EusLisp: An Object-Based Implementation of Lisp

Toshihiro MATSUI* and Masayuki INABA**

* Autonomous Systems Section, Electrotechnical Laboratory
Umezono, Tsukuba-city, Ibaraki, 305 JAPAN

** Department of Mechanical Engineering, University of TOKYO,
Hongo, Bunkyo-ku, Tokyo, 113 JAPAN

Abstract

This paper presents an object-based implementation of EusLisp. In EusLisp, every basic type of Lisp structured data — even cons or symbol — is represented as an object and its behavior is described in a class. Thus, by using only a limited set of object-oriented primitives, all the Lisp built-in functions can be defined in a fully uniform manner. Unlike CLOS, EusLisp allows subclassing of these system classes, and the system provides high extensibility. To keep the performance of object-oriented Lisp comparable with traditional Lisp and object-oriented languages, EusLisp has employed an efficient type discriminating method for tree-structured class hierarchies by making use of the simplicity of single-inheritance, and flexible memory management based on the Fibonacci buddy system. As a result, EusLisp demonstrates fair performance in benchmarks and in a practical application of a geometric modeler.

1. Introduction

The m;any superior characteristics of object-oriented programming such as modularity, inheritance, parallelism, modeling, and simulation have been recognized, and many object-oriented Lisps have been proposed. EusLisp is an object-based Lisp whose goal is the realization of a geometric modeler for robotics applications. In this paper, the strategy taken in EusLisp for the efficient integration of Lisp and object-orientation and the EusLisp implementation are presented.

The object-oriented programming facilities for Common Lisp[1] are converging in the ANSI standard CLOS (Common Lisp Object System) proposal[2]. CLOS extends traditional object-oriented languages in a more flexible and powerful direction by introducing many new concepts, such as generic functions, multi-methods, multiple-inheritance, meta-objects etc. Despite the advantages of these facilities, a most severe obstacle of CLOS remains: the difficulty of implementing these features efficiently enough in a reasonably simple manner.

Major factors which determine efficiency of the object-oriented languages are slot access, method dispatching, and instance creation [4]. In CLOS, multiple inheritance slows down the performance of slot access, and multiple-inheritance and multi-methods tend to degrade the method dispatching. To maintain higher performance, CLOS requests the compiler to do much optimization and the runtime routines to use complicated methods such as slot-name hashing and method caching. Furthermore, CLOS is often implemented on existing Lisps by changing their evaluator; but these Lisps usually attach great importance to list processing and we cannot expect their instance creation ability to be efficient enough.

To cope with these problems, Euslisp takes following strategies. First of all, instead of multiple-inheritance which degrades runtime performance and increases the complexity for method selection, we use single-inheritance as the basis. For the cases where multiple-inheritance is needed, we provide a mechanism called message-forwarding. In place of generic functions and multi-methods, which depend on the performance of type dispatching, we provide a constant-time type checking mechanism for the inclusion test in hierarchical class inheritance. This type checking is so efficient that it can be used in the EusLisp kernel and that all the built-in data types such as *cons* or *symbol* can be implemented with standard classes. Thus, having a built-in class as a super-class of user defined classes, system functions can be incrementally extended and the creation and management of built-in objects can be described in EusLisp. In order to meet the size varying and frequent requests for memory allocation, we use the Fibonacci buddy[7,8] method for memory management. To deal with the vast consumption of conses which is innate in Lisp, we can leave a particular number of heap pages unmerged when they are reclaimed, thus reducing time for dividing heaps to get small cons cells. This method benefits from a high memory usage ratio as well as ease of management since no copying and pointer redirection is required.

In the following section, advantages of defining built-in types as classes are discussed and compared with CLOS. The third section introduces the syntax of object-oriented primitives in EusLisp. In the fourth and fifth sections, the structure of objects and high speed type checking and the characteristics of the buddy memory management are presented. The sixth section describes the foreign language and window system interface as examples in which the hierarchical type system of EusLisp is effectively used. The last section demonstrates the performance of EusLisp using some benchmarks and a geometric modeler application.

