Representing Type Information in Dynamically Typed Languages

David Gudeman

gudeman@cs.arizona.edu
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, AZ 85721, USA

TR 93-27
October 1993

Abstract

This report is a discussion of various techniques for representing type information in dynamically typed languages, as implemented on general-purpose machines (and costs are discussed in terms of modern RISC machines). It is intended to make readily available a large body of knowledge that currently has to be absorbed piecemeal from the literature or re-invented by each language implementer. This discussion covers not only tagging schemes but other forms of representation as well, although the discussion is strictly limited to the representation of type information. It should also be noted that this report does not purport to contain a survey of the relevant literature. Instead, this report gathers together a body of folklore, organizes it into a logical structure, makes some generalizations, and then discusses the results in terms of modern hardware.

Contents

1 Introduction

In a statically typed language it is not a problem that each floating point number has a representation that is identical to the representation of some integer, because the type of representation is known at compile time and the compiler can generate the right code to deal with the representation. In a dynamically typed language however, the machine representations of data objects cannot, in general, be determined at compile time because there is nothing in the syntax of the language to tell the compiler what type of data object is being represented. In such languages, it is necessary to include information at runtime to distinguish every value from every other value. Theoretically, the only requirement for dynamic typing is that each representation must represent a unique value, regardless of type. Practically, the representation must be such that it is possible to efficiently convert between machine representations and the uniform representation, and to quickly determine what sort of object is being represented.

This report is a general discussion of representation schemes —tagging and otherwise— and the various trade-offs that are available. Speed is the major concern, but portability, space requirements, and register usage are also discussed. This is not a theoretical treatment, but a down-and-dirty discussion of real machines and real costs of real implementations. Dynamic type operations are frequent enough that a trick to save just one machine cycle can have noticeable effects on the performance of a set of benchmarks, and so this paper deals seriously with such tricks. Be aware that since this is a discussion of implementations, the word "type" is used here to mean an implementation type (an encoding of some set of abstract values into machine words) and not an abstract type.

One should also be aware that there is no general agreement in terminology about representational issues. This document introduces and defines many terms, but it should not be understood from such a definition that the term being defined is actually used that way in all of the relevant literature. In fact several new terms have been introduced to talk about things that previously had no names. Also, those terms that are used in the literature are often defined to make it easy to talk about a specific implementation or a specific language, and as such they tend to be unnecessarily restricted. In several such cases, this document offers a generalized definition of the term that leaves its meaning unchanged in its previous incarnations and at the same time makes it useful for more general discussions. A glossary is provided at the end.

The process of converting from a statically typed representation of a value to a dynamically typed representation is referred to as "wrapping" the value, and the process of converting back to a statically typed representation is referred to as "unwrapping". The representation of a wrapped value is called a "wrapper". Actually the wrapper is only the minimal representation used to encode a piece of data —the part that is passed to procedures, returned from procedures, etc. If the data value needs a large amount of memory to represent, then it is usually encoded in a special block of memory allocated for it, and the wrapper only contains a reference to this block of memory. Values that are represented in this way are said to be represented "indirectly" and are called "indirect" values. Other values are said to be represented "directly" and are called "direct" values [1]. In some of the literature, the operation of creating an indirect wrapper to a number is called "boxing". For example, an integer is boxed if it is represented by a tagged pointer to an integer, it is not boxed if it is represented directly. We will avoid this terminology to avoid possible confusion between the meanings of "wrap/unwrap", "tag/untag", and "box/unbox".

Given a data element of n bits, it is possible to represent 2^n different values. It is important to note that this means 2^n different values total, not 2^n different values of each type. The technique of implementing objects indirectly (by pointers) does not actually increase the number of objects that can be represented at one time, rather it has the effect of defining these representations dynamically. That is, the same data element can represent different values at different times because it is a pointer to a block

[1] *Direct* values should not be confused with *immediate* values which are compile-time constants represented in the machine code.

that can change.

There are several broad categories of techniques for wrapping data:

Tagged Words. Machine words are divided into a tag field and a data field. For indirectly represented objects, the value that is represented in the machine word is a pointer to the object.

Partitioned Words. The set of bit patterns that a word can represent is divided up among the types so that each type must restrict its values to those that can be represented by the bit patterns allocated to the type.

Object Pointers. A wrapped value is simply a pointer to a self-identifying block in memory (usually on the heap) and the type of the block is encoded in the block itself (for example the first word of every block might be an integer type code).

Large Wrappers. A wrapper consists of two or more words so it is large enough to represent a full-sized machine value as well as a type code.

Typed Locations. The type of a value is determined by where the value is stored. Static typing is a special case of this technique.

Hybrid Representations. A combination of the above techniques.

This paper discusses these representation strategies, their costs, and the various implementation tricks that can be used to reduce or redistribute the costs. Costs are discussed in terms of three generic operations performed on wrapped values:

`is_T(v)` — test a dynamically typed value `v` to see if it is of type `T`. This is probably the most common operation, but it does not fully reflect the cost of identifying dynamic types. Section 8, "**Dispatching on Dynamic Type**" discusses this in more detail. If `t` is a type name, then we use `t_T` to name the integer type code for that type (if there is such a code).

`wrap_T(v)` — given a machine representation `v` of some value, produce the appropriate dynamically typed value of type `T`.

`unwrap_T(V)` — given a dynamically typed value `V` with type `T`, produce the machine representation of `V`. This operation should be the inverse of `wrap_T` so that `wrap_T(unwrap_T(V))` = `V` and `unwrap_T(wrap_T(v))` = `v`, as long as `V` and `v` are in the appropriate domains.

Operations are described as C macros, so the term "produce" as used above has two meanings. When viewed as a C macro, a value is produced by making it the result of evaluating the macro. Viewed at a lower level, a value is "produced" simply by putting it into a register. One extension has been added to C for the purposes of this paper: binary numbers are given in the form `0b`n where n is a sequence of binary digits (`0` and `1`) representing a binary number. To increase readability, the macros are written without the extra parentheses that are needed in C.

The cost of an operation is the number of machine cycles that would be required to execute the operation on a generic RISC machine. Logical and arithmetic instructions are assumed to operate on registers and to require one cycle. Loads and stores take two cycles: one to load or store the word, and one to fetch and decode the load/store instruction itself (we ignore cache affects). The values being operated on are assumed to be in registers both before and after the operation. A general comparison is two cycles, one instruction to set a condition code and a second instruction to test the condition and (maybe) branch.

Many RISC machines have delayed branches that execute one or more instructions in the normal instruction stream after encountering a branch, before actually branching. These extra instruction positions after a branch are called "delay slots", and they complicate cost analysis considerably, because if these slots can be filled with instructions that are useful for the case where the branch fails as well as for the case where the branch succeeds, then the branch only costs one cycle. On the other hand if this slot cannot be filled with such an instruction, then the branch costs more. If a branch has a `nop` (for "no operation") instruction in a delay slot, then an extra cycle should be added to the cost of the branch. If a delay slot contains an instruction that is useful in one case but not another, then some fraction of a cycle should be added to the cost, depending on the frequency of the cases. For the `is_T()` macros we will assume that the success case is "expected" (if success is not expected, then the test is probably part of a dispatch, which is discussed in section 8). In this case the delay slot can generally be filled with an instruction from the expected control path, and so we assume it is filled. For other sorts of branches such as numeric tests we do not make this assumption.

A "word" is the size of object that fits in a general-purpose register, and the examples all assume that a word is 32 bits. An immediate operand that has non-zero bits in the upper half of the word adds a cost of 1 cycle if all bits in the lower half of the word are zero and 2 cycles otherwise. Such an operand is too big to fit in the instruction word with the other operands, so it either must be constructed at run-time or it must be fetched from the instruction stream after the current instruction, requiring another memory fetch cycle. Generally such constants are constructed by using a special "load-high" instruction that sets the high bits of a word, and then adding the low bits with an add instruction. If the constant has all zero bits in the lower half of the word, then it can be constructed with a single load-high instruction so it is given a cost of one cycle (which is also correct for a machine with full word-sized immediates). If the constant has non-zero bits in the lower half then it requires two cycles to construct on machines without full word-sized immediates and this is the cost used. This cost is correct for most RISC machines but not for machines that have full word-sized immediates. One RISC architecture, the SPARC, only has 13-bit immediates (less than half a word), but the costs given in this discussion are still applicable to SPARC machines.

Some systems speed up the use of large constants by dedicating general-purpose registers to hold them, but such techniques will not be considered in this report. The reason is that these techniques make one specific job faster at the cost of adding an overhead to the rest of the execution. In this case the implementor is taking a known cost (the cost of building a constant at runtime), associated with a known operation, and amortizing that cost over all of the rest of the execution by restricting register availability. Subsequently, the code that pays the cost of using these large constants may have nothing to do with them, and a particular program that makes little or no use of the constants still pays the cost of having them available. It is quite difficult to measure the cost of this technique, and quite difficult to show that it is actually an optimization.

Bits in a word are numbered from most significant to least significant. In other words, the most significant (or sign) bit is numbered 0, and the least significant bit is numbered 31 (in a 32-bit word). The least significant end of the word is called the "low" end, and the most significant end is called the "high" end.

A "cons cell" is a pair of words used to implement a linked list. Cons cell operations are common in Lisp and Prolog, and so this type is used frequently in examples.

2 Tagged Words

The most common way to represent dynamically typed data in Lisp and Prolog implementations is by the use of a tag field. In this representation each wrapped value is a single machine word and each machine word is viewed as a sequence of bits. The sequence of bits is divided up into one or more tag fields for encoding the type and (usually one) value field for encoding the data. This means that each value must be represented in a smaller number of bits than there are available in a machine word, so that in general, not all unwrapped values can be represented as wrapped values. However this technique leads to very compact representations and fairly good access times.

Another restriction of tagging is that the number of types must be severely restricted in order to leave enough value representations of each type. This is not a problem if there are only a few built-in types, but most modern languages provide for user-defined types[2], and there can be arbitrarily many of these. Tagged word representations generally allocate one tag to represent all user defined types together, and use the object pointer method (section 4) to distinguish among them. This 2 stage representation has additional overheads which should be taken into account.

Typically there is one field for the value and one field for the tag (some systems add a few more fields for garbage collection, cdr-coding, and such things). The operation of extracting the value field of a tagged word is referred to as "untagging", and the operation of constructing a tagged value by setting the tag and value portions of a word is referred to as "tagging" the value. In some cases (un)tagging is identical to (un)wrapping and in some cases it is not.

In the following, examples will generally assume a 4-bit tag field, and shifts are always unsigned unless indicated otherwise.

2.1 Tagged Pointers

There are different concerns when tagging different types of objects such as pointers, other unsigned values, integers, and floats. The difference between pointers and other unsigned values is minor so these two types are discussed together, but integers and floats are discussed separately in their own sections.

The tag field can either be placed in the high (most significant) or low end of a word, and there are advantages to each choice. Suppose the machine has a 32-bit word and an address space of 2^{32} bytes. If we restrict our pointers to the lower 2^{28} bytes of memory, then the upper four bits of the pointers will always be `0` so we can use these bits for tags. The following C macros might be used for wrapping, unwrapping, and testing:

```
#define is_T(v)     (v>>28) == t_T
#define wrap_T(v)    v | (t_T<<28)
#define unwrap_T(v) (v << 4) >> 4
```

where `v` is the value to be wrapped or unwrapped, and `t_T` is the tag code associated with the type `T`. Testing the type costs 1 cycle for the shift and 2 cycles for the compare for a total of 3 cycles.

Given that `t_T` is a constant, so is `(t_T<<28)`, so the cost of wrapping would appear at first to be 1 cycle, since the only runtime operation is the the bitwise-or. However, the constant `(t_T<<28)` is too large to be represented in a 16-bit integer, so it adds a cost of one cycle for the load-high (as discussed above) and wrapping costs 2 cycles.

Unwrapping is done with two shifts for a cost of 2 cycles. The alternative implementation

```
#define unwrap_T(v) v & 0x0FFFFFFF
```

costs 3 cycles on most RISC machines because `0x0FFFFFFF` is constructed in two cycles with a load-high followed by an addition. On a machine that has full word-sized immediate operands, the cost of this method is the same as two shifts.

Instead of putting the tags directly in the upper four bits, we can use a 4-bit shift and put the tags in the lower four bits. The macros would look like this:

```
#define is_T(v)     (v & 0b1111) == t_T
#define wrap_T(v)    (v << 4) | t_T
#define unwrap_T(v) v >> 4
```

The costs for these operations are listed in table 1, which compares the costs of the using low bits vs. using high bits for the tag. From this table it appears that it is better to use the low end of the word for

[2] The functor and arity of a Prolog terms is considered here a sort of "user defined" type.

