Common Lisp Object Representation Strategies: The Umass Concurrent & Common Lisp Implementation

Kelly E. Murray
Daniel D. Corkill

COINS Technical Report No. 88-35
May 4, 1988

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

An appropriate Lisp object representation is an important aspect of an efficient Common Lisp implementation. To be most effective, the representation must be tailored to the characteristics of the computer hardware and the characteristics of Common Lisp. In this paper we review object representation strategies and present the object representation for the Umass Concurrent & Common Lisp (C&CL) system currently being implemented at the University of Massachusetts on the Sequent Symmetry multiprocessor.

This research was sponsored in part by the National Science Foundation under CER Grant DCR-8500332.

1 Data Types in Common Lisp

In contrast to a strongly-typed language such as Pascal, a Common Lisp system uses *latent* data typing. Latent data typing allows a function to delay until *run-time* the determination of the type of an argument, so it can take different actions depending on the type of object supplied. Many Common Lisp functions are defined to accept many different types of objects (such functions are termed *polymorphic*). For example, the + function can add values of any numeric type.

Functions that manipulate pointers to Lisp objects, rather than the objects themselves, do not need to know the objects' data type. For example, when fetching the element of a general vector, the object returned can be of any data type. Conversely, any object (regardless of data type) can be stored in a general vector. Only when the value of the Lisp object is operated on is its data type required. This characteristic allows a Lisp program to manipulate "unknown-type" objects and is the basis for use of *futures* [Halstead, 1985] for concurrent Lisp.

Typically, most functions perform some operation on their arguments and therefore require the arguments to be a member of a set of data types. Data type checking can consume a significant portion of computational resources. While hard data is scarce, Shaw reports figures of 11% - 34% of instructions on conventional hardware are spent performing type checks [Shaw, 1988]. This is one reason why Lisp machines,[1] which perform these tests as part of their normal instruction cycle, have a performance advantage over "stock" hardware. This is also why compiler declarations for eliminating run-time type checking can result in significant increases in performance.

[1]In this paper, "Lisp machines" refers to specialized hardware for executing Lisp code. The Symbolics 3600 series and Texas Instruments Explorer workstations are examples of this class of hardware.

2 Data Type Checking and Lisp Object Representation

The techniques used to quickly determine the data type of a Lisp object are a critical aspect of an implementation's performance. We consider four different techniques:

Object Direct: In the object direct scheme, the data type is encoded with the object. The chief advantage of object direct encoding is that it allows objects of varying sizes to be allocated from one contiguous heap, simplifying heap management routines. A disadvantage of object direct encoding is the extra space required in each object to hold the data type information. When the size of objects is small, this can significantly increase the memory needed. For example, cons objects consist of only two pointers (typically, two machine words on stock hardware). Encoding the cons data type information might require the addition of another machine word to each cons object, a 50% increase in memory requirements. Another important disadvantage is that a memory reference is required to determine the data type of the object when only a pointer to the object is available.

Pointer Indirect: The pointer indirect scheme reduces the storage overhead associated with the object direct approach by dividing the heap into regions.[2] Each region contains a single Lisp data type.[3] To determine the type of an object, the pointer to the object is analyzed to determine the region containing the object, and the data type information associated with that region is retrieved. The pointer indirect approach has characteristics similar to the object direct scheme, but only requires data type information for each region instead of each object. However, because it subdivides the heap, storage allocated to a particular type can go unused while running out of storage for other types.

Pointer Direct: This scheme encodes the type in the pointer directly. It eliminates the need to access memory and hence can be faster. The

[2]This strategy has also been called BIBOP for BIg Bag Of Pages.

[3]Sometimes multiple regions are available for the same Lisp data type.

disadvantages depend on how the type is encoded in the pointer. We discuss some possible encodings of Pointer Direct in the sections that follow.

Tagged: The type is with the pointer, but *separate* from the address. This requires a word length longer than the address field and is generally only available with specialized hardware.

Combinations of the schemes are possible. For example, some types can be represented directly in the pointer, with additional type information in the object itself.

The number of distinguishable data types needed for a Common Lisp system is fairly large. It includes at least 6 kinds of numbers, symbols, conses, characters, compiled functions, hashtables, defstructs, streams, and a host of array and vector types. Encoding some system types, such as stack pointers, trap objects, etc., is necessary and adds to the count.

An important consideration in the type scheme is immediate objects. For objects such as small integers (fixnums), small floats, and characters, representing the object within the pointer itself eliminates having to allocate memory for them. Note that a pure Object Direct scheme cannot support immediate objects.