2. Object-Oriented Programming in EusLisp

2.1. Type Hierarchy

The concept of class is an extension of type. For the integration of Lisp and object-oriented programming, relationship to Common Lisp data types must be taken into consideration. Common Lisp has defined more than thirty data types[1], seventeen of which have corresponding classes in CLOS[3]. The rest are left undefined since either Common Lisp does not specify the relations between types in terms of supertype/subtype or ambiguities are left in the precedence order among supertypes. Functions such as *typep* and *subtypep*, however, remain applicable to all classes, and types in Common Lisp can appear as type specifiers in generic function definition.

Each of the Common Lisp types which has a corresponding class can be implemented either as a standard class defined with *defclass*, as a structure class defined with *defstructure*, or as a built-in class defined in an implementation dependent manner. Built-in classes have the following restrictions compared to standard class: (1) they do not have slots, (2) they use special functions for instance creation, and (3) they cannot have user-defined subclasses.

If basic data types like *cons* and *symbol* were implemented as standard classes in CLOS, on the one hand, almost all the Common Lisp functions would need to be redefined using *defgeneric*, thereby incurring much run-time overhead in method dispatching, and making in-line expansion by the compiler unavailable for generic functions. Also, if the object system is implemented on top of existing Lisp, like PCL (Portable Common Loops)[5], it is actually impossible to define Common Lisp types with standard classes. Therefore, almost all the implementations of CLOS support Common Lisp types with built-in classes.

On the other hand, if basic system data types could be defined with standard classes, two major advantages would arise: extension and integration of the system through the class inheritance could be achieved, and Lisp basic functionalities would be described in terms of objects. EusLisp represents all data but numbers as objects, and defines Common Lisp data types with classes. In this case, built-in data types can be extended by user defined subclasses. To apply Common Lisp functions to these extended objects, type checking according to hierarchical inheritance is important. Consequently, by restricting the class hierarchy to single inheritance, EusLisp implements an efficient type checking mechanism. To make up for the drawback of single-inheritance, message-forwarding facilities are added, providing a quasi-multiple-inheritance structure. The only exception to EusLisp object orientation is the number. Numbers are represented only in immediate forms, so that highest performance could be available in the numeric computations for the description of a geometric modeler, which is the ultimate goal of EusLisp.

2.2. Extensions and Integration through the Class Inheritance

From its origins as a classical symbol and list processing language, modern Lisp has been developed to manipulate numerous kinds of data types, and the trend of type extension will most probably continue in order to cope with the new environments that utilize networks, windows, etc. Defining Common Lisp built-in types with classes, the extension of system functions through the class inheritance is facilitated. Several examples showing the effectiveness of such extensions are discussed here.

Common Lisp has two categories of streams distinguished by a data source/sink object connected at the endpoint of each stream, i.e. *string-stream* and *file-stream*, although no predicates are provided to discriminate between them. As for output, although both streams behave similarly for the operation of filling data into buffers, creating and flushing methods are different: a *file-stream* needs to call *open* function at creation time, and it commits the OS to flush the contents of the buffer through a *write* operation. Therefore, it is appropriate to define *file-stream* as a subclass of *string-stream*, adding *open* and *write* methods. A channel for interprocess communication (IPC) could be regarded as a variation of the stream type too. The sockets of UNIX use *read* and *write* system calls for input and output, but needs to issue other system calls such as *connect* for the creation of a channel. Thus, by defining *socket-stream* as a subclass of *file-stream*, duplicated descriptions could be avoided.

EusLisp defines the *symbol* type as a subclass of *propertied-object* which defines the *get* and *putprop* functions. Thus, management of property lists is separated from *symbol*, and it becomes possible for other objects to use property lists. EusLisp functions callable from other languages are defined as instances of *foreign-symbol*, which is a subclass of *symbol* and includes a code portion for argument conversion between the languages as well as a definition of a function body which behaves as a usual Lisp function.

Cons could be extended to have an extra field for the representation of a list with reverse pointers, or classes like *alist, set* and *stack* could be derived from *cons* according to each usage. *Alist* and *set* are data structures whose purpose is the search by *assoc* or *member* function respectively. These functions are made generic for the arbitrary structure of lists by means of *:key* and *:test* arguments. If extended *cons*es instantiated from *alist* or *set* have additional slots where the *:key* or *:test* functions are held, these arguments for the invocation of *assoc* and *member* can be omitted, and these functions could become inherently generic. This interface agrees with *hash-table*, which allows a test function to be specified at the time of *make-hash-table*.