Tag Operation	Low-bits Tag	High-bits Tag
`is_T`	3	3
`wrap_T`	2	2
`unwrap_T`	1	2

Table 1: Costs of High End vs. Low End Tags

a tag field, but there are more considerations. In particular, the numbers in the table are for operations on general type codes but there are various optimizations for special type codes. For example, tags in the low end that have only one non-zero bit can be tested in only two cycles by masking with the tag value and then comparing to zero, assuming that comparison to 0 can be done in one cycle:

```
#define is_T(v)    (v & t_T) != 0
```

For high-end tag fields, the pointer that is given the tag `0b0000` does not need to be wrapped and unwrapped because the tagged representation is the same as the untagged representation. This cannot always be done if tags are kept in the lower end of the word; since the word must be shifted for tagging there is never a type of tagged value that is identical to its untagged representation. However, many machines and many implementations have alignment restrictions on certain kinds of pointers, and in such cases some number of the least-significant bits will always be `0b0`. These lower bits can be used for tagging without shifting the value, and the one tag of all zeros can be tagged and untagged for free. For example, on byte-addressable 4-byte-per-word machines the lower two bits of an aligned word pointer are always equal to `0b00` and these two bits can be used for up to four tags.

An implementation can enforce additional alignment restrictions to increase the size of this field. For example in Monaco [4], a concurrent implementation of FGHC, values are represented with two words, one for a tagged value and one for a semaphore (to prevent race conditions on changing the value). Since all values need two words anyway, the Monaco system loses nothing by allocating all values on two word boundaries, so they can use a 3-bit tagging scheme with no need to shift pointer values. Some Lisp implementations (for example [16]) enforce a two word allocation boundary for this reason even though it wastes some space. Instead of enforcing new alignment restrictions, an alternative is to just represent a few pointer types without shifting, and to shift the rest (see section 2.4).

Many machines have an addressing mode that will automatically add a small constant to an address (register) as part of a load or store. In this case, pointers that are tagged without shifting can be untagged for free as long as the unwrapped value is only used to access memory, and as long as this is done immediately after unwrapping the value. For example if word-aligned cons cells are given the 2-bit tag `0b01`, then the operations to access the fields of a cons cell can be defined as

```
#define get_car(v) *(word*)((char*)v-1)
#define get_cdr(v) *(word*)((char*)v+3)
```

In these macros, `v` is first cast to a byte-pointer so that the addition is not scaled. After the addition the pointer is re-cast to a pointer to a word. If the machine has load and store indirect-with-immediate-offset instructions, then the above instructions each take only two cycles (fetching and decoding the instruction is one cycle, loading the value from memory is another cycle). Compare this to

```
#define get_car(v) *(v>>2)
#define get_cdr(v) *((word*)(v>>2)+1)
```

which takes 3 cycles for each instruction (fetch and decode, shift, load value). If the machine does not have the load and store indirect-with-immediate-offset instruction, then the tag value can be chosen such that when the wrapped value is viewed as a pointer, it points to the most frequently accessed field.

The scheme that puts the tag field in the low end of the word, then, can get free wrapping and unwrapping for one value, and sometimes-free unwrapping for several other values. The restricted size of the tag field for this technique can be a problem, but this can be handled by having a staged representation (section 2.4).

2.1.1 Machine Values

If a value has a representation that the hardware machine can load into a register and operate on with special instructions (such as integers and floating point values) then this value is a "machine value". Since the operations of wrapping and unwrapping involve translating from and to machine representations, if such a value is represented by a tagged pointer instead of a direct wrapped value, then wrapping and unwrapping incur costs for referencing memory. If a tagged pointer references such a machine value, then the operation of unwrapping the tagged pointer involves not only untagging the pointer, but loading the referenced value into a register. Wrapping such a value requires allocating storage for it and storing it to memory as well as tagging a pointer to the newly allocated location. The operations are

```
#define wrap_T(v) (alloc_T, *new = v, tag_T(new))
#define unwrap_T(v) *(T*)untag_T(v)
```

where `tag_T()` and `untag_T()` are macros to tag and untag pointers, and incur the costs for these operations as given above. There is a potential confusion in this terminology. For example, if the machine values being wrapped are integers, then the operation of "unwrapping" an integer involves *untagging* a tagged pointer to the integer and then loading the integer. In other words, untagging is not the same as unwrapping in this instance. This is because a *wrapped* integer is represented by a *tagged* pointer to an (unwrapped) integer.

The reason it is important to distinguish between machine values and other sorts of values in this way is because it is possible to wrap machine values directly (thereby restricting the range of those values). Since direct representations can encode these values without allocating extra storage or referencing memory, it is necessary to discuss all techniques in such a way that the extra costs are taken into consideration.

2.2 Tagged Integers

Integers are often represented as direct wrapped values. However, (signed) integers are more difficult to represent this way than pointers or unsigned values because the upper bits all depend on the sign of the integer (at least on two's complement machines, which are now almost universal). Consequently, it is not possible to just restrict the range of integers in order to get zero bits at the high end of the word as it is for unsigned values. Because most machines provide a signed shift, it *is* still possible to put integer tags in the lower end of the word. All that is needed is to make sure the `unwrap_int()` macro does a signed shift, that is a shift where the sign bit is shifted in from the left. Many C compilers for machines that have a signed-left-shift instruction will do a signed shift on a signed quantity. However neither K&R nor the ANSI C standard requires that a shift on a signed value be a signed shift [11], so the following macros cannot be considered portable.

Words are unsigned, so to get signed shifts (from cooperative compilers) we cast the wrapped value to a signed quantity before shifting:

```
#define unwrap_int(v)      (int)v >> 4
```

This will cause the sign bit to be shifted in from the left, reversing the effect of the tagging operation, given that the integer is within the restricted range. If a signed shift costs no more than an unsigned shift, the cost of operations will be the same for signed quantities as for unsigned quantities (for a tag field in the low end). Also, if integers are given a two-bit code `0b00`, then they can be added to word pointers directly to get offset addresses. By contrast, integers in machine representation must be shifted two bits to the left before doing address arithmetic on word pointers.

It is more complicated to put integer tags in the high end of the word, the operations would look something like this:

```
#define wrap_int(v)    ((unsigned)(v << 4) >> 4) | (t_int<<28)
#define unwrap_int(v)  (int)(v << 4) >> 4
```

where `t_int` is the 4-bit tag for integers. The `wrap_int()` macro works by first shifting the upper four bits to `0b0000`, then attaching the `t_int` tag, for a cost of 4 cycles. Unlike pointers, it is not legitimate to assume the upper four bits are already `0b0000` because if the integer is negative, the upper four bits will be `0b1111`. If the tag `0b1111` or `0b0000` is used for integers, then the shifting can be avoided, reducing the cost to 2 cycles (one load-high and one bitwise-or or bitwise-and). However this makes the special tag unavailable for other objects, and since it is a specialization it will be discussed separately. The left shift in the `unwrap_int()` operation has two effects: it removes the `t_int` tag by shifting it out, and it moves the sign bit of the tagged integer into the position of the sign bit for the word. Then a signed right shift is done to shift in the correct sign bits.

One way to reduce the cost of wrapping and unwrapping integers in the high end of the word is to use the same representation for tagged integers as for untagged integers. Unfortunately, this requires assigning two different tags to integers, since non-negative integers have zeros in the high bits and negative integers have ones in the high bits. This scheme is "tagging integers by sign extension" and the pair of tags is referred to collectively as the "sign extension tag". Note that this scheme also rules out the use of `0b0000` as a pointer tag.

The problem with tagging integers by sign extension is that it makes the `is_int()` operation more expensive. There are several possible implementations ([20] discusses some options not discussed here). The most obvious one is to test the two in sequence:

```
#define is_int(v) (v>>28) == 0b0000 || (v>>28) == 0b1111
```

The cost of this is 1 for the right-shift (assume that this is done once and the result is cached in a register) and 1 for the first conditional branch (assuming the machine has a jump-if-zero instruction, the delay slot is filled with the compare for the second test) and another 2 cycles for the second conditional branch if the first is not taken. The total is 4 cycles (if `v` is not an integer the cost is 2 cycles, but this is not the expected case).

An alternative is to sign-extend `v` and see if the result is equal to `v`:

```
#define is_int(v) ((int)(v<<4) >> 4) == v
```

The cost of this is 4 cycles, which is equal to the worst-case time of the previous technique. But if the machine has no jump-if-zero instruction (for example, the i860 and the SPARC), then the 2-jump technique costs more. A third option that takes only three cycles but appropriates the sign bit is discussed in section 3.1.1.

Table 2 summarizes the costs of the various integer tagging schemes.

2.3 Arithmetic on Tagged Integers

Table int-tag-costs does not fully reflect the costs of using tagged integers because many uses of tagged integers are in arithmetic operations, and the cost of these operations may vary depending on the tagging scheme. In general, when doing arithmetic on tagged integer values, it is not necessary to fully untag the operands and to tag the result. The macro to do addition in this naive way would look like this:

```
#define ti_add(i,j) wrap_int(unwrap_int(i) + unwrap_int(j))
```

Tag Operation	Low-bits Any	Low-bits Zero	High-bits Any	High-bits Zero	Sign Extension
`is_int()`	3	2	3	2	4
`wrap_int()`	2	1	4	2	0
`unwrap_int()`	1	1	2	2	0

Table 2: Costs of Integer Tag Operations

Machine integers that come from tagged integers have fewer bits than machine integers, but generally the machine does arithmetic on full-sized machine integers. There are two ways to do an n-digit integer operation with integers that are fixed to have $n + m$ digits ($m > 0$), and to get an n digit result. The "excess-precision" method is to do the arithmetic normally, and if the result requires more than n digits, then there has been an overflow. This technique requires that the overflow be handled explicitly. An alternative is the "scaled " arithmetic method, which involves scaling the integers before the operation such that an overflow of the n-digit operation will result in an overflow in the $(n + m)$-digit operation as well. The advantage of this approach is that overflow is handled implicitly.

It assumed that one of two things is done when an operation overflows: either the overflowing digits are truncated, or an exception is raised. In either case, if an arithmetic operation causes an overflow of the machine word, then it is assumed that the machine does the right thing (either truncates or causes an interrupt), and that there is no cost in cycles to test for an overflow. In this case overflow is said to be "handled implicitly". If arithmetic is done in the low end of the machine word, and if the result requires too many bits to be represented as a tagged integer, then some specific check must usually be taken after every operation that might overflow, and the cost of these checks must be added to the cost of arithmetic. In this case overflow is said to be "handled explicitly".

2.3.1 Arithmetic on Integers with High-End Tags

If the tag field is in the high end of the word, then the integer itself is already in the low end of the word. All that is needed to implement excess-precision arithmetic is to remove the tag. Of course this is the advantage of tagging integers by sign extension —no masking is necessary. However it is necessary to check for overflow explicitly, with an `is_int` test, and this is a rather expensive test as shown above. The cost of each arithmetic operation that might overflow is 4 cycles plus the cost of the operation itself. This adds 4 cycles to the cost of addition, subtraction, and multiplication.

Scaled addition on high-end tags can be done with

```
#define ti_add(i,j) wrap_int(((i<<4) + (j<<4)) >> 4)
```

which saves one right shift by not shifting the two arguments back to the right, but shifting the result instead. This macro also provides for overflow to be handled implicitly, saving that overhead. Recall that tagging an integer with an arbitrary high-end tag is expensive (4 cycles) because the upper bits have to be shifted out first. However, the result in this case starts out in the high end of the word and must be right-shifted to position. If this right shift is unsigned, then zeros will be shifted in and no mask is necessary. So the above can be further optimized to

```
#define ti_add(i,j) ((unsigned)((i<<4) + (j<<4)) >> 4) | (t_int<<28)
```

where `t_int` is the integer tag. This operation costs 6 cycles. If `t_int` = 0 then the constant `(t_int<<28)` need not be constructed and the mask need not be done, so the cost is only 4 cycles. If integers are tagged by sign extension then the final right shift is done with a signed shift, and no masking is needed. Again, the cost is 4 cycles. The same reasoning applies to subtraction as to addition, so the costs are the same.

When integers are tagged by sign extension, integer negation is done with a machine integer negation, so the cost is 1 cycle. For other tags in the high end of the word, it is not necessary to unwrap a tagged integer before negation, but the result must be re-wrapped as an integer. The cost of this is 1 cycle plus the cost of wrapping an integer.

For an integer i represented with a tag in the high end, left-shifting by n bits (where n at least as large as the number of bits in the tag field), produces a machine representation of the untagged integer $2^n i$. This fact and the identity that $(2^n i)j = 2^n(ij)$ says that to do scaled multiplication, it is not necessary to fully untag both operands, one of them can just be left-shifted:

```
#define ti_mul(i,j) wrap_int(((i<<4) * unwrap_int(j)) >> 4)
```

This operation costs two shifts in addition to the one tag operation, one untag operation, and the multiplication. If integers are tagged by sign extension or with the tag `0b0000`, then the `wrap_int` is not necessary as long as the correct sort of right shift is used (signed or unsigned).

For division, note that $(16i/16j) = i/j$, so the implementation is

```
#define ti_div(i,j) wrap_int((i<<4) / (j<<4))
```

In this case, the `wrap_int` can only be avoided if integers are tagged by sign extension since there is no right shift done at the end.

