Design of CMU Common Lisp

Robert A. MacLachlan (ed)

July 22, 1991
CMU-CS-91-???

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report documents internal details of the CMU Common Lisp compiler and run-time system. CMU Common Lisp is a public domain implementation of Common Lisp that runs on various Unix workstations.

Contents

Part I

Compiler Organization

Part II

Compiler Retargeting

Part III

Run-Time system

Chapter 1

Object Format

1.1 Tagging

The following is a key of the three bit low-tagging scheme:

000 even fixnum

001 function pointer

010 even other-immediate (header-words, characters, symbol-value trap value, etc.)

011 list pointer

100 odd fixnum

101 structure pointer

110 odd other immediate

111 other-pointer to data-blocks (other than conses, structures, and functions)

This tagging scheme forces a dual-word alignment of data-blocks on the heap, but this can be pretty negligible:

- RATIOS and COMPLEX must have a header-word anyway since they are not a major type. This wastes one word for these infrequent data-blocks since they require two words for the data.
- BIGNUMS must have a header-word and probably contain only one other word anyway, so we probably don't waste any words here. Most bignums just barely overflow fixnums, that is by a bit or two.
- Single and double FLOATS? no waste, or one word wasted

- SYMBOLS have a pad slot (current called the setf function, but unused.)

Everything else is vector-like including code, so these probably take up so many words that one extra one doesn't matter.

1.2 GC Comments

Data-Blocks comprise only descriptors, or they contain immediate data and raw bits interpreted by the system. GC must skip the latter when scanning the heap, so it does not look at a word of raw bits and interpret it as a pointer descriptor. These data-blocks require headers for GC as well as for operations that need to know how to interpret the raw bits. When GC is scanning, and it sees a header-word, then it can determine how to skip that data-block if necessary. Header-Words are tagged as other-immediates. See the sections "Other-Immediates" and "Data-Blocks and Header-Words" for comments on distinguishing header-words from other-immediate data. This distinction is necessary since we scan through data-blocks containing only descriptors just as we scan through the heap looking for header-words introducing data-blocks.

Data-Blocks containing only descriptors do not require header-words for GC since the entire data-block can be scanned by GC a word at a time, taking whatever action is necessary or appropriate for the data in that slot. For example, a cons is referenced by a descriptor with a specific tag, and the system always knows the size of this data-block. When GC encounters a pointer to a cons, it can transport it into the new space, and when scanning, it can simply scan the two words manifesting the cons interpreting each word as a descriptor. Actually there is no cons tag, but a list tag, so we make sure the cons is not nil when appropriate. A header may still be desired if the pointer to the data-block does not contain enough information to adequately maintain the data-block. An example of this is a simple-vector containing only descriptor slots, and we attach a header-word because the descriptor pointing to the vector lacks necessary information – the type of the vector's elements, its length, etc.

There is no need for a major tag for GC forwarding pointers. Since the tag bits are in the low end of the word, a range check on the start and end of old space tells you if you need to move the thing. This is all GC overhead.

1.3 Structures

A structure descriptor has the structure lowtag type code, making `structurep` a fast operation. A structure data-block has the following format:

```
-------------------------------------------------------
|   length (24 bits) | Structure header type (8 bits) |
-------------------------------------------------------
```

```
|   structure type name (a symbol)                      |
-------------------------------------------------------
|   structure slot 0   |
-------------------------------------------------------
|   ... structure slot length - 2   |
-------------------------------------------------------
```

The header word contains the structure length, which is the number of words (other than the header word.) The length is always at least one, since the first word of the structure data is the structure type name.

1.4 Fixnums

A fixnum has one of the following formats in 32 bits:

```
-------------------------------------------------------
|        30 bit 2's complement even integer   | 0 0 0 |
-------------------------------------------------------
```

or

```
-------------------------------------------------------
|        30 bit 2's complement odd integer    | 1 0 0 |
-------------------------------------------------------
```

Effectively, there is one tag for immediate integers, two zeros. This buys one more bit for fixnums, and now when these numbers index into simple-vectors or offset into memory, they point to word boundaries on 32-bit, byte-addressable machines. That is, no shifting need occur to use the number directly as an offset.