Closure is defined under *compiled-code* with the information for the lexical environment when it is created. Program code of a foreign language becomes callable from EusLisp as an instance of *foreign-code* which extends *compiled-code* to have a slot for linking information. Classes *cstruct* and *foreign-string*, which inherit *string* class are also added for the purpose of the foreign language interface.

2.3. Object-Based Implementation of Lisp

Usually in Lisp, basic operations for data creation and access are implemented as built-in functions. Actually, Common Lisp has many atoms with internal states such as *readtable*, *hash-table*, *pathname*, *package*, etc., and also many individual functions for their creation and access. This could be contrasted to the user-definable structure that can automatically generate consistent constructors and accessors. This is so because every built-in type mixes Lisp pointers and binary data in an inconsistent slot structure to put the highest priority on the efficiency of time and space, and it is necessary to implement these primitive functions in another language which underlies Lisp.

Again, by defining basic data types with standard classes, data structures are arranged with sequences of slots, and consistent object creation and access are achieved. In EusLisp, all the primitive functions to access Common Lisp objects can be described with EusLisp functions. Also, the single *instantiate* function is enough for the creation of all the basic atoms. In this implementation, even the basic functions like *car*, *cdr*, *rplaca*, *rplacd* and *cons*, which are absolutely primitive to usual Lisps, become superficial. The EusLisp definition of these functions follows. In contrast with CLOS, which adds object-oriented programming on Common Lisp, EusLisp implements Lisp on the basis of object-oriented programming.

```
(defun car (x)
   (if (null x) nil
       (if (consp x) (slot x ’car) (error "not a list"))))
(defun rplaca (x a)
   (if (consp x) (setslot x ’car a) (error "not a list)))
(defun cons (a d &aux (c (instantiate cons)))
   (setslot c ’car a) (setslot c ’cdr d) c)
```

3. Syntax of EusLisp

3.1. Class Definition and Method Definition

Classes are defined by the *defclass* macro:

```
(defclass <class-name> :super  <superclass-name>
                       :metaclass  <metaclass-name>
                       :slots  (slot-description ...))
```

Employing single-inheritance, EusLisp allows only one superclass argument. A subclass of class *metaclass* can be given in the place of *:metaclass* argument. If no metaclass facilities are required, class *metaclass* is defaulted to every non-vector type class (refer to **4.2**), and class *vector-class* to vector-type class. For a slot-description, the data type (class name) and a forward specifier, which will be detailed in **3.4**, can be given along with the slot variable name. Slot variable names cannot collide with those defined in the superclasses. After evaluating a *defclass* form, a class object (i.e. an instance of *metaclass*) is created and is bound to the special value of the class name symbol, which is proclaimed to be special at the same time. Though we could have chosen another choice to store the class information in the property list of the symbol, it is too costly to search in plist whenever type checking is performed; also, dynamic binding provides a convenient way to alter the class that is referenced by an instance creating function, as is described in section 3.2.

Methods are defined by the *defmethod* special form.

```
(defmethod <class-name>
   (<method-selector> <lambda-list> . body) ...)
```

Each method definition looks like a function definition. When a method is invoked, however, not only variables that appear in the lambda-list, but slot variables and variables named *self* and *class* are also made visible. *Self* is bound to the object which received the message and *class* refers to the class object where the method invoked is found.

3.2. Instance Creation

The *instantiate* function is commonly used to create a new instance of a class.

```
(instantiate <class-name> [<size>])
```

Instantiate initializes all the slots to *NIL* for pointer objects or to zero for binary objects. For example, *(instantiate cons)* produces a dotted-pair (*NIL . NIL*), and *(instantiate float-vector 2)* yields #*f(0.0 0.0)*. No individual initializer can be defined, since users can customize the initialization procedure by defining *:init* method which is automatically invoked by the *instance* macro. To create a vector-type object, the second <size> parameter must be given unless the vector has been *defclass*'ed to have a fixed number of elements. To allow built-in functions such as *list, intern* and *open*, to refer to a user-defined class for the template of instance creation, special values of *cons, symbol, file-stream*, etc. should be bound to the user's class.

```
(defclass mysymbol :super symbol :slots (myvalue))
(setq x  (let ((symbol mysymbol)) (intern "XXX")))
```

An object is copied to another object by the *replace-object* function, which performs so called shallow copying. Also, *copy-object* performs so called deep-copying, that is, every object which is referenced from each slot in the object is recursively copied. Copying of multiple objects is performed efficiently preserving mutual references by using two mark bits in each object's header (see **Fig.4**) and a copy-done table allocated on the stack. This *copy-object* function is important in the geometric modeler application where the complex data structure of a model is often requested to be copied to duplicate a shape, or to be preserved before irreversible composition operations. Without this generic deep-copying function, we must define copying functions for each element of shape models in an ad-hoc manner.