The final integer operation considered is signed comparison. A test for equal or not-equal is always the same, regardless of representation (assuming all representable integers have just one wrapped representation), but the cost of testing for less-than, less-than-or-equal, greater-than, or greater-than-or-equal depends on the representation. Let us refer to these comparisons collectively as inequalities. Clearly, if integers are tagged by sign extension then integer inequality tests can be done directly with machine inequality tests. As mentioned above, we assume one instruction to set a status bit and one to branch conditionally, for 2 cycles (we will ignore delay slots when discussing inequality tests since the different techniques will not affect how they can be filled). If integers are given any single tag in the high end of the word, then the most efficient way to do an inequality test on two tagged integers is to shift them both to the high end of the word for a scaled comparison. In this case there are two extra cycles for the shifts, giving a total of 4 cycles.

2.3.2 Arithmetic on Integers with Low-End Tags

Like high-end tags, low-end tags can be optimized for arithmetic also, but there seems to be no trick for doing excess-precision arithmetic in such a way that saves cycles over the naive method of untagging the operands and tagging the result. Since excess-precision also requires an explicit overflow test at the end, it is more expensive than scaled arithmetic, so it will not be considered.

If integers are given a tag of all `0`'s in the low end of the word, then scaled addition and subtraction are free, and multiplication and division have only one cycle of overhead:

```
#define ti_add(i,j) i + j
#define ti_sub(i,j) i - j
#define ti_mul(i,j) i * (j>>4)
#define ti_div(i,j) (i/j)<<4
```

If a different tag is used, then the C operation `(`i`<<4)+t_int` is arithmetically equivalent to $16i+$`t_int`. So, since

$$-(16i + \texttt{t\_int}) + 2\texttt{t\_int} = 16(-i) + \texttt{t\_int}$$

negation can be implemented by

```
#define ti_neg(i)   -i + 2*t_int
```

as mentioned in [19]. Similarly, since

$$(16i + \texttt{t\_int}) + (16j + \texttt{t\_int}) - \texttt{t\_int} = 16(i + j) + \texttt{t\_int}$$

addition can be implemented by

```
#define ti_add(i,j) i + j - t_int
```

and since

$$(16i + \texttt{t\_int}) - (16j + \texttt{t\_int}) + \texttt{t\_int} = 16(i - j) + \texttt{t\_int}$$

subtraction can be implemented by

```
#define ti_sub(i,j) i - j + t_int
```

Note that `2*t_int` is a compile-time constant. The first `ti_add` and `ti_sub` given in this section are just special cases ($\texttt{t\_int} = 0$) of the ones given here.

If $i < j$ then $mi + b < mj + b$ for any positive number m and any number b. Since tagging in the low end is arithmetically equivalent to multiplying by a positive number and adding a number, comparisons can be done on the tagged integers without unwrapping them first, so comparison on wrapped integers is done directly with a machine comparison at a cost of 2 cycles.

The costs of integer arithmetic for the various tagging schemes are summarized in Table 3. The

Operation	Low-bits Any	Low-bits Zero	High-bits Zero	Sign Extension
negate	2	1	3	1
add/subtract	2	1	4	4
multiply	$4 + C_m$	$1 + C_m$	$4 + C_m$	$2 + C_m$
divide	$4 + C_d$	$1 + C_d$	$5 + C_m$	$2 + C_d$
inequality	2	2	4	2

Table 3: Costs of Tagged Integer Operations

constants C_m and C_d are the numbers of machine cycles needed for a multiply and a divide, respectively. From tables 1, 2 and 3 it appears that there is a noticeable advantage to tagging values in the low end of the word rather than the high end.

2.3.3 Testing After the Operation

It has been suggested [20] that addition of tagged integers can be improved by using an integer tag such that when two tagged integers are added directly, the result is the correctly tagged integer result of the operation (tagging integers by sign extension or by all zeros in the low end) and choosing the other tags in such a way that when adding two tagged values that are not integers, the result cannot look like an integer. This scheme allows the compiler to add two numbers without checking their types first, and to find out if they were integers after the fact by checking the type of the result. In the (supposedly most common) case where both operands where integers, this is a bit faster. On the other hand, it requires at least one more bit added to the tag field, and this technique will make addition of other numeric types more expensive. In order to be effective, this technique seems to require that integer addition be an extremely common operation, that type-checking integers be expensive, and that addition on other types be rare.

2.4 Staged Tags

A staged tagging scheme uses tags of different sizes for different types. For example it is possible to use three different 2-bit tags, as long as at least one 2-bit code is reserved to be a "staging tag". The other 2-bit codes are called "first-stage" tags, and any value that has a lower two bits equivalent to a first-stage tag has the type associated with that tag. A tagged value that has the first two bits equal to a staging tag has a second-stage (or higher) tag. Suppose that `0b11` is used for the staging tag, and that the larger tags use three bits in addition to the two used in the smaller (first stage) tags. There are then two classes of tagged values, which might be diagrammed like this (All `X`'s and `1`'s represent bits):

first-stage values:

	XX

second-stage value:

	XXX11

A tagging scheme that does not have different stages is called "1-stage", or "unstaged". A scheme that has more than one stage is called "staged", and if there are n stages then it is an n-stage tagging scheme. A 2-stage tagging scheme that uses m bits for the first-stage tag and another n bits for the second stage tag is called an $m+n$-bit tagging scheme, and similarly for schemes with more stages. Note that there could be more than one staging tag at a given level. For example, both of the 2-bit tag codes `0b10` and `0b11` might be staging tags that signal that more bits are used in the tag.

The $2+3$-bit scheme outlined above removes 5 bits from the address space of second-stage tags, and this may be considered excessive. If so, second-stage word pointers can be tagged by shifting just three places to take advantage of the two bits in the lower end of a word pointer that are always `0b00`. The cost of this technique is that untagging the pointer is now done with a 3-bit right shift, and this does not shift out all of the tag bits (except for the pointers that have tags of the form `0b00xyz`) so these tag bits must be subtracted or shifted out. If the pointer is being untagged just for dereferencing, then this extra step is subsumed in the store- or load-indirect-with-offset.

The most serious problem with staged tagging schemes is that extracting the tag as a separate value is expensive: in the worst case the cost is one mask (1 cycle) for each stage and one test (2 cycles) for each stage except the last, for a total cost of $s+2(s-1) = 3s-2$ cycles where s is the number of stages. In the scheme outlined above, $s = 2$ so the cost to extract a tag is as high as 4 cycles, which is four times the cost with a 1-stage tagging scheme.

The major reason to extract a tag as a separate value is to "dispatch" on the type; that is, to branch to a different address for each different type. For example a test to determine the equality of two values needs to compare different types of value in different ways. By contrast an operation to extract the car of a wrapped value representing a cons cell only needs to distinguish between the case where the wrapped value actually represents a cons cell and the case where it does not. For a test that only needs to determine if the type is `T` or not, the operation `is_T()` can be used, and this is no more expensive for staged tags than for unstaged tags. For example, if cons cells have the second-stage tag `0b10111` in the low end of the word (where `11` is the staging tag, as discussed above) then the implementation would be

```
#define is_cons(v) (v & 0b11111) == 0b10111
```

for a cost of 3 cycles. This is the same cost as testing an unstaged tag in the low end of the word because there is no need to check the staging tag separately.

So the penalty for using a staged tagging scheme depends on the dynamic distribution of tests in executions of programs in a language. If most tests are of the `is_T` variety, then there is little overhead for staged tags. If most tests are of the dispatching variety, then the overhead depends on the implementation of the tests. This in itself is a topic worthy of its own section (8), and is discussed later.

2.4.1 The Sign Bit as a Tag Field

For most bits in the top half of a machine word, on many machines, testing whether the bit is on or off requires 3 cycles: mask or shift out all of the other bits (2 cycles) and jump-if-zero. In contrast, the sign bit can often be tested in one or two cycles. This suggests the possibility of using the sign bit in a staged tagging scheme where one data type could be given a tag based on the sign bit [17]. For example, if the staged tag fields are in the high end of the word, cons cells might be given the 1-bit tag `0b0` in the highest bit, and all other types would be assigned 4-bit tags of the form `0b1`xyz. Then the test for `is_cons` would take only 1 or 2 cycles instead of 3, and wrapping and unwrapping cons cells would be free.

In this example, if the tag `0b1111` is given to a pointer, that pointer can be tagged and untagged by negation instead of by shifting and masking. For instance, if the tag is given to unbound variables, the operations would be

```
#define wrap_var(p) -p
#define unwrap_var(v) -v
```

In order for this to work it is necessary to ensure that negating a pointer will always produce `0b1111` in the high four bits of the word. Taking the two's complement negative of a 32-bit signed number i is equivalent to subtracting i from 2^{32} as an unsigned operation, so the condition can be satisfied by ensuring that $2^{32} - p \geq$ `0xF0000000` or that $p \leq 2^{28}$. Since the upper four bits are being used for a tag field, pointers are limited to this range anyway.

2.5 Distributed Tag Fields

There is no *a priori* reason why the tag field (or the data field for that matter) needs to be contiguous, and there are a number of interesting strategies that have one tag field in the low end of the word and another tag field in the high end. For example, the implementation of NIL described in [5, pp 54–57] uses a tagging scheme where integers are given the tag `0b00` in the low end of the word and the other three 2-bit codes in the low end of the word represent staging tags. The second stage tags all use the 2 low bits as well as the three high bits in the word. Since the high three bits are not used to tag integers, integers can be tagged and untagged by shifting alone, and arithmetic on integers is as described above for tagged integers that use a tag of all zeros in the low end of the word. With this scheme, three more pointer types can be given a distributed 5-bit tag with `0b000` in the upper end of the word, and these pointers can be dereferenced without untagging by using a store- or load-indirect-with-offset.

However type-checking is quite expensive with this technique. The simple type test

```
#define is_T(v) (v & 0xE0000003) == t_T
```

for types other than integers costs 7 cycles, because both the constants `0xE0000003` and `t_T` each cost 2 cycles to construct. Alternatively, the two separate tests can be done:

```
#define is_T(v) (v & 0b11) == t_T1 && ((v>>29) & 0b111) == t_T2
```

At a cost of 6 cycles (on success, the delay slot of the first jump is filled by the shift). The constants `t_T1` and `t_T2` are the two parts of the tag. When either field has just one non-zero bit then that field can be tested in 2 cycles, so some values can be tested with costs of 4 and 5 cycles. The tag could be smashed into a small constant for comparison by shifting the top three bits down:

```
#define is_T(v) ((v>>30)&0b11100) | (v&0b11) == t_T
```

but the cost is still 6 cycles. One masking cycle can be saved by taking advantage of the zeros that get shifted in. This requires reversing the 2 low bits with the three high bits:

```
#define is_T(v) ((v>>29)|(v<<2))  == t_T
```

and the cost is 5 cycles. This test reduces the maximum cost for testing these distributed tags to 5 cycles.

Table 4 compares the two-ended two-stage tagging scheme with a one-ended two-stage tagging scheme. All columns in the table represent a 2-stage scheme with 2+3-bit tags where integers get the tag `0b00`

Tag Operation	Low-bits Shift 5	Low-bits Shift 3	Two-ended
`is_int()`	2	2	2
`wrap_int()`	1	1	1
`unwrap_int()`	1	1	1
`is_T()`	3	3	5
`wrap_T()`	2	2	4 (2)
`unwrap_T()`	1	2 (1)	3 (1)
address bits	27	29	29

Table 4: Costs for 2-stage Tagging Schemes

and the other three 2-bit codes are staging tags. In the first column, word pointers get shifted 5 bits for tagging, in the second column they only get shifted 3 bits. The first two columns assume that all 5 tag bits are in the low end of the word, the last column has a 2-bit tag field in the low end of the word and a 3-bit tag field in the high end. The operations `is_T`, `wrap_T`, and `unwrap_T` are for non-integer types. The row labeled "address bits" tells how many bits are available for representing word pointers. The numbers in parentheses are the costs for the three special word pointers that have code of `0b000` for their upper three bits.

2.6 Tagged Floats

Floating point numbers present several special problems to tagged-word schemes. First, the IEEE standard requires the use of all bits in a 32-bit word, so if any bits are discarded to make room for tags, then the implementation does not adhere to the standard. Second, in the IEEE format (and most others) the only bits that can reasonably be discarded are in the low end of the word. This is in contrast to all other values where the discardable bits are in the high end of the word. Third, it may seem that most programs do not use floating point arithmetic, but for those that do use it the performance of the floating point arithmetic is often critical to the performance of the program. This leaves the implementer with the choice of optimizing for the common case at the expense of making the compiler unacceptable for a large class of important programs, or of burdening the common-case programs with overheads intended to improve the performance of relatively rare cases.

Wrapped floats can be represented by tagged pointers to unwrapped floating point numbers as described in section 2.1.1. In this case wrapping floats requires an allocate and a store, and unwrapping floats requires a load, but it is possible to use full unabridged machine floats. In particular, this allows IEEE compliance on machines that comply.

If wrapped floats are represented by tagging the float number directly, then the implementation cannot claim compliance with the IEEE floating-point standards on 32-bit machines that comply. Furthermore, the costs for tagging and untagging floats are different than for anything else because the discardable bits are in the low end. This makes the decision of which end of the word to tag a bit more complex. The operations to tag and untag floats in the high end are

```
#define wrap_float(x) (x>>4) + (t_float<<28)
#define unwrap_float(x) x<<4
```

at costs of 3 cycles and 1 cycle [3]. This makes wrapping floats more expensive than wrapping other types and unwrapping floats cheaper.

