

We declare a `VALUE` in one of two ways, e.g.,

```
VALUE v0 = { .as_double = 3.14 };
VALUE v1 = { .as_int = 0x7ffd000000000000 };
```

We can then introduce some macros to make life easier. First, a macro that returns 1 if it is one of our special NaN values by checking that the exponent, Q and FP are all set:

```
#define OUR_NAN(v)     ((v.as_int & 0x7ffc000000000000) == 0x7ffc000000000000)
```

The mask that targets the 48-bit address and the macro that selects it are

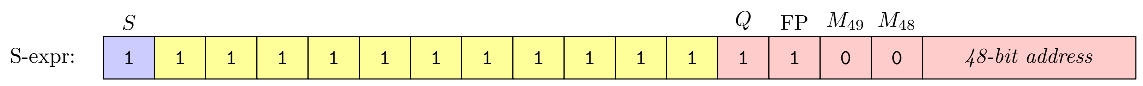
```
#define NAN_MASK       0xffff000000000000   /* 1 11111111111 1111 ... 0000 */
#define NAN_VALUE(v)   (v.as_int & (~NAN_MASK))
```

To compare two values, we simply compare the 8-byte integers:

```
#define EQ(v1, v2)     ((v1).as_int == (v2).as_int)
```

3.1 S-expressions

For S-expressions we use $\{S, M_{49}, M_{48}\} = \{1, 0, 0\}$:



```
#define SEXPR_MASK     0xfffc000000000000 /* 1 11111111111 1100 address */
#define IS_SEXPR(v)    ((v.as_int & NAN_MASK) == SEXPR_MASK)
#define MAKE_SEXPR(p)  { .as_int = (uint64_t) (p) | SEXPR_MASK };
```

One special value is the S-expression

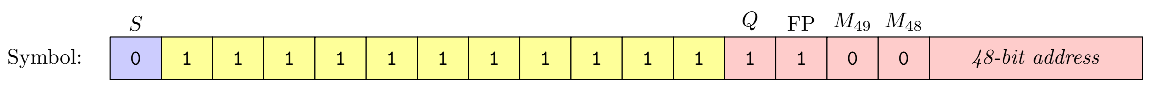
```
Obj NIL = MAKE_SEXPR(0);
```

and its companion

```
int is_nil(Obj p)      { return EQ(p, NIL); }
```

3.2 Symbols

A symbol is really an index into a hashtable, either an integer index into a closed hashtable, or an address to a bucket in a dynamic hashtable. Let's use $\{S, M_{49}, M_{48}\} = \{0, 0, 0\}$:



```
#define SYMBOL_MASK    0x7ffc000000000000 /* 0 11111111111 1100 address */
#define IS_SYMBOL(v)   ((v.as_int & NAN_MASK) == SYMBOL_MASK)
#define MAKE_SYMBOL(p) { .as_int = (uint64_t) (p) | SYMBOL_MASK };
```

These macros can be used with the symbol, like this:

```
Obj mksym(char *id) { Obj sym = MAKE_SYMBOL(lookup(id)); return sym; }
```

3.3 Primitives

A primitive needs to be recognised when a procedure is applied to some arguments: if the procedure is a primitive, like `car` or `+`, the normal eval/apply recursion hits a base case and a bit of C-code is executed. For primitives we use $\{S, M_{49}, M_{48}\} = \{0, 0, 1\}$:

3.4 Booleans

In my Lisp interpreter I have not used the boolean datatype. Instead, I use two symbols `True` and `False`:

```
Obj True = mksym("#t");
Obj False = mksym("#f");
```

To check whether a value is a boolean, we simply see if it's one of `True` or `False`:

```
int booleanp() {
    return (EQ(car(argl), True) || EQ(car(argl), False)) ? True : False;
}
```

3.5 Integers

Another decision in my LISP interpreter was to omit integers and instead use double floating point for all numbers. Not only does this simplify the code by not having to coerce between integers and doubles, but we also avoid the problem of having to store signed integers into the 48-bit part of the NaN-value.

Double values are printed as integers if the fractional part is zero:

```
void display(Value expr) {
  // ...
  } else if (IS_DOUBLE(expr)) {
    double x = DOUBLEVALUE(expr);
    if (x == trunc(x)) {
      if (fabs(x) < INT_MAX)
        printf("%.0f", x);      /* small int: show as e.g. 62 */
      else
        printf("%e", x);        /* large int: show as e.g. 3.14e12 */
    } else
      printf("%f", x);          /* show as e.g. 3.14 */
  }
```

If you wish to include integers, you need to consider negative numbers. 32-bit signed integers can be accomodated by adding a third member `as_signed_int` to the union, and using the following macros:

```
#define INT_MASK       0x7ffe000000000000  /* 0 11111111111 1110 signed int */
#define IS_INT(v)      ((v.as_int & NAN_MASK) == INT_MASK)
#define MAKE_INT(p)    { .as_int = (0xffffffffffff & ((int32_t) (p))) | INT_MASK };
#define GET_INT(v)     ( v.as_signed_int)
```

3.6 And the rest...

By now you get the picture. There are 51 bits we can use in the NaN: 48 will be used as an address or an integer value and the other three bits are enough to store a type tag for the eight different data types (of which I only used six in my Lisp interpreter).

Which bit pattern to choose for which data type doesn't matter because testing and masking are the same for each type, although the bit patterns are different. A future project would be to write a more general set of macros for creating, testing for and extracting, where each datatype is defined by the three bits $\{S, M_{49}, M_{48}\}$.