An additional consideration in a Common Lisp system is how all the numeric types are represented. The EQL function returns true if the two pointers are identical OR the objects are numbers with the same type and value. The EQL function is the default :TEST function for many Common Lisp functions. To keep the EQL test fast, it is desirable to be able to quickly determine that two objects are both numbers of the same type. By having the type information encoded in the pointer itself, this test can be made without forcing a memory reference.

Because type checking is critical to a Common Lisp system, whatever scheme or combination of schemes is employed should be designed for optimal run-time performance. For stock hardware, Pointer Direct supports immediate objects, can encode number types directly, and also gives the best performance for heap allocated objects. Therefore, we have adopted a Pointer Direct scheme for C&CL.

In the remainder of the paper, we review alternative methods of Pointer Direct, and the performance implications for a 80386-based Common Lisp.

3 Shifted, High-bits Encoding

Given a Pointer Direct scheme, the method of encoding the type in the pointer must be determined. A strategy similar to the Pointer Indirect scheme is a straightforward method. The high address bits are used to divide memory into regions, each with a pre-specified type. Thus each type has its own contiguous space of memory.

To determine the type from the pointer, the type bits must be compared with a given type code. This can be accomplished by either masking out the lower bits or shifting the pointer and doing a compare. Shifting results in a number representable with one byte, so it would reduce the immediate data needed for the compare. If the pointer is not to be destroyed, the result of the shift must be stored somewhere else, or the pointer must be copied to another location. On the 386, shifting and masking operations destroy the operand,[4] so the pointer must be copied.

For the 386, the following code would test for a CONS object:[5]

```
mov Arg0 Temp0          ; Copy the pointer.
lsr ptrsize Temp0       ; Shift it to get only type bits.
cmp Temp0 Cons-type     ; Compare it with Cons-Type.
jne  not-cons           ; Branch out if not a Cons.
...                     ; Operate on the Cons.
```

When dispatching on more than one type, only the compare and branch is necessary once the type code has been extracted. The time required depends on the ordering of the type checks and the type of the argument. When speed is more important than space, an indirect branch can be used with the type code as the offset.

[4]The 386 has a double-shift instruction that could be used to shift into another register without destroying the first operand, but to use it a zero must be stored into the result register first to insure only the type bits would be seen, which would have the same costs as the code presented. However, if the type code width is 8 bits, then a byte-compare instruction could be used, which would ignore the unknown higher bits in the result register, eliminating one instruction from the code presented.

[5]The assembly code presented is in our own syntax. In particular, the destination is the *second* operand.

This would only be appropriate when most types are expected, since it requires significant code space as well as a fixed overhead of referencing memory for the initial branch. Because most type checking is for only one type (or at most several) this would not be used often, if at all. For example, the + function only tests for the numeric types, and often the argument is a fixnum, which is tested for first.

The number of high bits used for the type determines both how many types can be distinguished as well as the size of each region. Five bits of type would yield 32 different types, which is adequate. For a 32 bit address, this would leave a 128 megabyte region size, which should be sufficient for most Common Lisp applications. If a full 32 bit address is not supported by the operating system, the entire pointer representation must be shifted down to the available size of the address, causing severe shrinkage in the address space available for a single type. A 24 bit address with 5 type bits yields only a one-half megabyte space for each type, leading to frequent garbage collections, and an inability to support large applications. A 28 bit address yields an 8 megabyte segment size, which may still be too small for some large applications.

One compromise is to decrease the type bits to 4 bits. This would double the region size to 16 megabytes, which may be tolerable. This would reduce the number of types to 16, which would force some objects to share the same pointer type, forcing some alternative method for their type representation.

The Shifted, High-bits encoding also requires the operating system to support a sparsely populated virtual address space, since segments are distributed throughout the entire address space, causing large "holes" of unallocated address space between regions.

3.1 Immediate Data Manipulation

For immediate data types, the non-type bits of the pointer bits contain the immediate object. For small integers (fixnums), the data must be shifted into the high bits to allow hardware detection of arithmetic overflow. Addition of two fixnums would look like:

```
lsl typesize Arg0       ; Put sign bits in place.
lsl typesize Arg1
add Arg1 Arg0           ; Add them.
bov needbignum          ; Make a bignum if overflow.
lsr typesize Arg0       ; Get bits back in right place.
```

If overflow is not possible, then the three shift instructions can be eliminated. Similar manipulations are required for the other immediate data types.

4 Low-bits Encoding

An alternative to using the high bits to encode the type is to use the low bits. This strategy has a number of advantages. It doesn't place any requirements on the operating system to handle a large or sparse address space. In addition, the sign bits of fixnums do not need shifting to detect overflow. Another advantage that is becoming more important in modern Common Lisp systems is that it allows objects to be placed anywhere in the address space. This includes the ability to allocate objects on the control stack, which can be automatically deallocated upon function return. The difficulty with this strategy is that there are not enough low bits available for type encoding.

