ThinLisp Manual

Version 0.5
6 October 1999

The ThinLisp Group:
Jim Allard
Ben Hyde

For additional information see `http://www.thinlisp.org/` .

Preface

ThinLisp is an open source Lisp to C translator for delivering commercial quality, Lisp-based application. It implements a subset of Common Lisp with extensions. ThinLisp itself is written in Common Lisp, and so must run on top of an underlying Common Lisp implementation such as Allegro, MCL, or CMU Lisp. The C code resulting from a translation can then be independently compiled to produce a small, efficient executable image.

Originally designed for real-time control applications, ThinLisp is not a traditional Lisp environment. It purposefully does not contain a garbage collector. ThinLisp produces compile time warnings for uses of inherently slow Lisp operations, for consing operations, and for code that is unoptimizable due to a lack of sufficient type declarations. These warnings can be suppressed by improving the code, or through use of lexical declarations acknowledging that the code is only of prototype quality. Code meeting the stringent requirements imposed by ThinLisp cannot by sped up by rewriting it in C.

Newest source distributions and this manual can be found at `http://www.thinlisp.org/` . Bugs should be reported to `bugs@thinlisp.org`

Acknowledgements

ThinLisp was begun in 1995 as a late night project born of the frustration of attempting to ship commercial quality software products written in Lisp. Four years later I'm still obsessed by the issues that drove me to it in the first uplace — practical use of high level languages to implement complex applications while achieving the performance that seems easy when using languages that are semantically `"close to the silicon"`. Hopefully this implementation will lead to some useful insights towards that goal.

Thanks to Ben Hyde for convincing Gensym and myself that this system should be taken off the shelf, polished up, and released as open source software. Mike Colena and Glen Iba wrote significant parts of the implementation. Nick Caruso, Joe Devlin, Rick Harris, and John Hodgkinson and also contributed their time, sage opinions, and code. Thanks to Lowell Hawkinson, Jim Pepe, and Dave Riddell for their support of this work at Gensym Corporation. Thanks to Kim Barrett and David Sotkowitz of IS Robotics, Inc. for contributions of multiple-inheritance code and opinions, and to Rod Brooks and Dave Barrett for their ideas. Finally, thanks to my wife, Gerry Zipser, for putting up with the bleary-eyed aftermath of coding sessions, and for helping to focus efforts during this past summer's release push.

Jim Allard
Newton, Massachusetts
October 6, 1999

1 Explanatory Rant

The long and probably uninteresting story of why TL was written.

2 Memory Architecture

Use pointers to a one word (4 byte) header structure as the basic mechanism. The type should be an 8-bit unsigned byte value. The upper 3 bytes of the header word should be used for type specific data. For vectors this should be the length, giving us a simple vector with one word of overhead (all previous Lisp implementations we've used, except the Lispm, had two or more words of header). All Lisp objects should be aligned on 4 byte addresses. There is an argument that they should be aligned on 8 byte addresses. There are two advantages to 8 byte alignment. One is that it would enable us to reliably store immediate double floats and pointers into the same array. The second is that it would can give us an immediate type tag for conses that is itself the offset to the cdr of a cons. The argument for 4 byte alignment is that all current C compilers we use align structures on 4 byte addresses, but some won't align them on 8 byte addresses, even if they contain doubles. The other argument for 4 byte alignment is that we would not need to occasionally skip forward 4 bytes when allocating from heap in order to find the next 8 byte aligned address. For now, the 4 byte alignment wins.

The exceptions to the pointer to a type tag rule are the following types, which have immediate type tags. These types are fixnums, conses, and managed-floats. If all pointers are 4 byte aligned, then the lower 2 bits of all pointers are always zeros. This gives us 3 non-zero immediate type tags. The following are the immediate type tag assignments:

```
0:  pointer to header of Lisp object
1:  immediate fixnum
2:  pointer to cons (mask with -4 for car*, follow for cdr*)
3:  immediate character
```

A determination of type can be made from a hex value of the pointer.

```
hex 0, 4, 8, C are pointers to Lisp objects
hex 1, 5, 9, and D are fixnums,
hex 2, 6, A, and E are pointers to conses
hex 3, 7, B, and F are immediate characters.
```

The arguments for a 30 bit fixnum instead of a 31 is that it makes for fast fixnum additions and subtractions without risking overflow. 31 bit fixnums require a sequence point between operations to avoid the potential for overflow.