This format has another advantage on byte-addressable machines when fixnums are offsets into vector-like data-blocks, including structures. Even though we previously mentioned data-blocks are dual-word aligned, most indexing and slot accessing is word aligned, and so are fixnums with effectively two tag bits.

Two tags also allow better usage of special instructions on some machines that can deal with two low-tag bits but not three.

Since the two bits are zeros, we avoid having to mask them off before using the words for arithmetic, but division and multiplication require special shifting.

1.5 Other-immediates

As for fixnums, there are two different three-bit lowtag codes for other-immediate, allowing 64 other-immediate types:

```
-----------------------------------------------------------------
|   Data (24 bits)        | Type (8 bits with low-tag)   | 1 0 |
-----------------------------------------------------------------
```

The type-code for an other-immediate type is considered to include the two lowtag bits. This supports the concept of a single "type code" namespace for all descriptors, since the normal lowtag codes are disjoint from the other-immediate codes.

For other-pointer objects, the full eight bits of the header type code are used as the type code for that kind of object. This is why we use two lowtag codes for other-immediate types: each other-pointer object needs a distinct other-immediate type to mark its header.

The system uses the other-immediate format for characters, the `symbol-value` unbound trap value, and header-words for data-blocks on the heap. The type codes are laid out to facilitate range checks for common subtypes; for example, all numbers will have contiguous type codes which are distinct from the contiguous array type codes. See section ?? for details.

1.6 Data-Blocks and Header-Word Format

Pointers to data-blocks have the following format:

```
-----------------------------------------------------------------
|      Dual-word address of data-block (29 bits)       | 1 1 1 |
-----------------------------------------------------------------
```

The word pointed to by the above descriptor is a header-word, and it has the same format as an other-immediate:

```
-----------------------------------------------------------------
|   Data (24 bits)        | Type (8 bits with low-tag) | 0 1 0 |
-----------------------------------------------------------------
```

This is convenient for scanning the heap when GC'ing, but it does mean that whenever GC encounters an other-immediate word, it has to do a range check on the low byte to see if it is a header-word or just a character (for example). This is easily acceptable performance hit for scanning.

The system interprets the data portion of the header-word for non-vector data-blocks as the word length excluding the header-word. For example, the data field of the header for ratio and complex numbers is two, one word each for the numerator and denominator or for the real and imaginary parts.

For vectors and data-blocks representing Lisp objects stored like vectors, the system ignores the data portion of the header-word:

```
-----------------------------------------------------------------
```

```
| Unused Data (24 bits)   | Type (8 bits with low-tag) | 0 1 0 |
-----------------------------------------------------------------
|           Element Length of Vector (30 bits)          | 0 0 |
-----------------------------------------------------------------
```

Using a separate word allows for much larger vectors, and it allows `length` to simply access a single word without masking or shifting. Similarly, the header for complex arrays and vectors has a second word, following the header-word, the system uses for the fill pointer, so computing the length of any array is the same code sequence.

1.7 Data-Blocks and Other-immediates Typing

These are the other-immediate types. We specify them including all low eight bits, including the other-immediate tag, so we can think of the type bits as one type – not an other-immediate major type and a subtype. Also, fetching a byte and comparing it against a constant is more efficient than wasting even a small amount of time shifting out the other-immediate tag to compare against a five bit constant.

```
Number   (< 30)
  bignum 10
    ratio 14
    single-float 18
    double-float 22
    complex 26

Array   (>= 30 code 86)
   Simple-Array   (>= 20 code 70)
 simple-array 30
      Vector  (>= 34 code 82)
 simple-string 34
 simple-bit-vector 38
 simple-vector 42
 (simple-array (unsigned-byte 2) (*)) 46
 (simple-array (unsigned-byte 4) (*)) 50
 (simple-array (unsigned-byte 8) (*)) 54
 (simple-array (unsigned-byte 16) (*)) 58
 (simple-array (unsigned-byte 32) (*)) 62
 (simple-array single-float (*)) 66
 (simple-array double-float (*)) 70
      complex-string 74
      complex-bit-vector 78
      (array * (*))   -- general complex vector. 82
```

```
   complex-array 86

code-header-type 90
function-header-type 94
closure-header-type 98
funcallable-instance-header-type 102
unused-function-header-1-type 106
unused-function-header-2-type 110
unused-function-header-3-type 114
closure-function-header-type 118
return-pc-header-type (a.k.a LRA) 122
value-cell-header-type 126
symbol-header-type 130
base-character-type 134
system-area-pointer-type (header type) 138
unbound-marker 142
weak-pointer-type 146
structure-header-type 150
```