In a byte-addressed machine with a 4-byte word size, a word-aligned address leaves the lower two bits zero. Thus it is possible to use these lower two bits to encode type information. However, two bits only supports four different types.

By enforcing an object size alignment of two words, a third bit becomes available, allowing 8 different types to be represented. However, adopting a two word allocation size has an impact on memory requirements. Cons objects are 2 words, so it doesn't effect these. Symbols use 5 words, and hence would be forced to use 6 words. Symbols are generally static, though, so the extra memory would be a relatively "fixed" cost.[6] Wasted space

[6]The extra word can be used for various things, making the extra memory required less of a concern.

required for padding out strings is a concern.

The problem with low-bit encoding is the similar to a high-bit encoding. The type code must either be shifted out of the pointer, or the address portion masked out before a comparison can be made on the type.

```
mov  Arg0 Temp       ; Copy the pointer into a free register.
and  #b111 Temp      ; Mask out the address.
cmpb Temp Cons-Type  ; Check the type.
bne  not-cons        ; Branch on Test
...                  ; Operate on the cons.
```

A subtle complication with using the low-bit encoding is that the pointer address may not be word aligned. Memory accesses can be slower with unaligned addresses, since additional bus cycles may be required. Alignment can be maintained by adding a displacement to the pointer when performing a memory reference.

5 Shifted-Address, Low-Bits Encoding

Another variation of low-bit encoding is to use a full byte to encode the type. The lower type byte must now be shifted out before the pointer is used to reference memory. However, we now get 256 types represented directly in the typed-pointer. A type check can test the low byte by using a compare-byte instruction.

```
cmpb Cons-Type Arg0 ; Test for Cons-type.
bne  not-cons       ; Branch on Test.
lsr  8 Arg0         ; Shift out the type byte.
...                 ; Operate on the cons.
```

One disadvantage of this scheme is that the shifting destroys the type information from the pointer. Very often the type is only checked once, so this may not be a significant disadvantage.

Another problem with this scheme is that it limits the address space available to 24 bits. However, if objects are word-aligned, it is possible to have the 24 bit address extended to 26 by having the low two bits "hidden." By adopting two word alignment, we pick up yet a third bit, giving us a 27 bit address. Using this strategy the top 3 bits of the type code become the low 3 bits of address. We can deal with any non-alignment this causes using the same alternatives as the previous strategy.

The major problem with this scheme is that the pointer must always be shifted before a memory reference, even when type checks are not being performed.

6 Pointer-direct, Bit-assignment Encoding

An alternative to using a bit pattern as a type code is to assign individual bits of the type data to particular data types. To check for a particular type, code only needs to test for the presence of a particular bit. When validating an argument is of a particular type, only two instructions are needed:

```
tst Arg0,Cons-Bit ; Test for Cons-type
beq  not-cons     ; Branch if bit wasn't on.
...               ; Operate on the cons.
```

An advantage of this scheme is that it doesn't require the use of an additional register, which is important given the relatively small number of registers on the 386.[7] A push and pop would be necessary to free up a register if none were free for a type test, adding to the cost.

This technique can be used with either low or high-bit type encoding. In a high-bit encoding, a disadvantage is that a large part of the possible address space is not usable. Since objects can only be of one type, only

[7]The 386 has only 8 general registers, used for both data and addresses. Four of those are allocated to dedicated purposes in the C&CL system (the stack-pointer, frame-pointer, binding-pointer, and the active-function), leaving only 4 for general use.

one bit can be set. Thus, using 5 type bits allows only 6 of the 32 possible combinations to be used.[8]

The more problematic difficulty with this scheme is there are only a limited number of bits available. In the shift scheme 5 type bits allowed 32 types, whereas in the bit scheme, it only allows 6. In the low bits case, 3 bits yields 8 different types, whereas a bit scheme yields only 4.

7 The Umass C&CL Type Implementation

The type representation currently in use in C&CL is a hybrid of the previously discussed strategies. We have attempted to exploit the advantages of each scheme while avoiding their disadvantages. We combine both Object Direct and Pointer Direct, using a combination of Pointer Direct encoding that is tailored to the characteristics of Common Lisp.

To reduce virtual memory requirements, and allow fast fixnum arithmetic, the low bits are used for type encoding. By adopting two word alignment of objects, the 3 low bits are available. We eliminate the need to mask out the higher bits by assigning types to particular bits. Figure 1 shows their assignments.