If floats are tagged in the low end of the word, then the operation

```
#define wrap_float(x) (x & 0xFFFFFFF0) + t_float
```

will tag them at a cost of 2 cycles. Note that although 0xFFFFFFF0 looks like a large constant, it is really just -15 so it can be included as an immediate operand on a machine that sign-extends immediate operands to logical operations. Otherwise, something like

```
#define wrap_float(x) (x | 0b1111) - (0b1111 - t_float)
```

can be used.

Float values can be untagged by shifting out the lower bits:

```
#define unwrap_float(x) (x>>4) << 4
```

at a cost of 2 cycles.

As another complication, note that on many machines the floats must be tagged and untagged in an integer register (because bit operations are only allowed in such registers) but arithmetic must be done in a float register. This will involve extra register moves in some situations.

2.6.1 Using IEEE NaN Codes

The IEEE floating point standard defines a large set of encodings for NaN values, where "NaN" stands for "Not a Number". These encodings are those that have an exponent of all ones and a non-zero significand. The standard basically allows these codes to be used for anything, so they can be used to encode non-float values in a tagged-word scheme where all values except floats have `0xFF` in bits one through eight of a 32-bit word [19, page 96]. This restricts other data types to 24-bit representations, but it allows full IEEE floating point compliance.

For example, `0b011111111` can be used as a 9 bit tag in the high end for a non-zero object pointer (see section 7.1), and `0b111111111` can be used to tag integers. All other 9 bit tags represent IEEE floating pointer numbers. There is a problem with this representation in that the integer 0 also encodes the IEEE value of negative infinity and the object pointer that points to byte 0 is positive infinity. There are various ways to get around this problem: bit 9 could always be set for pointers and integers, thereby reducing the representation spaces of these types even further, or there could be special tests before and after each floating point operation that would recode the infinite results into some other representation. Either option is rather expensive.

3 Partitioned Words

As mentioned earlier, the essential requirement to implement wrapped values in a single word is to partition the set of possible bit patterns of a machine word among the various types, so that a type can be determined by examining the bit pattern. This can be done by dividing the words into a separate tag field and value field as discussed in section 2, or it can be done by allocating each type a certain subset of the available bit patterns. In other words, the available representations are divided up among the types in such a way that each type is restricted to representing those values that it can represent in the bit patterns allocated to it. This strategy is different from using a tag field because with the tag field technique, only the value field of a word is a legitimate value, while with partitioned words the

[3] In fact, this C code does not do what is wanted since conversion between floats and ints changes the representation. To do this in C, it would be necessary to declare a tagged value as a union.

entire word is a legitimate value in a format that the machine uses. This strategy, like tagging, makes it impossible to represent all one-word unwrapped values as direct wrapped values.

Generally, words are partitioned by the magnitude of the number they represent in the machine representation of (signed or unsigned) integers, and this special case will be referred to as "partitioning words by magnitude". For example a word of 32 bits can be used to represent the natural numbers from 0 to $2^{32} - 1$ and this set of numbers can be divided up into ranges based on magnitudes. The numbers

```
#define INT_START   0x0
#define ARRAY_START 0x40000000
#define CONS_START  0x80000000
#define ATOM_START  0xC0000000
```

can be used to define the ranges for four types of data: integers, arrays, cons cells, and atoms (symbolic constants). Arrays and cons cells are implemented as pointers to blocks on the heap. Array pointers are restricted to the range of addresses from `ARRAY_START` to `CONS_START` $- 1$, so all array blocks must be allocated between those addresses. Likewise, cons cell pointers are restricted to the range of addresses between `CONS_START` and `ATOM_START` $- 1$. Atoms may or may not be implemented as pointers, but however they are implemented, every atom must be represented by a bit pattern that represents a natural number larger than or equal to `ATOM_START`. Of course the acceptability of these addresses depends on the virtual memory system. If virtual memory is not large enough, or if the operating system requires virtual memory to be allocated contiguously, then this scheme becomes much less practical.

The representation above has no support for negative integers, but this problem can be remedied by making use of the two's complement representation of integers. In two's complement, a negative integer $-i$ is represented by subtracting the absolute value i from 2^b (where b is the number of bits in the word). This means that a negative integer with a small absolute value is represented as a positive integer with a very large value, so it is necessary to reserve the largest numbers for negative integers. For example

```
#define POS_INT_START 0x0
#define ARRAY_START   0x20000000
#define CONS_START    0x60000000
#define ATOM_START    0xA0000000
#define NEG_INT_START 0xE0000000
```

gives us a sort of "wrap-around" integer range that includes both positive and negative integers.

The beguiling characteristic of partitioning words by pattern is that there is no direct cost for wrapping and unwrapping data. Every object that is represented by a pointer is simply allocated in the right place so that its type can be identified by location alone, and tagged integers look exactly like untagged integers. However the integer representation in this scheme is the same as tagging integers by sign extension, and there are extra costs of that arithmetic as discussed in section 2.3. Also, there may be a hidden cost in the constraints that this scheme imposes on memory allocation: the requirement for multiple memory regions can translate into more overhead for storage management.

For example, for a simple allocation method where space is allocated merely by incrementing a heap pointer, allocation can be made faster by keeping the heap pointer in a register; but if there are multiple memory regions then multiple heap pointers are needed, and all registers used to hold the pointers become unavailable for other uses. If the implementer decides not to keep the heap pointers in registers because there are too many of them, then the extra cost of allocation must be partly ascribed to the tagging scheme. Similarly, if the allocation system uses a free list allocation instead of a heap pointer, and if the tagging scheme is part of the reason for using this slower allocation method, then the overhead of maintaining and allocating from free lists should be partly ascribed to the tagging strategy. The consequences of a particular representation can be far-reaching, and the costs can be hidden behind other features of an implementation.

The requirement to allocate structures in certain places may cause other problems depending on how much space is allocated and how the virtual memory system works. In particular, many operating systems do not allow allocation of arbitrary regions in the middle of virtual memory without also allocating all of the virtual memory between that section and one end of the virtual memory space. Also, if data is to be shared with programs written in other languages there may be restrictions on where the data must be allocated.

3.1 Testing Ranges

Another problem with partitioning words by magnitude is the large cost of testing the type of a wrapped value. For example, the C macros to test the types would look like this

```
#define is_int(x)   x >= NEG_INT_START || x < ARRAY_START
#define is_array(x) x >= ARRAY_START && x < CONS_START
#define is_cons(x)  x >= CONS_START && x < ATOM_START
#define is_atom(x)  x >= ATOM_START && x < NEG_INT_START
```

where the cost of a successful test is 6 cycles (each test involves 2 comparisons with 2 large immediates values and 2 branches. The delay slots are filled with instructions from the "success" branch). The integer test can succeed with just one branch, and it might be advisable to switch the order of the tests since non-negative integers are probably more common than negative integers.

Range tests can be optimized by using a trick based on the two's complement representation of integers and the fact that most machines have both signed and unsigned comparison. If `C` is a positive constant and `x` is a signed two's complement integer variable, then consider the effect of casting `x` to an unsigned quantity before comparison

```
(unsigned)(x) < C
```

This has no effect if `x` is greater than 0. But since negative signed integers test as very large unsigned integers, if `x` is less than 0 (meaning that it is also less than `C`) then this test will fail. In other words, `(unsigned)(x) < C` is equivalent to

```
x < C && x >= 0
```

This fact can be used to optimize range tests. For example, suppose the desired range test is

```
x < B && x >= A
```

where `x` is a variable and `A` and `B` are non-negative constants such that `A<B`. Subtract `A` from each side of each inequality to get

```
x-A < B-A && x-A >= 0
```

which is equivalent to

```
(unsigned)(x-A) < B-A
```

as shown above. Consequently, the previous range tests can be changed to

```
#define is_int(x)   (unsigned)(x-ARRAY_START) >= NEG_INT_START-ARRAY_START
#define is_array(x) (unsigned)(x-ARRAY_START) < CONS_START-ARRAY_START
#define is_cons(x)  (unsigned)(x-CONS_START) < ATOM_START-CONS_START
#define is_atom(x)  (unsigned)(x-ATOM_START) < NEG_INT_START-ATOM_START
```

at a cost of 5 cycles for each test.

The integer range test was formulated by first negating the original test to

```
#define is_int(x)   !(x < NEG_INT_START && x >= ARRAY_START)
```

transforming to

```
#define is_int(x)   !((unsigned)(x-ARRAY_START) < NEG_INT_START-ARRAY_START)
```

and then simplifying. If positive integers occur with a great enough frequency then it might be better to use the two-test method for integers, testing for positive integers first, since the best case on that test is just 2 or 3 cycles (depending on how the delay slot is filled).

3.1.1 Negative Integer Encoding

Testing for integers can be further optimized by assigning the range of all values that represent negative machine integers to negative wrapped integers, and keeping the range of all integers contiguous. In other words, every negative machine integer represents the same integer as a wrapped value, and every non-negative machine integer in a range from 0 to some n represents the same integer as a wrapped value. This allocates half of the available representations to negative integers, but the test for integers can be done with a single signed comparison against the maximum representable positive integer:

```
#define is_int(x) x <= MAX_POS_INT
```

The cost of this test is 3 cycles (assuming `MAX_POS_INT` is too large to represent as an immediate). Note that this same technique could be used to test integers that are tagged by sign extension.

3.2 Segmented Ranges

Let us call a range in the form 0bα000...0 to 0bα111...1 where α is any initial sequence of bits, a "segmented range" or a "segment". Segmented ranges have the useful property that membership in the range 0bα000...0 to 0bα111...1 can be determined by testing the upper $|\alpha|$ bits to see if they match α. If words are partitioned by magnitude into segments, then testing the type of a wrapped value is identical to testing the type of word tagged in the high bits.

In a segmented scheme, wrap-around integers become identical to integers tagged by sign extension, with the testing overheads involved in that, but the scheme outlined above in section 3.1.1 still works. For example

```
#define POS_INT_START 0
#define ARRAY_START   0x20000000
#define CONS_START    0x40000000
#define ATOM_START    0x60000000
#define NEG_INT_START 0x80000000
```

provides a set of constants that could use the tests

```
#define is_int(v)   ((int)v) < 0x20000000
#define is_array(v) (v>>29) == 0b001
#define is_cons(v)  (v>>29) == 0b010
#define is_atom(v)  (v>>29) == 0b011
```

This scheme costs 3 cycles for each test. Because of the extra cycle required for large constants, it would cost 5 cycles to mask out the lower bits and then compare with the upper ten bits.

3.2.1 Dynamically Typed Segments

Several lisp implementations [4] use a dynamic variant of segmented ranges by assigning type codes to ranges dynamically. This is done by setting aside a large range for all pointers, and then partitioning this range into many segments. Each segment is given a type code dynamically, and the types of values that are allocated in a particular segment is recorded in a table of segments called the BIBOP (for BIg Bag Of Pages). Given segments of 2^{10} bytes the macro to test the type is

```
#define is_T(v) *(type_table+(v>>10)) == t_T
```

Assuming that the address `type_table` is kept in a register (but keep in mind the costs of having one less register available for local computation), the cost of this operation is 6 cycles: one to shift `v`, one to add the address of the table, two to load the type, one compare, and one conditional branch. Testing is slower with this strategy than with statically partitioned segments, but it can be used even if virtual memory must be allocated contiguously.

Types that are represented directly in the word —instead of by a pointer— are sometimes given special tags so that only pointer types are looked up in the table. This is essentially a staged representation scheme, where the highest stage tag is an index for the BIBOP.

Instead of keeping the type information in a table, it can be kept in the segment itself. For example if the information is kept in the first word of the segment, type extraction would be implemented as

```
#define is_T(v) *(v&0xFFFFF000) == t_T
```

at a cost of 5 cycles. Note that the constant `0xFFFFF000` is actually `-4096` so it can be an immediate constant. This scheme is faster than the BIBOP method and does not need a reserved register, but it does reserve the first record in each block. It also requires a staged representation to represent non-pointer types, whereas this is optional with the BIBOP method.

4 Object Pointers

In the object pointer scheme, each wrapped value is simply a machine pointer to a block of memory, and the block contains all type information. This representation has been used in a relatively pure form in Smalltalk[5] [3, 6], Prolog [21], and functional languages [14].

Although the word "object" has a special meaning in some other contexts, it is used here to mean simply a block with a structure that contains enough information to identify what type of value the block represents (and how it represents that value). One simple way to accomplish this is to simply have the first word of every object be an integer type code. In this case, testing the type of an object pointer requires a reference to memory to load the type:

```
#define is_T(p) p->type_code == t_T
```

at a cost of 4 cycles.

Wrapping and unwrapping is a bit more complicated since the cost depends on whether the data type is already a pointer to a block or is represented directly in machine registers. Wrapping a pointer to a block is done by assigning a type code to the correct field in the block that the pointer references

```
#define wrap_T(p) p->type_code = t_T
```

[4] For example, MacLisp [15] and [5, pp 31–33]; Franz Lisp [5, pp 51–53]; Interlisp-VAX [19, page 29].