3.3. Message Sending

Send function is used to send a message to an object and to invoke a method.

```
(send <receiver> <method-selector> . args)
```

A method to be invoked is searched for first in the direct class of the receiver, and then in its superclasses in turn toward the ultimate superclass *object*. If no method with the same method-selector is found, *message-forwarding* is tried (see **3.4**). Then, if still no appropriate method is found, a method named *:nomethod*, which is always found in the *object* class, is looked for. Method combination facilities are not provided. Specifying *self* for the receiver, messages can be sent recursively to the receiver. Using the *send-super* macro, method searching is directed to start from the superclass of the class where the method being executed is defined. The method search procedure uses the method cache which is global to every class and has 512 entries. Each entry is hashed with the combination of the address of the class object and method-selector. In geometric modeler applications, a hit ratio of more than 99% has always been observed.

3.4. Message-Forwarding

In contrast with the single-inheritance of EusLisp, most of modern object-oriented languages including CLOS allow multiple-inheritance. However, in our early experience on LEO[13], which experimentally implemented multiple-inheritance on a small Lisp for robotics, we soon encountered a few problems with multiple-inheritance. The first problem is duplication of methods and slots between several superclasses; we must always pay attention to the order of message passing to superclasses and the consistency between parameter lists. For an example, *robot* may be defined to have *vehicle* and *manipulator* as its superclasses, both of which have a *:move* method. But we cannot simply send a *:move* message to a *robot*, since the movement of the *vehicle* or *manipulator* depends on the precedence list for superclass combination. Eventually, we need to add a new *:move* method in the *robot* class to discriminate based on given arguments between two superclasses to which the *:move* is actually to be sent. Second, there are cases where it is hard to determine whether we should reference a component as a superclass or a slot variable. In the above example, the *manipulator* class may be defined either way. If we change the *robot* to have two arms on a vehicle, however, two *manipulator* objects should be bound in the separate slots of a *robot*, since the inheritance of two *manipulator* classes is not allowed. This drastic migration from a superclass to a slot affects much code scattered in all the classes in the inheritance tree. The third problem is the efficiency. Although method dispatching is relatively easily overcome by method caching, slot variable access requires searching some table at runtime to find correct slot locations for each instance. In addition, a test to see if an object is made from a subclass of a particular class necessitates traversing in the inheritance tree, since the faster class discriminating method detailed in the next section cannot be applied.

Despite these problems, multiple-inheritance languages have demonstrated their effectiveness and flexibility in handling many problems: it is well known that the elaborate programming environment of the window system on Lisp Machines highly depends on the multiple inheritance of the Flavor system[6]. Flavors has reduced the amount of duplicated description for new window tools by combining existing component classes like software ICs. Also, single-inheritance cannot represent such a hierarchy where *vector* is a subclass both of *sequence* and of *array*.

Usually, it is impossible to alter the superclass of a class once it has been defined, but the slot values can be changed at runtime. In other words, the *part-of* relation is more flexible than the *is-a* relation in a normal object-oriented language. EusLisp provides *message-forwarding* facilities to achieve a balance between the simplicity of single-inheritance and the flexibility of multiple-inheritance: if a message is sent to an object and no applicable method is found either in its class or in the inherited superclasses, then a slot which has a forwarding specification in the slot-description is searched for, and the message is transferred to another object bound to the slot. Method names which should be transferred can be listed in the slot-description with the *:forward* keyword, or if *T* is specified, every message is forwarded. This provides the equivalent function of multiple-inheritance that does not share slot variables between classes in the inheritance. In the example illustrated in **Fig.2**, a *:telephone* or a *:mail* message sent to a *president* is dispatched to a *secretary*, and a *chauffeur* is delegated for a *:go-home* message. Taking advantage of the method cache, message-forwarding can avoid the overhead of method searching.