Characters could be either immediate or remote, though there is some performance benefit to characters being immediate. We could have a preallocated array of remote character objects, and allocate a Lisp character is arefing into this array. For now we are going with immediate characters, especially since we are thinking hard about UNICODE characters, which require 16 bits and so make a pre-allocated array become somewhat too large.

The type tags in headers should not conflict with any of the immediate type tags, so that type-case can turn into a fixnum-case of the type values.

Given these set ups, type tests against straight types are at worst a null test, a mask and an integer equality test, a character byte fetch, and another integer equality test.

In C, the type for object should be an unsigned integer type 32 bits long. There are several advantages to this. All architectures (including the Alpha OSF when using a linker option) can have pointer values be represented in 32 bits, but for some platforms (the Alpha OSF) pointer types can consume something other that 4 bytes, 8 for all pointers on the Alpha OSF. By forcing all objects to consume 4 bytes, we can get interesting packing in structures and vectors on all platforms. [The following comment applies only to 8 byte alignment: For example, we could put immediate floats into odd-indexed simple vector locations, and those floats would consume 2 elements. For structures and frames, this could provide significant savings. -jra 8/30/95]

The type for managed float should have an immediate type tag so that we may have the smallest possible representation for floats, since so many are used. For each type that involves a heap allocated block of memory (i.e. all but fixnum and characters), there should be a corresponding type for a reclaimed instance of that type. This gives us a fast means of testing if a data structure is currently reclaimed or not, and would cause type tests in a safe translation. Even in production systems, we could check for double reclamation of data structures. Note that this is not possible for heap allocated data structures that use an immediate type tag.

The type tags for heap allocated data structures have no special issues, except that these values should not collide with any immediate type tags (i.e. be greater than 3), and be able to quickly determine if a data structure is reclaimed (i.e. use bit 7 as a flag).

So the type tag table is as follows:

```
Tag Value Reclaimed Tag
Dec     Hex Lisp Type C Type & Conv Dec    Hex
------------- --------- ------------- -------------
0 00 immed pointer (Header *)Obj
1 01 immed fixnum ((sint32)Obj)>>2
2 02 cons (Obj *)(Obj-2)
3 03 immed character (unsigned char)(Obj>>2)
4 04 managed-float (Mdouble *)Obj 132 84
5 05 double-float (Ldouble *)Obj 133 85
6 06 simple-vector (Sv *)Obj   134 86
7 07 string (w/fill ptr) (Str *)Obj   135 87
8 08 (simple-array ubyte 8) (Sa_uint8 *)Obj   136 88
9 09 (simple-array ubyte 16) (Sa_uint16 *)Obj  137 89
10 0A (simple-array double) (Sa_double *)Obj  138 8A
11 0B symbol (Sym *)Obj   139 8B
12 0C compiled-function (Func *)Obj   140 8C
13 0D package (Package *)Obj   141 8D
```

Each of the data structures are described below.

2.1 Simple-vectors

Simple vectors are the most often used data structure, so we'd like to keep it as small and fast as possible. It has a one word (i.e. 4 byte) header. The only components of a simple vector are the type tag, the length, and the body of the array. If the type tag is 8 bits wide (unsigned) then the length can be 24 bits of unsigned integer, giving a maximum length of 16 Meg.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:24:length;
  Obj[1]:body;
} Sv
```

In C references to array are guaranteed to not be bounds clicked. this means that this one type can be used to all elements of arbitrarily sized simple vectors. Constant vectors can be made by having a type of simple-vector local to the C file containing the constant so that we can use an initialized structure. For example, a constant simple vector containing 5 fixnums could be emitted as C code as follows.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:24:length;
  Obj[5]:body;
} Sv_5

static Sa_5 const1 = {8, 5, {fix(23), fix(2), fix(3), fix(9), fix(-12)}};
```

An example of the translation of a fetch of a simple vector element follows.

```
  Lisp: (setq x (svref y 5))
  C:    x = (Sv *)y->body[5];
```

Though the detail of casting and fetching the body component could be hidden in a C macro, at first we will leave all the details exploded out. One criticism of Chestnut's translations is that no one can figure out what the implementation is actually doing. Exploding out the details will help train development in the details.