[5] Smalltalk implementations sometimes represent values with an extra level of indirection between the object pointer and the block. This representation is not motivated by the need to represent dynamic types, so it will be ignored here.

This requires 1 cycle to load the constant `t_T` into a register (assuming the machine does not have an instruction to store an immediate value), 1 cycle to fetch and execute the store instruction, and 1 cycle to store the value, for a total of 3 cycles.

Unwrapping is free, because the pointer itself is the unwrapped value:

```
#define unwrap_T(p) p
```

For non-pointer types such as integers, wrapping is done by creating an object on the heap, storing the value, and storing the tag:

```
#define wrap_T(x) (alloc_T, new->type_code = t_T, new->val = x, new)
```

at a cost of 5 cycles plus the cost of allocation. Allocation can cost as little as one cycle if hardware interrupts are used to detect allocation beyond the heap boundary and if the heap pointer is used to reference the newly allocated memory [9].

Unwrapping a non-pointer value requires fetching the value from memory:

```
#define unwrap_T(p) p->val
```

at a cost of 2 cycles.

4.1 Executable Type Descriptions

Instead of putting an integer code in the first word of an object, it is possible to place there a piece of executable code. Then determining the type of the object is done by jumping to the code. For example the executable code fragment for each type `T` might be a subroutine that sets a condition code if its first argument is `t_T`, the integer type code for `T`. Then type checking is done with

```
#define is_T(v) (*v)(t_T)
```

Wrapping and unwrapping is the same for this scheme as for the previous one. On a machine with the shortest possible calling sequences, this method requires at least two more cycles than the in-line test: one for the call and one for the return. Another problem is that it requires a separate piece of executable code to be allocated with each object. This can get expensive in space, so it is more practical to put in the object a pointer to the executable type description, and to execute it via an indirection at a cost of at least one more cycle. However, for programming languages where objects frequently have to be "executed" (such as object-oriented languages or languages with lazy evaluation), executable type descriptions help avoid some extra tests by providing a piece of code specific to the type.

There are many possible variations on the notion of executable type descriptions. For example, instead of the `is_T` operation as given above, the type dispatching might be combined with unwrapping. Another variation is to handle the conditional jump in the type description rather than just setting a condition code. These two notions can be combined with the additional indirection discussed above as follows:

```
#define check_T(v,lbl) (**v)(v,t_T,lbl)
```

If `v` has the type represented by `t_T`, then the value is unwrapped into some fixed register and control returns normally to the instruction after the call. If `v` does not have type `t_T` then there is a return to `lbl` instead of the instruction immediately following the call. The executable type description for an integer would look something like this:

```
int_code:
  compare r1,t_Int
  branch-eq return-lbl
  load r0, r0[4]
  jump r2
```

Here, `r0`, `r1`, and `r2` are the input argument registers and `r0` is the output argument register that contains the unwrapped integer after a successful test. The label of the instruction after the call is stored in the register `return-lbl` (presumably by the call instruction). The execution of this code takes 3 or 4 cycles, which must be added to the cost of the dereferencing and the call which is at least 3 cycles.

Another variation is to have multiple fragments of code associated with an executable type description. The multiple code fragments can be used for different purposes (for example, garbage collection). They can also be used in several different ways for dispatching on the type. For example, a value might be defined as an object pointer to a block `b` that contains several function pointers:

`b->test(code,lbl)` — return to `lbl` if `b` has type code `code`, otherwise return normally.

`b->eq(v,lbl)` — return to `lbl` if `b` is equal to `v`, otherwise return normally.

`b->jump(tbl)` — return to the address `tbl+code` where `code` is the type code of `b`. This is used for jump-table dispatching on `b`.

Of course this is an open-ended technique and it is possible to come up with any number of useful procedures. In fact in an "object oriented" language that fixes all possible message names at compile time, it is possible to do all operations on objects by jumping to addresses in the executable type description. However, these tables can get large, and even small ones add considerably to the size of small objects such as ints and floats. So to save space, the type codes are generally implemented with an extra level of indirection, so that all values of a given internal type references the same table of functions. Of course this further level of indirection reduces speed even further.

4.2 Hooked Values

Object pointer representations and executable type descriptions are convenient for representing hooked values. A hooked value is a value that causes special things to happen when it is accessed. For example, in Icon [8] there is a hooked variable[6] `&pos` that must have an integer value in a restricted range. An attempt to assign a value outside of this range to `&pos` causes a piece of code to be executed that does something special. Icon has other hooked values as well. For example an Icon substring expression `s[i:j]` creates a hooked value that evaluates to a normal substring if it is dereferenced, but this hooked value can be assigned to, in which case it executes a special piece of code that changes the value of `s`.

In languages with concurrent constraint evaluation (for example: [9, 4]), a procedure evaluation can suspend when it needs the value of an unbound variable and it must be woken up when the variable gets assigned a value. One way to implement this behavior is by assigning a hooked value to the unbound variable, with the behavior that when there is an assignment to the hooked value, the suspended process gets woken up.

Hooked values also arise in languages with lazy evaluation. Lazy evaluation of an expression means (more or less) that the expression is not evaluated until the value of the expression is actually needed. Lazy evaluation is implemented by creating hooked values for unevaluated expressions. The hooked value, when accessed, will evaluate the expression and then replace itself with the value of the expression.

The Spineless Tagless G-machine [14] is a virtual machine for implementing functional languages with lazy evaluation. It represents all wrapped values with object pointers (hence the name "tagless"), where the type of the object is encoded in a piece of executable code as described above. Whenever the value of an object is needed, the code component is executed, and this code fragment puts a type code and the unwrapped value someplace accessible. This makes for a very elegant model, but the cost of accessing non-hooked values is relatively high. We assume a very light procedure call that has an overhead of 2 cycles for call and return: the call instruction loads the return address in a register and jumps in one cycle —the delay slot is filled with an instruction from the jump target. The return instruction takes

[6] The Icon literature uses the term "trapped variable"

one cycle also and the delay slot for this instruction can almost always be filled with an instruction from the called procedure. We assume that executable type descriptions are shared among different values so the address being jumped to must be loaded from memory at a cost of 2 cycles. Finally, the value being tested needs to be moved to the first argument register at a cost of 1 cycle. The total overhead of doing a hookedness and type check is 5 cycles.

In a system that uses non-executable type codes, each point in the program that accesses a possibly-hooked value must insert special code to test for hooked values. Typically, the code to test for hookedness is immediately followed by code to verify the type. If hooked values are the "unusual" case then it may be best to test for expected types first and delay the test for a hooked value. Now we assume that the tag must be extracted regardless of the hookedness test, because other type tests need to be made. So the cost of doing a hookedness test first is just the cost of a compare and jump. If the value is not hooked then the overhead of the test is just 2 cycles (we assume that the delay slot can be filled with an instruction from the not-hooked continuation). If the value is hooked then the test costs 3 cycles (because the instruction in the delay slot is wasted), and the call to handle the hooked value is 5 cycles, so the total overhead in case the value is hooked is $3 + 5 = 8$ cycles.

When the hookedness test is done second, there is no overhead if the value is not hooked. If the value is hooked, there must be a call to handle it. After return from the handling code the type test must be made again. For example, if `v` is the value being tested then the code might look like this

```
lbl:
  t = extract_type(v)
  if (t==int_t) {/* execute normal case */ }
  else if (t==hooked_t) {
    v = handle_hook(value);
    goto lbl;
  }
```

The overhead if `v` is hooked involves not only the hookedness test (3 cycles) and the call (5 cycles), but also another tag extraction (1 cycle, say) and another type test (2 cycles). So the total overhead for hooked values is 11 cycles.

Table 5 presents a summary of the relative costs under the assumptions given above. The table shows

Technique	Not Hooked	Hooked
executable type code	5	5
test hookedness first	2	8
test type first	0	11

Table 5: Costs of Hooked Values

that the choice of which is the best scheme depends on what percentage of values are found to be hooked at run time. These figures are partially confirmed by the experiments reported in [10] where there is also a discussion of re-introducing tags to speed up the case where the value is not hooked.

Note that the techniques of executable type descriptions are not restricted to object pointer schemes. Such type descriptions could be used quite easily with large wrappers, and any representation with type codes can use the type codes as indexes into a table of addresses of executable type descriptions. However it is often the case that a hooked value wants to modify itself (usually into a non-hooked value) when it is evaluated and object pointers are better for this than are other techniques that have direct representations of some values. The reason is that if the representation is direct, then there may be no good way to modify all of the instances of a value since there may be copies of it. With object pointers, all "copies" of an object are pointers to the same block.

4.3 Preboxing

One of the more serious problems with object pointers is that they require all values to be heap allocated. In order to reduce the overhead of heap allocating small objects like integers, some Lisp systems use a technique called preboxing. This involves preallocating some of the values, say those integers from -1000 to 1000 and then using these preboxed values instead of wrapping new ones. For example each arithmetic operation might include a test to see if the result is in the range -1000 to 1000 and if so, it produces the preboxed value rather than wrapping a new integer. A modification of this scheme is to use special addresses to represent these values without actually ever allocating them. Of course these schemes are not limited to object pointer representations —it is generally applicable whenever integers are represented indirectly.

5 Large Wrappers

It is not necessary to restrict data representations to a single word as the above schemes do. An alternative is to simply use wrappers that are large enough to represent the type information and a complete machine value as well. Keeping a large wrapped value in registers requires more registers than schemes that use a single word, but with modern machines this is less of a problem than it once was. Similarly, large wrappers require more memory than single word wrappers, but memory sizes today are very large and getting rapidly larger. The most serious problem with these representations is that loading and storing wrapped values takes longer than for single-word representations, but there are several techniques for reducing these costs, and there are some time optimizations available with large wrappers that are not available with single-word wrappers. Consequently, multi-word representations deserve careful consideration.

If wrappers are restricted to representing word-sized values (which means using pointers for larger values) and if the number of types is restricted to the number of word representations, then wrapped values can be represented in two words: one for the type encoding and one for the value encoding. The implementations of Icon described in [8] and [22] both represent dynamically-typed data in such pairs (with some extra information in the type word). In the following, such two-word representations are called "double wrappers".

5.1 Double Wrappers

Since double wrappers have the type code and the value code in different registers, the operations of unwrapping and extracting the tag are essentially free (but recall that there is an extra cost in loading the two registers in the first place). In the following, let `v` represent a pair of general purpose registers, and let `v.type` refer to one of the pair and `v.val` refer to the other. Then the operations on double wrappers might be described in C by

```
#define is_T(v)      v.type == t_T
#define wrap_T(v.val) (v.type = t_T, v)
#define unwrap_T(v)   v.val
```

This notation is a little convoluted because it attempts to maintain a uniform framework with the other representation techniques. The definition of `is_T` says that to test the type code of a pair of registers `v`, representing a double wrapper, simply compare against the register that contains the type code. If `t_T` is a small immediate, then this costs 2 cycles. The definition of `wrap_T` says that to construct a double wrapper for a value in register `v.val`, simply load the type code into `v.type` and the double wrapper is then the register pair represented by `v`. Note that if `t_T` is shifted to the high end of the word (for reasons discussed later) this operation costs no extra for the load-high, because then the load-high replaces a load-immediate. The definition of `unwrap_T` says that the machine representation of the value in the two-register pair `v` is in register `v.val`. This notation assumes some uniform arrangement of registers into value registers and their corresponding type registers.

5.2 Qualifiers

Sequences with statically unknown lengths can be represented in several ways. Assuming that the sequence is stored in a contiguous array, the minimal requirement for the representation is that it is possible to easily determine the beginning and end of the contiguous array. The beginning of the array is generally represented by a pointer. The end can be represented by another pointer, by a length, or by having a special end-of-sequence object at the end of every sequence (such as the `'\0'` character at the end of C strings). The end-of-sequence marker is not a good implementation since it requires a special value that cannot appear in sequences and it requires traversing the entire sequence to find the length of the sequence. The choice of whether to mark the end with a pointer or a length depends on how the sequence is used. The pointer representation is probably a bit better as a general rule, but it is more common to use a length, so we will assume this representation.

Given that the sequence is represented with a pointer and a length, it is necessary to decide how the length is to be encoded. There are two common approaches. One approach is to put the length just before the contiguous array of data. Then the pointer to the sequence is all that is needed to encode the entire sequence. Another approach is to allocate a "qualifier", a two-word block that contains a length and a pointer to the contiguous array of data. The qualifier representation is more versatile since it allows the same contiguous array of data to be shared by several different sequence representations. For example, if strings are represented with a qualifier, then it is possible to create a substring of the string `s` just by creating a new qualifier that references a subrange of the array of characters used to represent `s`; it is not necessary to copy any characters. Of course this assumes that sequences are immutable or that it is acceptable for side-effects on one sequence to change the elements of another sequence. The disadvantage of the qualifier representation is that it may require an extra allocation to create a sequence that is represented with a qualifier, and accessing the elements of the sequence requires an extra level of indirection.