7.1 Symbols, Lists, and NIL

Testing for either a symbol or a cons object is a simple bit test. For a symbol, the two-word-aligned pointer is displaced by -7, which forces the 0 bit to be set. Cons pointers are similarly displaced, but use a displacement of -6, which forces the 1 bit to be set. When accessing symbols or conses, a corresponding positive displacement is added to align the pointers to word boundaries. For example, taking the CDR with full type checking becomes:

```
tstb Cons-Bit Arg0 ; Test for Cons-type
beq  not-cons      ; Branch to error if not a cons.
mov  (Arg0 6) Arg0 ; Replace Arg0 with its CDR (slot 0).
```

[8]The system could "cheat" and use the other regions for objects that have well-defined characteristics (e.g., file buffers, assembly routines), but in general, they wouldn't be available for user-visible objects.

```
     | Low Bits  |
-----------------
...  | 2 | 1 | 0 |
-----------------
               ^
           ^   0 Symbol
       ^   1 Cons
       2 Object Direct Encoding
```

Figure 1: Low bit assignments.

Because NIL is both a symbol and a list, we should set both the 0 and 1 bits. However, this causes some complication since we add 6 for a cons access and 7 for a symbol access, neither of which is appropriate for NIL. When accessing the CDR slot of NIL, 6 is added to the pointer, which yields a lower three bits of 001. Similarly, 10 is added to the pointer for a CAR access. Thus, the CDR and CAR slots of NIL are unaligned by one byte, forcing a shift of the slot contents (NIL itself) forward one byte.

This works fine for Cons access, but what about Symbol accesses? Performing a Symbol-Value access on NIL must return NIL as well as the CAR and CDR. When accessing the first slot in a symbol, 7 is added to the pointer, yielding a lower three bits of 010, which is *two* bytes forward. Therefore, the shifted contents that work properly for Cons access are not correct for a Symbol access. One solution to this problem is to always check for NIL before a symbol access, but clearly this is undesirable. Fortunately, this problem only exists for the *first two* symbol slots, since symbol slots greater than two don't overlap with the CONS slots. Therefore, we layout a symbol with the two most infrequently accessed slots (Package and Pname) as the first two slots. Before accessing a Pname or Package slot, a check for NIL must be made, but accesses to the Plist, Value, and Function slots do not require the check, since we are free to store two byte shifted contents in these slots for NIL. Figure 2 shows the details.

```
NIL == #x0002101B

Address   Contents
-------   --------
21021     1B <== CDR [0002101B]
21022     10   <-- Would be Symbol Slot 0 (Pname)
21023     02
21024     00
21025     1B <== CAR [0002101B]
21026     10   <-- Would be Symbol Slot 1 (Package)
21027     02
21028     00
21029     00 (not accessed)
2102A     1B <-- Symbol Plist (User Writeable)
2102B     10
2102C     02
2102D     00
2102E     1B <-- Symbol Value [0002101B]
2102F     10
21030     02
21031     00
21032     44 <-- Symbol Function [00021044] (Undefined)
21033     10
21034     02
21035     00
```

Figure 2: Memory layout for NIL.

```
    | F   I          N | All Zeros |
------------------------------------
.. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
------------------------------------
I = Immediate (except Fixnums)
N = Allocated Number
F = Future

  0 :  0 0 0 0 0 0 0 0  Fixnum
  8 :  0 0 0 0 1 0 0 0  Bignum
 24 :  0 0 0 1 1 0 0 0  Ratio
 40 :  0 0 1 0 1 0 0 0  Single Float
 56 :  0 0 1 1 1 0 0 0  Complex
 64 :  0 1 0 0 0 0 0 0  Character
 80 :  0 1 0 1 0 0 0 0  Short Float
 96 :  0 1 1 0 0 0 0 0  System Immediate
128 :  1 0 0 0 0 0 0 0  Future
```

Figure 3: Low Byte Encodings.

In addition, notice that only a NIL pointer has the low two bits set. Thus, checking for NIL requires only comparing the low byte of a pointer, reducing a NIL test by 3 bytes.

We delay discussing the Object Direct bit until later.

7.2 Low Byte Encoding

If all 3 lower bits are zero, then the full lower byte contains the type of the object. Figure 3 shows their assignments.

For these objects, a simple compare byte is a sufficient check. Note that a fixnum does not have the immediate bit set. Fixnums are discussed in a later section.