One of our applications of EusLisp to 3D graphics needed to draw models both on an Xwindows display and on a special display hardware, the *Multi-Media Display (MMD)*[16]. MMD manages models in its own storage and a specific ID is assigned to each model. Therefore, we defined the class *MMDagent* which manages the correspondence between the models in EusLisp and MMD. Referencing the *MMDagent* object through *message-forwarding*, we could use both of these display systems without changes in the superclasses and without duplicated description among the many MMD models.

3.5. Slot Variables and Access

When a message is sent to an object and the evaluation of a method body begins, all the slot variables defined in its class and superclasses become visible in the same manner as local variables introduced by *lambda* or *let*. When the body is evaluated interpretively, slot position is linearly searched for in the slot variable name vector within the class object. Once compiled, a slot is directly accessed with the offset from the origin of the object without any searching or hashing. With this access method, the definition of function *car* given at the end of previous section can be written as follows, although it cannot handle *car* of *NIL*:

```
(defmethod cons  (:car () car))
(send ’(a b) :car)  --> a
```

It is also possible to access slots from outside an object. When a class is *defclass*ed, accessor macros for all the slots, whose names are composed by concatenating the class name and slot name, are automatically generated. For example, the *car* of a *cons* can be taken by an expression like (cons-car ’(a b)). These macros are recognized by *setf* to allow slot values to be updated as generalized variables.

In case of reading and writing a slot value, specifying the slot name dynamically, the *slot* and *setslot* functions, to which the accessor and updater macros mentioned above are expanded, are provided. Each of these functions takes a class name as its second argument. Although the class name can be known at run time from the object given as the first argument, this advisory information can direct the compiler to search the slot position at compile time. From the view point of data abstraction, slots should be protected from any trespass. These features are included, however, to allow accessing method compatible with *structure* defined by Common Lisp.

4. The Structure of Objects and Efficient Type Discrimination

4.1. The Structure of Pointers

The representation of pointers affects much the approach of object-oriented programming and performance of Lisp. One of the important characteristics inherent in objects is that every object carries information about the class it belongs to. From this point of view, it is an improper idea to tag pointers with all type information as in some traditional Lisps. Besides, since the number of classes dynamically increases as a user defines his own, a discriminating method which makes use of predetermined address ranges is hard to apply. EusLisp pointers have only two tag bits, indicating number or address, and class information is designated by the class-id field in the header of each object.

A pointer is represented by a 32-bit long-word, as is depicted in **Fig.3**. Assuming each object cell is aligned with a long-word boundary, the upper thirty bits can be used for an immediate number value or an address pointing to an object, and the lower two bits can be assigned to tag. The tag is interpreted as follows: 00 for integer, 01 for floating-point number (abbreviated floating, hereafter), 10 for address, and 11 for unused. No long floating or long integer form is provided. This pointer representation has the following characteristics: (1) full 4Gbytes of memory space is addressable, (2) immediate values can represent integers with the range of nine decimal digits and floating with 10^{-6} computer epsilon, precise enough for practical robot applications, (3) integer addition and subtraction can ignore the tag, and (4) slots in objects are directly addressed at an offset of -2.

4.2. The Structure of Objects

An object is composed of a one-word header and subsequent slots, which are shown in **Fig.4**, with the meanings of each field in the header. Every object of either user-defined or built-in classes has the same framework, and uniform memory allocation and garbage collection are applied.

A vector type object is distinguished from a non-vector type one through the *elmt* field in the header (**Fig.4**), which represents the type of every element (see **Fig.5**). A non-vector type object has a fixed number of slots determined by each class, and each slot is accessed by its slot-variable name. A vector type object can have any number of slots, determined at instantiation time. The first slot of a vector object is assumed to have the number of elements and each element is accessed with an index from the second slot. Only vector objects can hold binary data, and the representations for bit, char, integer, and floating are compatible with other languages and can be used interchangeably between EusLisp and foreign languages. Since the EusLisp objects never migrate in the heap as described in the next section, they are safely addressed from C's pointers. The *bix* (buddy index) field is an index representing the physical size of a cell and is used by the memory manager in conjunction with the *mark* field. The class to which an object belongs is represented by a 16-bit *cid* (class-id) field. *Cid* is really an index into the *class-table* from which actual class objects are addressed. *Cid* representation not only saves space normally used in all objects for pointers to class objects, but also enables fast type inclusion checking using relationships between their magnitudes.