However, since a qualifier only requires two words of storage, it can be represented within a single double wrapper (after removing a few bits for tagging). In this case, there is no extra overhead for the qualifier representation beyond the overhead of accessing double wrappers. A qualifier is implemented in a double wrapper by keeping either the length or the pointer in the value word of the double wrapper and keeping the other in the type word. Of course the set of type codes must be distinguishable from any length or pointer kept in the type field (to allow dynamic type checking). In many ways, the problem of how to encode a length or pointer in the type word of a double wrapper is very similar to the problem of how to tag words, and many of the same considerations apply. For example, one might limit the upper magnitude of representable pointers or lengths in the type field of double wrappers, and choose type codes larger than the largest magnitude. This is analogous to the technique of partitioning words by magnitude.

The problem with this encoding of types is that the type codes are all immediate values too large to fit in a half-word, and therefore they must be constructed at run-time. It was said previously that this is not a problem in the double wrapper representation, because the type codes do not have to be used as bit-masks, they are simply loaded directly into the type word and it makes no difference whether they are loaded with a load-immediate or a load-high. However, if some type of double wrapper requires the type word to contain other data (like a sequence length) along with the type code, then this type of double wrapper requires that the type code be used as a bit-mask. This strategy is about the same as using a tag field in the high end of the word.

Alternatively, the type word can be divided up into a tag field and length/pointer field, and the tag field can be in either the high end of the word or the low end. As might be expected, there are advantages and disadvantages to either method. The following examples will discuss some of the options, assuming a 4-bit tag field and a 28-bit field for qualifier lengths in the type word. If the type code is put in the low end of the type word and the length is put in the high end, then all non-qualifier types can be operated on without masking:

```
#define is_T(v)      v.type == t_T
#define wrap_T(v.val) (v.type = t_T, v)
#define unwrap_T(v)   v.val
```

The costs are: 2 cycles for `is_T`, 1 cycle for `wrap_T`, and 0 cycles for `unwrap_T`.

Qualifier operations must be masked in general:

```
#define is_T(v)    (v.type & 0b1111) == t_T
#define wrap_T(v)   (v.type = (v.type<<4) & t_T, v)
#define unwrap_T(v) (v.type = v.type>>4, v)
```

Here it is assumed that unwrapped qualifiers are kept in the same register pairs as double wrappers, with the length field of the qualifier replacing the type word of the double wrapper and the pointer of the qualifier replacing the value word. The operation `is_T` operates on the double wrapper representation and costs 3 cycles; `wrap_T` begins with a qualifier representation and converts to a double wrapper representation at a cost of 2 cycles; and `unwrap_T` begins with the double wrapper representation and converts to the qualifier representation at a cost of 1 cycle.

If the tag field is kept in the high end of the word, then both qualifiers and non-qualifiers must shift the type word (or construct large constants at run-time). The implementation for non-qualifiers is

```
#define is_T(v)    (v.type>>28) == t_T
#define wrap_T(v)   (v.type = t_T<<28, v)
#define unwrap_T(v) v.val
```

The costs are: 3 cycles for `is_T`, 1 cycle for `wrap_T` (recall that `t_T<<28` can be constructed and loaded at the same time with a single load-high instruction), and 0 cycles for `unwrap_T`.

Qualifiers have the same `is_T` operation but the other two operations have to take care of the length representation:

```
#define wrap_T(v)   (v.type = v.type & (t_T<<28), v)
#define unwrap_T(v) (v.type = (v.type<<4)>>4, v)
```

The costs are: 3 cycles for `wrap_T` and 2 cycles for `unwrap_T`. If one type of qualifier is given the tag `0b0000`, then this type of qualifier gets free wrapping and unwrapping. Likewise, if one type of qualifier is given the tag `0b1111`, then this type can be wrapped and unwrapped just by negating the type word at a cost of 1 cycle each.

If there is only one type of qualifier, then the sign bit can be used to distinguish this value from all of the other types. If negative words are used for type codes the operations for non-qualifiers are

```
#define is_T(v)    v.type == -t_T
#define wrap_T(v)   (v.type = -t_T, v)
#define unwrap_T(v) v.val
```

The expression `-t_T` is a compile-time constant small enough to use for an immediate operand (assuming sign extension of immediate operands) so the costs are: 2 cycles for `is_T`, 1 cycle for `wrap_T` and 0 cycles for `unwrap_T`.

For the single qualifier type the double wrapper representation is the same as the qualifier representation, so the operations are

```
#define is_T(v)    v.type <= 0
#define wrap_T(v)   v
#define unwrap_T(v) v
```

and the costs are: 1 cycle for `is_T` (assuming a branch-if-le-0 operation), and 0 cycles for the other two.

5.3 Double Wrapper Optimizations

There are several ways to reduce the extra costs of loading and storing double wrappers, beginning with the technique of passing parameters in registers instead of on the stack (which is a good idea anyway). This avoids the need for pushing double wrappers on to a stack and popping them off for every call and eliminates a large number of memory references. The two major remaining sources of memory traffic are saving values on the stack between procedure calls and accessing double wrappers in aggregate objects like lists. These memory accesses can be reduced considerably by using local unwrapped values, passing unwrapped values as parameters, and storing unwrapped values in aggregates (sections 7.4 and 9.1). Since double wrappers can represent qualifiers so efficiently, it may be worthwhile for a language that has arrays and represents values with double wrappers to have special representations, where the contiguous array of data contains unwrapped values.

It is also possible to avoid storing one word of a double wrapper when it is known that the word is already correct. For example, an Icon expression `i +:= 1` to destructively update the number `i` will produce a value that has the same type as `i` (in the absence of overflow). In this case it is only necessary to store the new value of `i` in the value word of the double wrapper, the type word does not need to be updated. Similarly, the Icon expression `s := s[1:-1]` to remove the last character from the string `s` might be implemented by subtracting one from the length field of the double wrapper/string-qualifier without touching the pointer field.

Double wrappers can be used to optimize the representation of 32-bit IEEE floating point numbers, which must be allocated on the heap with other methods. If data is being represented with double wrappers, then a float can be represented directly in the value word of the double wrapper.

Double wrappers can also be used to directly represent types of data that might otherwise be represented with two-word heap blocks. For example, there could be a special type code for directly represented cons cells. The type word of the double wrapper could represent a pointer to the car as well as the type, and the value word would be a pointer to the cdr. This encoding might save space, but it would not save any cycles by shortening reference chains because the car and cdr of the cons cell are no longer represented directly in the cons cell. For example, if the cons cell were represented as a pointer `p` to a pair of elements on the heap, then it would take one memory access to reference the cdr at `*(p+1)`. With the encoding suggested above, the cons cell is directly represented in the pair of registers `v`, but the cdr is represented as a pointer in register `v.val` so it take one memory access to reference the cdr at `*v.val`.

Since there are a very large number of type codes available, user-defined types can be represented in a double wrapper with a type code and a pointer rather than using the object-pointer representation. This makes type checking more efficient for these objects. In Prolog, terms could be represented similarly with a functor in one word and a pointer to the args in another word, this could save a few cycles in unifying terms, since the functors would already be in registers.

6 Typed Location

There are several ways to type values based on where they are stored. Note that this refers to where the data element is stored, not to where it points. For example if the data element is a pointer and the type is determined by the region of memory which the pointer references, then the type is encoded in the representation of the pointer, not in the location of the pointer. This is an example of a partitioned word representation, not of a typed location. With a typed location the type of the pointer is known just because the pointer was found in a certain place on the stack, or in a certain register, or something similar; it does not matter where the pointer points.

In a pure statically typed language the type of any value is known because of its stack or register location at a given point in the program. In a sense, it is the locations that have types rather than the values. But there are more dynamic versions of typed locations where types are encoded by a type code, but the value and the type code are kept in separate places. For example a stack frame may contain a sequence of bytes containing type codes followed by a sequence of local variables, and the type of value in local variable i is given by the type code in byte i. Type codes that are kept separately from the values they refer to are "segregated type codes". Segregated type codes have costs that are mostly identical to the costs of double wrappers, the only differences are in how the type is located once the data element is known and in the fact that segregated type codes can be smaller than a word. If type and data are kept together, then the sum of their sizes should add up to a natural size for the machine, but with segregated type codes this is not necessary.

Typed locations are common in languages with statically typed values that need dynamic type information for purposes such as storage management and debugging. In such languages it is often not necessary to keep the type information as separate codes at all. Instead, the type can be "hardwired" into special functions for doing storage management and debugging. In other words, each user-defined function has a fixed pattern of static types in its stack frame. For each such pattern of static types it is possible to compile a specialized function for garbage collecting or debugging that stack frame, and this function does not need to deal with type codes since it knows what type is at each stack slot by static criteria [7].

7 Hybrid Techniques

Representation schemes can be combined in various ways to trade off the various advantages and disadvantages. Sometimes it is possible to merge two schemes so that they share representations. For example an implementation might use tagged words to represent everything except integers, and use partitioning by magnitude for integers. This is essentially what tagging integers by sign extension amounts to.

More commonly, two techniques are combined by nesting one representation inside another one. In other words, there is one specific representation scheme that is used for all wrapped values. But some of the "types" of these wrapped values are in fact dynamically typed values using another representation (which may or may not use the same technique as the first representation). This sort of scheme is called a staged representation, which should be distinguished from a staged tagging scheme. A staged tagging scheme is a particular form of staged representation where all stages use tags. There are certain optimizations possible for staged tags (especially involving dispatching) that do not apply to staged representations in general.

7.1 Tagged Object Pointers

Object pointers and tagged words are often combined with each other in a staged representation scheme. Pure object-pointer schemes are rare because there are some sorts of values (such as integers) that are small enough to be represented in one word, and so common that it might be impractical to allocate them all on the heap. Pure tagged word schemes are rare because tag fields do not provide a large enough range of type codes for all of the types that are needed —especially in languages that provide for user-defined types.

Objects are typically word-aligned and on most machines the lower two bits of an object pointer are available for "free" tagging. This suggests a staged representation scheme where the first stage is a tagging scheme using the lower two bits of the word, and the second stage is a general object-pointer scheme. Operations on the three first-stage types are as described in section 2. The operations on object pointers are:

```
#define is_object_T(v) (v & 3) == t_object && *(v-t_object) == t_T
#define wrap_T(v)       (*v = t_T, v+t_object)
#define unwrap_T(v)     v-t_object
```

The number `t_object` represents the first-stage staging code (it is assumed that addition and subtraction treat `v` as an integer instead of scaling `t_object`, otherwise the necessary casts would make the expressions very complicated). The cost to test the type of a value is 7 cycles if `v` is any type represented by a tagged object, 3 cycles if it is represented by a direct tag. It is presumed that one load-indirect instruction is generated for the expression `*(v-t_object)`, so this expression is only charged two cycles. By choosing the tag appropriately, the cost of testing that the wrapped value is an object pointer can be reduced one cycle (see section 2.1) to get a total cost of 6 cycles.

Wrapping an object requires storing the type code in the object (3 cycles, see section 4) and adding the tag to the pointer (1 cycle) for a total of 4 cycles. Unwrapping an object only requires subtracting the staging tag for a cost of 1 cycle (except for values with machine representations that are not represented directly). Unwrapping pointers is unlikely to be a common operation since it is possible to access the fields of the object by offsetting from the tagged pointer as efficiently as offsetting from the untagged pointer.

7.2 Tagged Words and Partitioning by Magnitude

If a set of words is partitioned by magnitude into segments that are distinguished by the upper n bits of the word, then these n bits can also be viewed as a tag field. In this case, some type of values can be represented by an n-bit tag in the upper part of the word while others are represented by a word that have a fixed n-bit segment. Recall that the difference between tagging and partitioning is that for tagged words the value must be extracted from the value field while for partitioned words the whole word is a legal value. Clearly it is possible to mix up these these two strategies, using a value field to represent some types of unwrapped values and using the whole word to represent other types.

7.3 Tagged Words and Double Wrappers

To some extent, the representations described in sections 5.2 and 5.3 are hybrids between tagged words and double wrappers, but since those representations are a part of the benefit of using double wrappers in the first place, they are treated as part of the double wrapper representation rather than a hybrid scheme. However, it is possible to have a true hybrid scheme if the implementer is willing to give up the property that all wrapped values have the same size. This allows a representation where some values are tagged words and others are double wrappers. For example integers and a couple of pointer types could be represented with two bits in the low end of the word, and the fourth tag value could be used as a staging tag that informs the runtime system that there is more to the value. The remainder of the value would be in different places in different situations.

For example, if parameters are passed by register, then if argument register i is tagged with the staging tag this would indicate that register i represents the type word of a double wrapper and that the value word is in another register, say $i + x$ where x is the maximum number of argument registers. On machines with special floating-point registers there might be a special staged tag that indicates the value is a float in a float register. If a wrapped value is loaded from the stack, and it has the staging tag, then this would indicate that the value word is the next word on the stack. Of course the problem with this is that it is not possible to save the registers on the stack without examining each one to see if it requires saving another register. This might be very expensive for programs that use a lot of procedure calls.