7.3 Allocated Number Types

All the non-immediate numeric types have their type encoded in the low byte of the pointer, and furthermore, bit 3 is set for them all. Thus, an EQL test can quickly determine two objects are not EQL without requiring a memory reference. An EQL test becomes:

```
cmp  Arg1 Arg0          ; EQ?
beq  equal              ; Yep.
cmpb Arg1 Arg0          ; Same Byte Type?
bne  not-equal
tstb Number-type-bit Arg0 ; Maybe Numbers?
bne  not-equal
tstb Low-byte-Mask Arg0   ; Really Low Byte type?
...                     ; Full number comparison.
```

A disadvantage with this strategy is that the pointers to these non-immediate numbers must be shifted before accessing their values. Furthermore, the address space available to them is reduced to only 27 bits (shifting by 5, leaving the high three type bits as the low address bits). However, these type of numbers were not designed for "high performance," nor are they used heavily by most applications. Furthermore, time spent manipulating their values overshadows the extra instruction needed to shift the pointer.

7.4 Futures

For multiprocessing applications, C&CL supports futures as first class objects. This has an impact on the EQ test that forces it to behave similarly to the EQL test for numeric types. Two objects are EQ if either they are identical pointers OR if either or both of them is a future object which has a determined value that is EQ to the other object. Thus, all EQ tests must check for futures before failing. Because the Future type is encoded directly in the pointer, a future test does not require a memory reference. For a multiprocessing application, the EQ test becomes.[9]

```
       cmp  Arg1 Arg0          ; Pointer-EQ?
       beq  equal              ; Yep.
       cmpb Arg0 future-type   ; Arg0 a future?
       beq  get-arg0-future    ; Get the determined future value.
 arg0-determined               ; Return here with arg0 value.
       cmpb Arg1 future-type   ; Arg1 a future?
       beq  get-arg1-future    ; Get the determined future value.
 arg1-determined               ; Return here with arg1 value.
       cmp  Arg1 Arg0          ; Try again.  Maybe still not EQ.
       beq  equal
```

Having the future type in the low byte has the same disadvantages as the non-immediate numeric types; the pointer must be shifted before accessing the future object. However, a large majority of the time the objects will not be futures, making this an appropriate tradeoff.

7.5 Fixnums

Fixnums have their lower byte all zeros. This allows two fixnums to be added or subtracted directly without any shifting, masking, or correction. Thus adding two fixnums looks like:

```
       add  Arg1 Arg0          ; Add them.
       bov  needbignum         ; Make a bignum if overflow.
```

If conversion to machine integers is required, only a simple shift is necessary. This gives us a signed 23 bit value for fixnums, providing a range of about plus or minus 8 million, which is reasonable.

[9]This is a large amount of code for what is considered the most simple of all operations in Lisp. It demonstrates the important ability of the compiler to determine when future objects are not possible.

7.6 Other Immediates

For Characters, the second byte contains the code. Their manipulation is fairly straightforward. Short-Floats have only 24 bits of value, making the precision fairly low. We are currently using a 1 bit sign, 6 bits of exponent, and an 17 bit mantissa, yielding around 4 decimal digits of precision.

7.7 Object Direct Encoding

If bit 2 is set, then the object begins with a header word. The low byte in the header word contains the type. These include all the array types, as well as structures and compiled functions. Testing for these types will require a memory reference to access the header. However, when accessing or updating an array element, the index is generally validated before the operation. Thus, the access to the header required for the index validation is already performed by the type check.

The format of an array header is similar to a fixnum, except the lower byte contains the array type, instead of zero, as with a fixnum. Thus a general-vector array access with full type checking would look like the following (Arg0 contains the index, and Arg1 contains the array):

```
cmpb Arg0 0             ; Index a fixnum?
bne  illegal-index
tstb #b100 Arg1         ; Is it an Object-Direct pointer?
beq  not-vector
mov  (Arg1 4) Temp      ; Get the header in Temp.
cmpb Temp Gvectortype ; Is it a general-vector?
bne  not-vector
;; Clear the type byte for the index compare.
movb 0 temp
cmp  arg0 temp          ; A legal index?
bae  illegal-index      ; UNsigned test, so negatives fail.
lsr  6 arg0             ; Shift index into a WORD offset.
mov  (Arg1 Arg0 header-offset) Arg0 ; Replace with element.
```

If no checks are performed at all, only the last two instructions would be needed.

8 Summary

Type checking is an important component of all Lisp systems. We have reviewed some alternatives designs, and from them generated a hybrid strategy that yields high performance for a 80386-based Common Lisp system.

9 References

Halstead, 1985 Robert H. Halstead, Jr, "Multilisp: A Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems*, Vol 7, No. 4. 1985.

Shaw, 1988 Robert A. Shaw, *Empirical Analysis of a Lisp System*, Stanford Technical Report CSL-TR-88-351, February, 1988.