4.3. Rapid Type Discrimination

Common Lisp has an established and complicated type hierarchy. In this hierarchy, functions defined for a type are also applicable to its subtypes, but not to supertypes or disjoint types. So, when a function is applied to data, it is necessary to check whether the data comes from the class for which the function is defined, or from a subclass. This type checking greatly affects the overall performance of the system[9].

In a CLOS implementation where built-in data types are restricted not to be inherited, it is possible and effective to optimize tag encoding to get the best result. For example, one may choose the tag bit for a *list* to be the most significant bit of the header so that it directly appears in the sign flag of the processor. On the other hand, since EusLisp allows dynamic extension of basic data types, tag encoding cannot be statically fixed. Pure object-oriented languages rely upon method searching for this type discrimination. The mechanism in CLOS of dispatching to a specific method through a generic function is also accompanied by searching, although this may be improved by caching methods. Of course, method searching is the first thing that slows down the execution of object-oriented languages, and a great disadvantage in efficiency would arise in a Lisp that compiles primitive functions in line. EusLisp functions discriminate between types in the following manner, taking advantage of single-inheritance and the *cid* encoding of class information in each object.

The problem is how to check whether the class of an object inherits another class or not. For example, in the inheritance structure described in **Fig.6**, to see D as a subclass of A, tracing the inheritance chain in the order of D, C, B, A requires time linear with the depth of the inheritance tree. In EusLisp, successive cid's are assigned to each class in an inheritance tree. Let the direct subclasses of class C be C_1 ... C_n, then their *cid* and *maxsubcid* are determined according to the following formulae. Here, *maxsubcid* means the maximum cid in the subclass tree.

no subclasses:	C.maxsubcid = C.cid
with subclasses:	C1.cid = C.cid + 1
	Ci+1.cid=Ci.cid + 1
	C.maxsubcid= Cn.maxsubcid+1

Each of these *cids* and *maxsubcids* for all classes is entered in the *class-table*. To see if an object x belongs to class C, x.cid is tested to determine whether it lies between C.cid and C.maxsubcid ($C.cid \le x.cid \le C.maxsubcid$), a process which needs at most two comparison operations.

To maintain the relations in the formulae above, re-assignment of *cids* for all the existing classes and objects is required when a new class is defined. In **Fig.6**, for instance, if class F' is newly defined under F, following procedures are executed: (1) assign (F.maxsubcid +

1) to the *cid* of F', (2) increment the *maxsubcid* of every superclass of F' by one, (3) for each class having a *cid* greater than that of F', increment its *cid* and *maxsubcid* by one, and shift downwards its entry in the *class-table*, and (4) scan objects in the heap and increment *cids* by one if they are greater than F'.

Process (4) takes a time proportional to the total number of objects. We performed an experiment to define a subclass under *cons*, which happens to the *cid*s of most instances to be updated. Thereby, we observed the time for a new class definition to be almost identical to 15% to 20% of the time required for a garbage collection, which also requires linearly increasing time to the amount of memory occupied by objects. This may be a crucial problem for an application in which many classes are dynamically defined and destroyed. For robotics applications, however, we have never encountered with such dynamic problems, and classes are always defined at load time.

After this class ordering, inheritance trees are discriminated with simple range check instructions emitted in line by an intelligent compiler. Actually, the *subclassp* function for the test of subclass inclusion and the *derivedp* function, which checks if an object is instantiated from a specified inheritance tree, are executed in constant time.

5. Memory Management

There is an apparent characteristic of memory usage in object-oriented Lisps: whereas variously sized memory cells are frequently requested, the allocation of the smallest cons cells is still the greatest. **Fig.7** shows statistics on the number of memory requests and the quantities allocated for every memory object size observed during the execution of a geometric modeling application in EusLisp. In order to achieve flexible and efficient memory management, EusLisp has employed a variation of the Fibonacci buddy method.