1.8 Strings

All strings in the system are C-null terminated. This saves copying the bytes when calling out to C. The only time this wastes memory is when the string contains a multiple of eight characters, and then the system allocates two more words (since Lisp objects are dual-word aligned) to hold the C-null byte. Since the system will make heavy use of C routines for systems calls and libraries that save reimplementation of higher level operating system functionality (such as pathname resolution or current directory computation), saving on copying strings for C should make C call out more efficient.

The length word in a string header, see section "Data-Blocks and Header-Word Format", counts only the characters truly in the Common Lisp string. Allocation and GC will have to know to handle the extra C-null byte, and GC already has to deal with rounding up various objects to dual-word alignment.

1.9 Symbols and NIL

Symbol data-block has the following format:

```
-------------------------------------------------------
|     5 (data-block words)     | Symbol Type (8 bits) |
-------------------------------------------------------
|    Value Descriptor               |
-------------------------------------------------------
```

```
| Function Pointer        |
-------------------------------------------------------
|       Raw Function Address        |
-------------------------------------------------------
|  Setf Function         |
-------------------------------------------------------
|  Property List       |
-------------------------------------------------------
|    Print Name       |
-------------------------------------------------------
|     Package       |
-------------------------------------------------------
```

Most of these slots are self-explanatory given what symbols must do in Common Lisp, but a couple require comments. We added the Raw Function Address slot to speed up named call which is the most common calling convention. This is a non-descriptor slot, but since objects are dual word aligned, the value inherently has fixnum low-tag bits. The GC method for symbols must know to update this slot. The Setf Function slot is currently unused, but we had an extra slot due to adding Raw Function Address since objects must be dual-word aligned.

The issues with nil are that we want it to act like a symbol, and we need list operations such as CAR and CDR to be fast on it. CMU Common Lisp solves this by putting nil as the first object in static space, where other global values reside, so it has a known address in the system:

```
-------------------------------------------------------  <-- space
| 0       |      start
-------------------------------------------------------
|     5 (data-block words)     | Symbol Type (8 bits) |
-------------------------------------------------------  <-- nil
|     Value/CAR       |
-------------------------------------------------------
|   Definition/CDR       |
-------------------------------------------------------
|        Raw Function Address        |
-------------------------------------------------------
|   Setf Function         |
-------------------------------------------------------
|   Property List       |
-------------------------------------------------------
|     Print Name       |
-------------------------------------------------------
|      Package       |
-------------------------------------------------------
```

```
|           ...          |
-------------------------------------------------------
```

In addition, we make the list typed pointer to nil actually point past the header word of the nil symbol data-block. This has usefulness explained below. The value and definition of nil are nil. Therefore, any reference to nil used as a list has quick list type checking, and CAR and CDR can go right through the first and second words as if nil were a cons object.

When there is a reference to nil used as a symbol, the system adds offsets to the address the same as it does for any symbol. This works due to a combination of nil pointing past the symbol header-word and the chosen list and other-pointer type tags. The list type tag is four less than the other-pointer type tag, but nil points four additional bytes into its symbol data-block.

;;;; Array Headers.