This representation causes problems in representing aggregate objects also. For example if a word in an array on the heap is tagged with the staging tag, then this indicates that there is another word somewhere that contains the value. It is difficult to implement such aggregates efficiently without using double the space, and in this case there is not much benefit of this method over a pure double wrapper strategy.

7.4 Hybrids With Typed Locations

It is often the case that aggregate structures have elements of all the same type of object. In such cases it is often possible to avoid including dynamic type information with the elements themselves, since their type is known by the fact that they occur in a particular place. This is particularly useful in tagging representations —where it allows one to avoid the costs of tagging and untagging elements of the aggregate— and in double wrapper representations —where it allows one to save space and to avoid the cost of storing type a type code for each element of the aggregate.

8 Dispatching on Dynamic Type

In the following, a "type" is a variable, the dynamic type of a variable value. A "type code" is a constant, a number or bit-pattern used to represent a type at runtime. To dispatch on a type means to select a piece of code to execute based on the type code that represents the type. As mentioned earlier, many type tests are of the "is or is not" variety, which can be done efficiently by the `is_T` macros as presented throughout this document. However a sequence of `is_T` macros is seldom an efficient way to implement a more general dispatch. The best way to dispatch on a type is to extract the type to a register and then do one of the following:

- A sequence of equality comparisons on the register.
- A binary search on the value in the register.
- Indexing into a jump table and then jumping to the correct piece of code.
- A combination of the above.

When doing a sequential or binary search, let d be the number of tests done that compare the type to a type code (this includes all failed tests as well as the successful test). The value d depends on the order in which the tests are generated as well as the statistical properties of the value `v` being tested. For example, if at one point in the program, `v` is dynamically seen to be a wrapped integer $n\%$ of the time then by testing for `t_int` first, the cost of the dispatch will be the cost of a single test $n\%$ of the time.

8.1 Sequential Search

For a sequential search the total cost is the cost of extracting the type plus the cost of d comparisons on the type. For a representation that uses two-bit tags in the low end, the dispatch would look something like this:

```
t = v&3
if (t==0)       {/* type 0 case */ ...}
else if (t==1)  {/* type 1 case */ ...}
else if (t==2)  {/* type 2 case */ ...}
else            {/* type 3 case */ ...}
```

The delay slot of each test except the last can be filled with a compare instruction for the next test, so as long as a jump is to the next test it only costs 2 cycles. However for the jump that exits the chain the compare instruction in the delay slot will not be useful, so that jump costs 3 cycles (unless it is the last jump in the static sequence, but we will ignore this one). So the cost is 2 cycles per jump plus 1 extra cycle for the final jump, giving a cost of $2d + 1$. Let E be the cost of extracting the type from the wrapped value; then the cost of the dispatch is $E + 2d + 1$. The maximum value of d is one less than the total number of type codes.

8.2 Staged Tags

For most staged representations, a dispatch on the type will contain branches to second-stage dispatching code for the second-stage types. However, staged representation schemes that use bit patterns in the word for the types can make use of special optimizations. In the following we will assumed staged tags, but many of these comments apply to types partitioned by magnitude as well.

For a 2-stage tagging scheme the cost of extracting the type is larger than for a 1-stage scheme since a test is needed to find whether the value has a first or second stage tag. But the test that tells the stage also reduces the search space for the other tests, much like a binary search comparison would do. In other words, if the type code is a second-stage tag, then there is no need to test against first-stage tags. Let E_i be the cost of extracting a tag at stage i. Then the cost of a sequential search for 2-stage tags is $E_1 + 2d + 3$ for first-stage types and $E_1 + 2 + E_2 + 2d$ for second-stage types. In either case, the worst case value of d is one less than the number of non-staging tags at one level. In particular, if there is only one non-staging tag at the first level, then $d = 0$ for that stage. The above assumes that either there is only one staging code at the first level or only one non-staging code so that testing if a code is a first-stage code only takes 2 cycles. If there are multiple staging and multiple non-staging codes, then they can usually be arranged in such a way that stages can be identified with a 2-cycle test.

If the most likely tags are all in the first stage, then it might be more efficient to first extract the type code for that stage and compare against the most likely types before testing for the stage of the tag.

8.3 Binary Search

If the set of type codes is representable as a sequence of machine integers, then dispatching can be done by a binary search. For example if wrapped values are words tagged by the lower four bits, a dispatch might be done like this:

```
t = v&0b1111
if (t <= 7)
  if (t <= 3)
    if (t <= 1)
      if (t == 0)  {/* type 0 case */ ...}
      else         {/* type 1 case */ ...}
    else
      if (t == 2)  {/* type 2 case */ ...}
      else         {/* type 3 case */ ...}
...
```

Delay slots can be filled here as for the sequential search, but it cannot be predicted how many jumps will make use of the compare in the delay slot and how many will ignore it, so the worst case cost is $E + 3d - 1$ and the best case is $E + 2d$; but d is logarithmic in the number of type codes rather than linear. The type code is selected by ruling out all other type codes (that is, one settles on type code `t_T` because it has been determined that the type is either greater or less than all other type codes). At each branch, half of the type codes are ruled out, so that the number of tests d is always either $\lceil \log_2 n \rceil$ or

$\lfloor \log_2 n \rfloor$, where n is the number of type codes. In the following, we will assume that n is a power of two so that floors and ceilings can be ignored.

To reduce the expected value of d (possibly based on frequency distributions of tags) it is possible to insert equality tests among the inequality tests. For example, the search above might be changed to

```
if (t == 7)          {/* type 7 case */ ...}
else if (t <= 7)
  if (t == 3)        {/* type 3 case */ ...}
  else if (t <= 3)
    if (t == 1)      {/* type 1 case */ ...}
    else if (t <= 1) {/* type 0 case */ ...}
    else             {/* type 2 case */ ...}
...
```

In the worst case, each three-way test only cuts the search space in half except at the leaves where one less test is needed, giving $(\log_2 n) - 1$ three-way tests. Each three-way test requires two branches so in the worst case, $d = (2\log_2 n) - 2$; which is worse than the previous strategy, but the best case is $d = 1$. Each branch is either 2 or 3 cycles, giving a cost of $E + 2d$ to $E + 3d - 1$.

Another variation is to do the equality test second rather than first. For example the fragment above might be changed to

```
if (t <= 7)
else if (t == 7)     {/* type 7 case */ ...}
  if (t <= 3)
  else if (t == 3)   {/* type 3 case */ ...}
    if (t <= 1)
    else if (t == 1) {/* type 1 case */ ...}
    else             {/* type 0 case */ ...}
  else               {/* type 2 case */ ...}
...
```

Here, the worst case for d is the same as previously, but only two type codes require the worst-case number of tests (in this case they are the type codes `1` and `0`). If `t` ≤ 7 then the three-way test on 7 requires one compare and two branches, otherwise it just requires one compare and one branch. So in one direction there is a test that costs one cycle and in the other direction there is not. Binary searches can also be done with a mixed strategy, using two-way branches at some points and three-way branches at other points, mixing the two forms of three-way branches as well. The optimal strategy depends on the dynamic distribution of type codes.

8.4 Jump Table

A jump table is an array of addresses. It can be used for dispatching on the value of an integer i by storing at each position c the address of the code that should be executed if $i = c$. Dispatching on i is done by jumping to the i^{th} address of the the jump table. In general it is not safe to index an array without first checking the range of the index variable, and this holds true for jump tables as well, so most jump table dispatches require a range check before indexing the table. However if the index is known to be a type code, and type codes have a small fixed range, then this check can be avoided. If a range check is needed it can usually be done in no more than 3 cycles using the range testing method described in section 3.1 (but here the constants are both assumed to be small enough to be immediates, so 2 cycles is subtracted from the range test).

The code for dispatching on the value of a tag variable `t` is something like this

```
t <<= 4                  /* scale to word size */
addr = jump_table        /* get address of jump table (2 cycles for large immediate) */
addr = addr[t]           /* load the code address (2 cycles) */
goto addr;               /* jump (2 cycles, delay slot is not filled) */
```

for a total cost of $E + 7$ cycles (where E is the cost to extract the type code).

For staged tags it is not necessary to determine the stage of the tag to extract it. Instead, it is possible to extract just the largest-size tag and for smaller tags, to fill several slots in the jump table. In an $n + m$ bit staged tagging scheme, each n-bit type code `t_T` gets m slots in the jump table, one for each $(n + m)$-bit number that has the lower n bits equal to `t_T`. For example consider a 2+1 staged tag scheme in the low end of the word with the tags

```
#define INT  0b00
#define FLT  0b01
#define CONS 0b10
#define ARR  0b011
#define NUM  0b111
```

For such a scheme, a jump table would be implemented something like this:

```
switch (v & 0b111) {
case 0b000: goto int_case;
case 0b001: goto flt_case;
case 0b010: goto cons_case;
case 0b011: goto arr_case;
case 0b100: goto int_case;
case 0b101: goto flt_case;
case 0b110: goto cons_case;
case 0b111: goto num_case;
}
```

Note that a similar table of type codes could be written to extract the tag as a value. In other words, it would be possible to construct an array indexed by $(n + m)$-bit numbers that has for each element the corresponding $(n + m)$-bit tag.

An implementation ought to pick a good method for dispatching on the type of a variable [1], given the expected value of d, and ought to order the tests in such a way as to minimize d. If this is done, then the overhead of using staged representations is often small enough that the staging is useful.

When writing a test in C, the switch statement should not be used if speed is very important. Although a good compiler will generate dispatch code intelligently, the C compiler cannot know about type code frequencies, and this can seriously affect the best method. Also, few C compilers are smart enough to notice that the range test is not needed.

8.5 Testing with Traps

Some percentage of type tests are merely to verify the applicability of a certain operation. For example, before taking the car of a tagged cons cell, it is necessary to confirm that the tagged value is, in fact, a cons cell. In some cases it is possible to avoid a specific test by arranging that the operation will cause a machine interrupt if the value is not as expected. For example, some machines will trap if a word access is done with an address that is not word-aligned. On such a machine, one pointer type can be given the tag `0b00` and this pointer type can be dereferenced without checking its type, as long as the trap is caught and handled correctly when the type is not correct. Of course this method can be generalized for type-checking situations where it is not an error for the type to be other than as expected, but the costs of interrupts on modern machines are so expensive [13], that this technique is unlikely to be cost effective in other cases.

9 Miscellaneous Considerations

9.1 Avoiding Operations on Dynamic Representations

There are a number of ways to avoid dynamic type operations altogether, and it is probably these techniques that will eventually make dynamically typed languages as efficient as statically typed languages. For example, within a procedure, it is often possible to infer the types that values will have. If so, then these values can be manipulated in the unwrapped state, there is no need to wrap them. This applies not only to numbers (as in several implementations of Lisp, Scheme, and Prolog), but to addresses as well. For example, a tagged sequence type that is represented by a contiguous array on the heap can be manipulated by getting a pointer to the array on the heap and using that directly [9].

There is an optimization —or rather a class of optimizations— called "call forwarding" [2] that skips unnecessary instructions at the beginning of a call, including conditional branches (and it can be generalized to skip instructions at other points as well). This optimization is particularly effective at skipping type checks in a program that uses types in a consistent manner. Also, if a single value is used multiple times in different branches of a procedure, then (in some languages) it is possible to test the type just once, at the beginning of the procedure, and unwrap the value immediately. Then call forwarding can often skip the test and sometimes even the unwrapping. The resulting code is very close to what would be produced in a statically typed language.

If global type inference is done, then it is sometimes possible to pass unwrapped parameters and return unwrapped values. This is especially important in functional and logic languages that rely heavily on recursion (since these languages use recursion instead of loops, local unwrapped values will have a limited usefulness). Of course the presence of unwrapped values in a dynamically typed language can cause difficulties for garbage collection.

9.2 Language Design

There are some language design choices that can seriously impact the effectiveness of type inference, and thereby the effectiveness of tag optimizations. For example, separate compilation makes global type inference much less effective unless type inference is delayed to the linking phase or the language has typed import declarations. Doing type inference at link-time tends to defeat the main purpose of separate compilation —to speed up compilation time by avoiding the need to re-compile modules that have not changed— because global type inference is typically responsible for a substantial fraction of compilation time. If typed import declarations are required for external procedures, then this tends to defeat the purpose of having a dynamically typed language. But if these declarations are optional then they can make global optimizations work much better for programs that use them. Of course it may still be necessary to do some link-time or run-time checking to ensure that the declarations are correct.

If the type system is set up such that there is arbitrary conversion between integers and floats, then type inference cannot do as well at inferring the types of numbers, and unwrapped arithmetic cannot be used as often. On the other hand, if the language provides explicit conversions from floats to ints and requires integer operands in places where integers are needed (as an array index, for example), then this provides more information for type inference. For example if the variable `I` is used as an array index, and `I` is constructed from the expression `I=K*N`, then this gives the information that not only `I`, but `K` and `N` are going to be required to be integers. This allows the tests to be moved early in the function (and then frequently skipped by call forwarding). The only burden this places on the programmer is that in the (unusual) case where it is really intended that an array index be formed from a floating point number, one of the explicit float-to-integer conversions must be used.