2.2 Strings

In order to give fast performance for all strings, we will have fill pointers for all strings in G2. The fill pointer and length will both consume 3 bytes, plus one byte for the tag. The body of the string will be packed directly against the fill pointer, meaning that the most efficient packing of strings into our 8 byte aligned space will be for strings with lengths that have mod (length, 8)=1. Since strings in C must be null terminated, that adds 1 extra byte to the length as compared to the loop length of the string. Having 1 extra byte in the header word will allow us to efficiently pack strings into words when they have lengths that are multiples of 4.

```
typedef struct {
  unsigned int: 8: type;
  unsigned int:24: length;
  unsigned int:24:fill_length;
  char[9]: body
} Str
```

2.3 Immediate integer and double arrays

All of these types are implemented in the same way, with a 4 byte-aligned one byte type tag, a 3 byte length, and then a body. These types are Sa_uint8 and Sa_uint16. Sa_double is similar except that it is 8 byte aligned.. The type for bit-vectors is built on top of Sa_uint8, by fetching bytes and bit-shifting to fetch values and modify values.

2.4 Symbol

There are several different optimizations that one can imagine for the slots of symbols. In the first pass of this implementation, no such memory squeezing will be attempted, and all five typical slots of symbols will be directly provided in the symbol structure. Also, since virtually all symbols are included in packages in G2 and TW, the slots needed for holding the symbol in package balanced binary trees will be included in the symbol itself.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:1:local_value;
  unsigned int:1:external;
  signed int:4:balance;
  unsigned int:1:imported;
  unsigned int:16:name_hash;
  Obj:symbol_name;
  Obj:symbol_value;
  Obj:symbol_plist;
  Obj:symbol_package;
  Obj:symbol_function;
  Obj:package_left_branch;
  Obj:package_right_branch;
} Symbol
```

The symbol-value slot is typically a pointer to the location containing the true symbol value, which will typically be a C global variable. When a runtime generated symbol has a value, the local-value bit will be 1, and the value is directly within the symbol-value slot. Note that slows down explicit calls to Lisp symbol-value since it has to check the local_value bit, but these explicit calls are rare and so I don't think that's so bad.

If packages are implemented as balanced binary trees, using the hash value of the name as an index, then there can be collisions between these hash values. In this case, the symbol names will be alphabetically ordered to determine left, right or match. These choices give us a constant size for symbols, given the size of the symbol-name.

2.5 Compiled Functions

Compiled functions will contain a pointer to the C function for this compiled function, the number of arguments for the function; the number of those arguments that are optional in the Lisp function, and a list of the constants that are default values for the optional arguments. Within the C runtime system, all functions will receive all arguments. If a funcall of a compiled function occurs where some optional arguments are not provided, the default values are extracted from the list of default values. (In circumstances where the default value semantics require more than a constant, the no-arg argument value will be passed, and further computation will happen within the function to implement the default value selection).

```
typedef struct {
  unsigned int:8:type;
  unsigned int:8:arg_count;
  unsigned int:8:optional_arguments;
  Obj:default_arguments;
  void *f(): c_function;
} Func
```

The type spec for compiled functions will typically have arguments of type Obj: and a return type Obj: In special cases we may provide unsafe optimizations for functions that do not have object type arg and return values, but these should never be given to the default funcall operations.

2.6 Characters

Since all immediate type tags are in use, we will use heap allocated character objects. To avoid the need to reclaim them, we will preallocate all 256 character objects (at 8 bytes per, that's 2K). The value is explicitly unsigned to avoid confusion in parts that have a default signed char type.