The array-header data-block has the following format:

```
----------------------------------------------------------------
| Header Len (24 bits) = Array Rank +5   | Array Type (8 bits) |
----------------------------------------------------------------
|               Fill Pointer (30 bits)                   | 0 0 |
----------------------------------------------------------------
|               Available Elements (30 bits)             | 0 0 |
----------------------------------------------------------------
|               Data Vector (29 bits)                  | 1 1 1 |
----------------------------------------------------------------
|               Displacement (30 bits)                   | 0 0 |
----------------------------------------------------------------
|               Displacedp (29 bits) -- t or nil       | 1 1 1 |
----------------------------------------------------------------
|               Range of First Index (30 bits)           | 0 0 |
----------------------------------------------------------------
      .
      .
      .
```

The array type in the header-word is one of the eight-bit patterns from section "Data-Blocks and Other-immediates Typing", indicating that this is a complex string, complex vector, complex bit-vector, or a multi-dimensional array. The data portion of the other-immediate word is the length of the array header data-block. Due to its format, its length is always five greater than the array's number of dimensions. The following words have the following interpretations and types:

Fill Pointer: This is a fixnum indicating the number of elements in the data vector actually in use. This is the logical length of the array, and it is typically the same value as the next slot. This is the second word, so LENGTH of any array, with or without an array header, is just four bytes off the pointer to it.

Available Elements: This is a fixnum indicating the number of elements for which there is space in the data vector. This is greater than or equal to the logical length of the array when it is a vector having a fill pointer.

Data Vector: This is a pointer descriptor referencing the actual data of the array. This a data-block whose first word is a header-word with an array type as described in sections "Data-Blocks and Header-Word Format" and "Data-Blocks and Other-immediates Typing"

Displacement: This is a fixnum added to the computed row-major index for any array. This is typically zero.

Displacedp: This is either t or nil. This is separate from the displacement slot, so most array accesses can simply add in the displacement slot. The rare need to know if an array is displaced costs one extra word in array headers which probably aren't very frequent anyway.

Range of First Index: This is a fixnum indicating the number of elements in the first dimension of the array. Legal index values are zero to one less than this number inclusively. IF the array is zero-dimensional, this slot is non-existent.

... (remaining slots): There is an additional slot in the header for each dimension of the array. These are the same as the Range of First Index slot.

1.10 Bignums

Bignum data-blocks have the following format:

```
-------------------------------------------------------
|      Length (24 bits)        | Bignum Type (8 bits) |
-------------------------------------------------------
|          least significant bits       |
-------------------------------------------------------
    .
    .
    .
```

The elements contain the two's complement representation of the integer with the least significant bits in the first element or closer to the header. The sign information is in the high end of the last element.

1.11 Code Data-Blocks

A code data-block is the run-time representation of a "component". A component is a connected portion of a program's flow graph that is compiled as a single unit, and it contains code for many functions. Some of these functions are callable from outside of the component, and these are termed "entry points".

Each entry point has an associated user-visible function data-block (of type `function`). The full call convention provides for calling an entry point specified by a function object.

Although all of the function data-blocks for a component's entry points appear to the user as distinct objects, the system keeps all of the code in a single code data-block. The user-visible function object is actually a pointer into the middle of a code data-block. This allows any control transfer within a component to be done using a relative branch.

Besides a function object, there are other kinds of references into the middle of a code data-block. Control transfer into a function also occurs at the return-PC for a call. The system represents a return-PC somewhat similarly to a function, so GC can also recognize a return-PC as a reference to a code data-block. This representation is known as a Lisp Return Address (LRA).

It is incorrect to think of a code data-block as a concatenation of "function data-blocks". Code for a function is not emitted in any particular order with respect to that function's function-header (if any). The code following a function-header may only be a branch to some other location where the function's "real" definition is.

The following are the three kinds of pointers to code data-blocks:

Code pointer (labeled A below): A code pointer is a descriptor, with other-pointer low-tag bits, pointing to the beginning of the code data-block. The code pointer for the currently running function is always kept in a register (CODE). In addition to allowing loading of non-immediate constants, this also serves to represent the currently running function to the debugger.