Arbitrary-precision arithmetic is a performance problem since it requires either that all arithmetic be done in the less efficient arbitrary-precision format or that there be (at least) two different types of integers (the same comments apply to arbitrary arithmetic on other sorts of numbers). Type inference is not likely to be able to infer that a value is a "small" integer as opposed to just an integer, so this language feature makes it practically impossible to use unwrapped numbers without actually producing two different copies of the code —one for unwrapped numbers, and one for arbitrary precision numbers.

9.3 Machine Considerations

With an instruction set that sets condition codes on logical operations, it is sometimes possible to save a cycle in the common sequence where a tag is first tested then extracted. For example, suppose cons cells are tagged with the tag `0b0110` in the low end of the word. The normal test and untag sequence needed to get the pointer would take 4 cycles. But this can be optimized with a sequence like

```
#define test_get_cons(v,lbl)  (w = v-0b0110, (v & 0b1111) != 0 ? goto lbl : w)
```

This sequence costs just 3 cycles because the test `(v & 0b1111) != 0` can be done in two cycles.

The MIPS machine has single instructions for equality and disequality tests [12], and this changes some of the costs, possibly changing the preferred strategies in some cases.

The SPARC has some special arithmetic instructions that are intended to work with 2-bit integer tags of `0b00` in the low end of the word [12, 13]. There are instructions `taddcc` and `tsubcc` to add or subtract such integers and set the overflow flag if either operand did not have the lower two bits equal to `0b00`. The related instructions `taddcctv` and `tsubcctv` are similar but also cause a trap if the lower two bits are not `0b00`. This is intended to support tagged-integer arithmetic, and may be useful if (1) it is practical to use two-bit tags, (2) it is practical to use `0b00` for integers, and (3) the integer case is common enough to overcome the extra cost of handling the other cases. The non-trapping instructions can be used to test operands after the operation (section 2.3.3) at a cost of only one cycle (and no extra tag bits). The trapping instructions can be used to test the types of arithmetic operands with traps (section 8.5). There are several special-purpose architectures with support for tagging, but such machines are outside the scope of this paper.

If a machine has "circular" shift instructions that shift bits out one end of the word and into the other end (instead of just shifting in `0b0` or the sign bit), it is possible to have tags bits in both the high and low end of the word, and to shift them all to the low end for masking in two cycles. This would save one cycle from the cost given in section 2.5 for testing distributed tags.

The M88000 has support for manipulating bit fields. These may have some impact on the choice of a tagging scheme for that machine.

Some machines, including the M88000 and SPARC have instructions to load and store two words at once. These instructions take 3 cycles instead of 4 to load and store double wrappers, which may make this representation more attractive.

It is only realistic to count a memory access as one cycle if the memory word being accessed is in the cache. For words of machine code, this is generally reasonable, but for data it is less applicable, and depends a great deal on the access patterns of individual programs. In particular the times given to wrap, unwrap, and test data for the object-pointer strategy will be much worse if the program does not access data in a favorable sequence. Also, the extra memory used by the method of double wrappers may make cache behavior worse. Of course similar comments apply to the caching of virtual memory in physical memory.

On machines that always have `0b00` as the lower two bits of a word pointer, if integers are given `0b00` as a tag, then the garbage collector can traverse a stack that contains machine pointers (such as return-addresses and stack pointers) and ignore everything that has a `0b00` in the lower two bits. There is no need to tag these machine addresses or otherwise distinguish them from wrapped values. Note that this only applies to correctly initialized locations (not to arbitrary registers for example), and only applies if the code is not relocated during garbage collection.

On segmented architectures like the Intel 80x86 family, many different addresses can specify the same location. On these machines, the technique of partitioned word representations can be used without breaking up the allocation space. The lower half-word of the pointer just has to be chosen so that it points to the right space, given the tag in the upper half-word. Similarly, some machines have a word with more bits than are needed to address the addressable memory and ignore bits above those needed. These extra bits can be used in a partitioned word scheme with no restriction on where objects are allocated.

9.4 Automatic Choice of Representation

Clearly the choice of the best representation scheme involves complex cost analysis; but given a finite number of possible representations, it looks like something that could be automated. In other words, it looks like one could write a heuristic that would take a machine description and some sort of description of type frequencies and produce a good representation scheme. Of course the number of representations is exponential in several parameters, so asking for an optimal scheme is probably not reasonable. There has been a program written [19, 18] that actually implements a great deal of this suggestion for a subset of the possible representations ([19] contains an excelent summary of data representations used in many language systems). This program is intended to help decide on representations for a given implementation of a runtime system, but there is no reason in principle why such an automatic choice of representation cannot be made for each individual program. The trade-off is that the runtime system must be compiled uniquely for each representation.

acknowledgements

Many of the techniques discussed in the sections on tagging and tagged arithmetic were brought to the attention of the author during discussions with Saumya Debray and Koen De Bosschere. It is not entirely clear which of the techniques were already known to one of us and which ones we reinvented. Saumya Debray, David Bartley, Andre Marien, Mikael Pettersson and Mark Tillotson offered many helpful comments on an earlier draft of this paper.

During the writing of this paper the author posted an article to the internet, asking for information about any representations that were not already discussed in the paper. Over 30 people responded with helpful ideas and suggestions, and some of these replies lead to further useful exchanges. The author wishes to thank the respondents who were so helpful in this project. In particular electronic mail exchanges with Paul Tarau, Aubrey Jaffer, Stavros Macrakis, and David Keppel had a substantial impact on this paper.

Glossary

alignment restrictions. Machine restrictions on the addresses of objects stored in memory, based on the size of the object.

BIBOP. an abbreviation for Big Bag of Pages. A technique for representing values with dynamically typed segments using a table of memory pages associated with types (section 3.2.1).

boxing a number. allocating space for a number, copying it to memory, and and representing it with a wrapper containing a pointer to the value.

cycle. On a machine in which most instructions take the same amount of time to fetch and execute, a cycle is the time required to fetch and execute one of these instruction.

delay slot. an instruction position after a jump that will have its instruction executed whether the jump is taken or not.

direct representation. a representation of a wrapped value that is entirely within the wrapper —no part of the value is kept elsewhere. The most common type to have a direct representation is a small integer.

dispatching on the type. branching to one of several locations depending on the type of a dynamically typed value.

double wrapper. a large wrapper consisting of two machine words (section 5.1).

excess-precision arithmetic. integer arithmetic done in such a way that the result might have more precision than can be handled in the final representation. This requires a overflow check when the final representation is constructed from the result (section 2.3).

hooked value. a value that requires special handling when it is referenced (section 4.2).

hybrid representation. a representation using two or more of the basic representational techniques (section 7).

immediate value. a constant used in a machine instruction that is encoded in the instruction itself.

machine value. a type of value that is supported directly by machine operations.

large wrapper. a wrapper consisting of more than one machine word (section 5).

object. a block of memory that contains the information needed to determine what sort of value the block represents. In the literature this term has widely varying meanings and seldom (if ever) means what it does here.

object pointer. a representation of a dynamically typed value where the wrapper contains a machine pointer to an object, a block of memory that can be identified by its structure (section 4).

partitioned word. a representation of a dynamically typed value where the set of possible bit patterns of a machine word is divided up among the various types (section 3).

partitioned word (by magnitude). a partitioned word representation where each type is given a sequential range of integers for its representation.

preboxing. boxing some numbers before the start of the program and using these boxed values whenever they are needed, instead of boxing new versions of them (section 4.3).

qualifier. a representation of a sequence that makes use of a pointer and a length or two pointers, both separate from the sequence itself. Elsewhere this term has various other meanings.

range test. testing a number to see if it is in a given range (section 3.1).

scaled arithmetic. integer arithmetic done in such a way that the result will have exactly the precision that can be handled in the final representation, although the result may not be positioned correctly for the final representation (section 2.3).

segment. a range of numbers in a segmented range. In the literature this term is often used in other ways.

segmented ranges. a partitioning of a set of natural numbers by magnitude such that membership in a range is decidable by some subsequence of the higher-order digits in the binary representation (section 3.2)

segregated type codes. a typed location representation where the type code is encoded dynamically, but separately from the value (section 6).

sign-extension tag. a tag of either all ones (for negative integers) or all zeros (for non-negative integers) used to tag integers.

significand. the part of a floating point representation that encodes the mantissa.

stage. a single level of representation in an overall staged representation scheme.

staged representation. a representation where one or more of the type codes are shared among several types that are distinguished from one another by another representation scheme.

staged tags. a staged representation where all stages use tagged words.

tag field. a section of a tagged word that is used to represent type information.

tag. a type code in a tagged representation.

tagged word. a representation of a dynamically typed value where the wrapper contains a single word divided into separate fields for representing the type and value.

tagging. setting the tag field of a word.

type. a set of values and a set of representations for those values at the implementation level. Elsewhere, this term has various other meanings.

type code. an integer (or bit pattern) used to represent a specific type.

typed location. a representation of a dynamically typed value where the type of the value is known by the location of the value.

unbound variable. a variable that has not yet had a value assigned to it. This term applies especially to languages where it is legal and well-defined (in some way) to refer to the value of a variable before the value is assigned (eg: Prolog).

unwrapping. converting a wrapped value to an unwrapped value.

untagging. removing the tag from a tagged word. This may or may not be equivalent to unwrapping a tagged value.

value field. the part of a tagged word that represents the value.

word. either a unit of memory of the same size as a machine register, or the set of values representable in a machine register or memory word. For example "word" in "tagged word" refers to the representable values of registers and memory words.

wrapping. converting an unwrapped value to a wrapped value.

wrapper. the part of the representation of a wrapped value that is copied from place to place when the value is passed from one procedure to another, returned from a procedure, assigned to a variable, etc. This does not include any portion of the representation that is accessed indirectly (say through a pointer) and does not move around when the value is moved around. In most representation schemes the wrapper is a single machine word.

wrapped value. a dynamically typed value encoded with type information.

References

[1] Robert L. Bernstein. "Producing good code for the case statement". *Software — Practice and Experience*, 15(10):1021–1024, October 1985.

[2] Koen De Bosschere, Saumya Debray, David Gudeman, and Sampath Kannan. "Call forwarding: A simple interprocedural optimization technique for dynamically typed languages". In *Proc. 21st ACM Sym. on Principles of Programming Languages*, Portland, OR, January 1994 (to appear).

[3] Timothy Budd. *A Little Smalltalk*. Addison-Wesley Publishing Company, Reading MA, 1987.

[4] S. Duvvuru. "Monaco: a high performance implementation of FGHC on shared-memory multiprocessors". Technical Report CIS-TR-92-16, Department of Computer and Information Science, University of Oregon, 1992.

[5] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. The MIT Press, Cambridge, MA, 1985.

[6] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, Reading MA, 1985.

[7] Benjamin Goldberg and Michael Glover. "Polymorphic type reconstruction for garbage collection without tags". In *Proceedings of the 1992 ACM Conference on LIsp and Functional Programming*, pages 53–65, San Francisco CA, June 1992.

[8] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ, 1986.

[9] David Gudeman, Koenraad De Bosschere, and Saumya K. Debray. "`jc`: An efficient and portable sequential implementation of Janus". In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 399–413, Cambridge, MA, 1992. MIT Press.

[10] Kevin Hammond. "The spineless *tagless* G-Machine — NOT!". Technical Report (not yet published), Department of Computer Science, Glasgow University, 1993.

[11] Samuel P. Harbison and Guy L. Steele Jr. *C A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo CA, 1990.

[13] Douglas Johnson. "Trap architectures for Lisp systems". In *Proceedings of the 1990 ACM Conference on LIsp and Functional Programming*, pages 79–86, Nice, France, June 1990.

[14] Simon L. Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". *Journal of Functional Programming*, 2(2):127–202, April 1992.

[15] Guy Lewis Steele Jr. "Data representations in PDP-10 MacLisp". In *Proc. 1977 MACSYMA Users' Conference*, Washington, D.C., July 1979. NASA Scientific and Technical Information Office.

[16] Atsushi Nagasaka, Yoshihiro Shintani, and Tanji Ito. "Tachyon Common Lisp: An efficient and portable implementation of CLtL2". In *Proceedings of the 1992 ACM Conference on LIsp and Functional Programming*, pages 270–276, San Francisco CA, June 1992.

[17] Michael O. Newton. "A high performance implementation of Prolog". Technical Report 5234:TR:86, Computer Science Department, California Institute of Technology, April 1987.

[18] Stan Shebs and Robert Kessler. "Automatic design and implementation of language datatypes". In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques*, pages 26–37, St. Paul, Minnesota, 1987. ACM Press.

[19] Stanly T. Shebs. "Implementing primitive datatypes for higher-level languages". Technical Report UUCS-88-020, University of Utah, 1988.

[20] Peter Steenkiste and John Hennessy. "Tags and type checking in LISP: hardware and software approaches". In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59. Computer Society Press of the IEEE, October 1987.

[21] Paul Tarau. personal communication about his implementation of BinProlog. April 1993.

[22] Kenneth Walker. "The implementation of an optimizing compiler for Icon". Technical Report TR 91-16, Department of Computer Science, The University of Arizona, August 1991.