```
typedef struct {
  double [0]: align;
  unsigned int:8:type;
  unsigned char:value
} Lchar
```

2.7 Package

Packages are implemented as balanced binary trees. They contain a list of used packages, and a symbol (or null pointer) that is the root node of the balanced binary tree. The balanced binary tree of symbol structures will hold only symbols that are local to this package and that have been imported to this package. External symbols of a package are represented by mask bits on the symbol itself. Imported symbols are represented by a new symbol structure in the binary tree, and its symbol value points to the imported symbol. The intern function will be implemented by searching through the binary trees of this package then through the binary trees of used packages and their used packages, but only accepting external symbols.

```
typedef struct {
  unsigned int:8:type;
  Obj:root_symbol;
  Obj:used_package_list;
} Pkg
```

2.8 Obsolete 8 Byte Alignments

[The following section describes tagging for 8 byte alignments. Keeping for historical perspective. -jra 8/30/95]

The exceptions to the pointer to a type tag rule are the following types, which have immediate type tags. These types are fixnums, characters, managed-floats, and conses. If all pointers are 8 byte aligned, then the lower 3 bits of all pointers are always zeros. This gives us 7 non-zero immediate type tags. The following are the immediate type tag assignments:

```
0:  pointer to header of Lisp object
1:  immediate even fixnum
2:  immediate character
3:  pointer to managed-float (mask with -8 for double*)
4:  pointer to cons (mask with -8 for car*, follow for cdr*)
5:  immediate odd fixnum
6:  pointer to double-float (mask with -8 for double*)
7:  unused
```

With this implementation, some determination of type can be made from the printed hex value of the pointer:

```
hex 0 and 8 are pointers to Lisp objects,
hex 1, 5, 9, and D are fixnums,
hex 2 and A are characters,
hex 3 and B are managed-floats,
hex 4 and C are conses,
hex 6 and E are double-floats,
hex 7 and F are unused, so must be corrupted objects
```

By having a cons tag of 4 and a headerless cons, the access to one of the elements of the cons will be a straight pointer dereference. The other requires subtracting 4 from the pointer first. (On the Alpha OSF or other 8 byte pointer, machines, some address computation is necessary for both the car and cdr, unless we make object a 4 byte wide data type, which we could do).

The type tags for heap allocated data structures have no special issues, except that these values should not collide with any immediate type tags (i.e. be greater than 7), and be able to quickly determine if a data structure is reclaimed (i.e. use bit 7 as a flag).

So the type tag table is as follows:

```
Tag (Hex) Lisp Type C Type  Reclaimed Tag
------------- --------- ------  -------------
0 00 immed pointer (Header *)Obj
1 01 immed even fixnum (sint32)Obj>>2
2 02 immed character (char)(Obj>>2)
3 03 managed-float (double *)Obj
4 04 cons (Obj *)Obj-1
5 05 immed odd fixnum (sint32)Obj>>2
6 06 double-float (double *)Obj
7 07 unused
8 08 simple-vector (Sv *)Obj  136 88
9 09 string (w/fill ptr) (Str *)Obj  137 89
10 0A (simple-array ubyte 8) (Sa_uint8 *)Obj  138 8A
11 0B (simple-array ubyte 16) (Sa_uint16 *)Obj 139 8B
12 0C (simple-array double) (Sa_double *)Obj 140 8C
```

```
13 0D symbol (Sym *)Obj  141 8D
14 0E compiled-function (Func *)Obj  142 8E
15 0F package (Package *)Obj  143 8F
```

Note the use of a zero length double array at the beginning of the struct. This ensures double word alignment, which gives us two things. The first is that pointers to constant structures of this type will be on 8 byte boundaries, required for our use of 3 bits of immediate type tag. The second benefit is that with a known alignment we can reliably mix objects and immediate doubles in simple-vectors. This is needed for structure and frame extensions I would like to make. The Harbison and Steele manual suggests that this zero length double array technique will ensure alignment, but this has not yet been tested.

Index

E

R

Table of Contents