LRA (labeled B below): The LRA is a descriptor, with other-pointer low-tag bits, pointing to a location for a function call. Note that this location contains no descriptors other than the one word of immediate data, so GC can treat LRA locations the same as instructions.

Function (labeled C below): A function is a descriptor, with function low-tag bits, that is user callable. When a function header is referenced from a closure or from the function header's self-pointer, the pointer has other-pointer low-tag bits, instead of function low-tag bits. This ensures that the internal function data-block associated with a closure appears to be uncallable (although users should never see such an object anyway).

Information about functions that is only useful for entry points is kept in some descriptors following the function's self-pointer descriptor. All of these together with the function's header-word are known as the "function header". GC must be able to locate the function header. We provide for this by chaining together the function headers in a NIL terminated list kept in a known slot in the code data-block.

A code data-block has the following format:

```
A -->
****************************************************************
|  Header-Word count (24 bits)   |   Code-Type (8 bits)       |
----------------------------------------------------------------
|  Number of code words (fixnum tag)        |
----------------------------------------------------------------
|  Pointer to first function header (other-pointer tag)        |
----------------------------------------------------------------
|  Debug information (structure tag)        |
----------------------------------------------------------------
|  First constant (a descriptor)        |
----------------------------------------------------------------
|  ...                               |
----------------------------------------------------------------
|  Last constant (and last word of code header)                |
----------------------------------------------------------------
|  Some instructions (non-descriptor)        |
----------------------------------------------------------------
|     (pad to dual-word boundary if necessary)        |

B -->
****************************************************************
|  Word offset from code header (24)   |   Return-PC-Type (8)  |
----------------------------------------------------------------
|  First instruction after return        |
----------------------------------------------------------------
|  ... more code and LRA header-words        |
----------------------------------------------------------------
|     (pad to dual-word boundary if necessary)           |

C -->
****************************************************************
|  Offset from code header (24)  |   Function-Header-Type (8)  |
----------------------------------------------------------------
|  Self-pointer back to previous word (with other-pointer tag) |
----------------------------------------------------------------
|  Pointer to next function (other-pointer low-tag) or NIL     |
----------------------------------------------------------------
```

```
|  Function name (a string or a symbol)        |
-------------------------------------------------------------
|  Function debug arglist (a string)       |
-------------------------------------------------------------
|  Function type (a list-style function type specifier)       |
-------------------------------------------------------------
|  Start of instructions for function (non-descriptor)       |
-------------------------------------------------------------
|  More function headers and instructions and return PCs,     |
|  until we reach the total size of header-words + code       |
|  words.       |
-------------------------------------------------------------
```

The following are detailed slot descriptions:

Code data-block header-word: The immediate data in the code data-block's header-word is the number of leading descriptors in the code data-block, the fixed overhead words plus the number of constants. The first non-descriptor word, some code, appears at this word offset from the header.

Number of code words: The total number of non-header-words in the code data-block. The total word size of the code data-block is the sum of this slot and the immediate header-word data of the previous slot. header-word.

Pointer to first function header: A NIL-terminated list of the function headers for all entry points to this component.

Debug information: The DEBUG-INFO structure describing this component. All information that the debugger wants to get from a running function is kept in this structure. Since there are many functions, the current PC is used to locate the appropriate debug information. The system keeps the debug information separate from the function data-block, since the currently running function may not be an entry point. There is no way to recover the function object for the currently running function, since this data-block may not exist.

First constant ... last constant: These are the constants referenced by the component, if there are any.

LRA header word: The immediate header-word data is the word offset from the enclosing code data-block's header-word to this word. This allows GC and the debugger to easily recover the code data-block from a LRA. The code at the return point restores the current code pointer using a subtract immediate of the offset, which is known at compile time.

Function entry point header-word: The immediate header-word data is the word offset from the enclosing code data-block's header-word to this word. This is the same as for the retrun-PC header-word.

Self-pointer back to header-word: In a non-closure function, this self-pointer to the previous header-word allows the call sequence to always indirect through the second word in a user callable function. See section "Closure Format". With a closure, indirecting through the second word gets you a function header-word. The system ignores this slot in the function header for a closure, since it has already indirected once, and this slot could be some random thing that causes an error if you jump to it. This pointer has an other-pointer tag instead of a function pointer tag, indicating it is not a user callable Lisp object.

Pointer to next function: This is the next link in the thread of entry point functions found in this component. This value is NIL when the current header is the last entry point in the component.

Function name: This function's name (for printing). If the user defined this function with DEFUN, then this is the defined symbol, otherwise it is a descriptive string.

Function debug arglist: A printed string representing the function's argument list, for human readability. If it is a macroexpansion function, then this is the original DEFMACRO arglist, not the actual expander function arglist.

Function type: A list-style function type specifier representing the argument signature and return types for this function. For example,

```
(function (fixnum fixnum fixnum) fixnum)
```

or

```
(function (string &key (:start unsigned-byte)) string)
```

This information is intended for machine readablilty, such as by the compiler.

1.12 Closure Format

A closure data-block has the following format:

```
-----------------------------------------------------------------
|  Word size (24 bits)          |  Closure-Type (8 bits)      |
-----------------------------------------------------------------
|  Pointer to function header (other-pointer low-tag)        |
-----------------------------------------------------------------
|    .         |
|                 Environment information              |
|    .         |
-----------------------------------------------------------------
```

A closure descriptor has function low-tag bits. This means that a descriptor with function low-tag bits may point to either a function header or to a closure. The idea is that any callable Lisp object has function low-tag bits. Insofar as call is concerned, we make the format of closures and non-closure functions compatible. This is the reason for the self-pointer in a function header. Whenever you have a callable object, you just jump through the second word, offset some bytes, and go.

1.13 Function call

Due to alignment requirements and low-tag codes, it is not possible to use a hardware call instruction to compute the LRA. Instead the LRA for a call is computed by doing an add-immediate to the start of the code data-block.

An advantage of using a single data-block to represent both the descriptor and non-descriptor parts of a function is that both can be represented by a single pointer. This reduces the number of memory accesses that have to be done in a full call. For example, since the constant pool is implicit in a LRA, a call need only save the LRA, rather than saving both the return PC and the constant pool.

1.14 Memory Layout

CMU Common Lisp has four spaces, read-only, static, dynamic-0, and dynamic-1. Read-only contains objects that the system never modifies, moves, or reclaims. Static space contains some global objects necessary for the system's runtime or performance (since they are located at a known offset at a know address), and the system never moves or reclaims these. However, GC does need to scan static space for references to moved objects. Dynamic-0 and dynamic-1 are the two heap areas for stop-and-copy GC algorithms.

What global objects are at the head of static space???

```
NIL
eval::*top-of-stack*
```

```
lisp::*current-catch-block*
lisp::*current-unwind-protect*
FLAGS (RT only)
BSP (RT only)
HEAP (RT only)
```

In addition to the above spaces, the system has a control stack, binding stack, and a number stack. The binding stack contains pairs of descriptors, a symbol and its previous value. The number stack is the same as the C stack, and the system uses it for non-Lisp objects such as raw system pointers, saving non-Lisp registers, parts of bignum computations, etc.

1.15 System Pointers

The system pointers reference raw allocated memory, data returned by foreign function calls, etc. The system uses these when you need a pointer to a non-Lisp block of memory, using an other-pointer. This provides the greatest flexibility by relieving contraints placed by having more direct references that require descriptor type tags.

A system area pointer data-block has the following format:

```
-------------------------------------------------------
|     1 (data-block words)        | SAP Type (8 bits) |
-------------------------------------------------------
|          system area pointer       |
-------------------------------------------------------
```

"SAP" means "system area pointer", and much of our code contains this naming scheme. We don't currently restrict system pointers to one area of memory, but if they do point onto the heap, it is up to the user to prevent being screwed by GC or whatever.