5.1. The Fibonacci Buddy Algorithm

Buddy memory management is roughly described as follows (refer to **Fig.8**): (1) on a memory request, a larger cell is split and the smallest cell which can satisfy the request is returned; (2) if an unused cell is found during garbage collection, its adjacent buddy is tested and merged if it is also unused; this merging is recursively repeated to form the largest unmergeable cell, which is then reclaimed. To facilitate the split and merge procedures, the sizes of cells are restricted to powers of 2 or Fibonacci numbers, etc. The former is referred to as the binary-buddy system and the latter, the Fibonacci-buddy system.

As the algorithm and implementation of the Fibonacci buddy method are detailed in the literature [7], only the outline is stated here. Since the Fibonacci buddy method merges adjacent cells of different sizes called buddy pairs to form a larger cell, a tag bit *b* is required to indicate which side of the cell its buddy should be found (**Fig.4**). One more tag bit *m* is also necessary to remember the *b* and *m* bits of the cell that was split. When a cell with a *bix* of n is split, the *b* of the child cell with a *bix* of n-1 is set to 0 and the *b* of the other child with a bix of n-2 is set to 1. The parent's *b* and *m* bits are saved in the *m* of each child. When a cell is reclaimed, the address of its buddy is calculated by using the *bix* (buddy size) field in conjunction with the *b* bit. If it is also a garbage cell, the buddy pair are combined and *b* and *m* are restored.

Since a Fibonacci buddy can have more cell size variations than a binary buddy, better memory usage is accomplished. According to the analysis of Peterson and Norman [8], internal loss (the fraction of unused space in a cell) is estimated to be approximately 22% for the Fibonacci buddy and 30% for the binary buddy. With a 5-bit bix field, 32 kinds of cell sizes are prepared (see second column of **Fig.7**), and the largest cell is bigger than 16 Mbytes. Through measurement, the actual internal loss has been observed to range from 10% to 20%.

5.2. Suppression of Merging

Since Lisp consumes many conses, which are the smallest data structure, external fragmentation in the buddy system can be ignored, and the total memory efficiency is fairly high. The most critical problem with the Fibonacci buddy is the cost of the split and merge procedures. This is because, upon the request for a cons, which occurs most frequently, the larger cell, formed in preparation for massive memory requests at great expense, always needs to be split into the finest pieces. The EusLisp garbage collector, therefore, leaves a particular amount of memory unmerged. These unmerged regions are filled up with the smallest cells during execution. Thereafter, conses are allocated without calling the splitting procedure. The proportion of unmerged memory is controlled by the **gc-merge** parameter, which can dynamically be changed. Larger values of **gc-merge** speed up programs that make many conses, and smaller values improve the memory efficiency of allocation of variously sized object.

5.3. Comparison with BiBoP and Copying Methods

BiBoP (Big Bag of Pages)[10,12] and copying methods[11] are often employed as the traditional memory management strategy in Lisp on stock hardware. The BiBoP scheme manages memory by dividing it into many pages, each of which has objects of the same type or of the same size. BiBoP has many advantages: it allocates fixed sized objects like built-in *cons* or *symbol* very quickly since it only needs to dequeue a free-list, no internal fragmentation occurs for such objects, and the type information can be derived from the page where the object is allocated. Object-oriented languages like EusLisp, however, do not benefit from these points, because built-in types may be extended to have different sizes, and allocating independent pages for each user-defined class seems to be impractical. Moreover, even the BiBoP scheme causes internal fragmentation for variable length objects like vectors, and no little external loss is accumulated, since, if the pages for a type are exhausted, much space for other types are often left unused. In the buddy method, any buddy cell can accommodate any type of object that fits in it, and since cons has the greatest probability of being created, external loss is little or non-existent. Dynamic acquisition of heap memory is possible with both buddy and BiBoP methods, since pages are allocated to a collection of buddies.

The copying method can also allocate memory quickly regardless of object size. However, its memory efficiency never exceeds 50% due to the superfluous space for copying. Although copying improves the paging behavior at runtime by compaction, garbage collection is degraded and complicated by pointer redirection which extends to all the references, even the stack and registers. On the other hand, as the addresses of objects in the buddy system never change, even compiled code can be allocated in the same space and linking with foreign language programs is facilitated. Also, if an object is confirmed to be unreferenced, it can be given back to the memory manager, thus avoiding garbage collection. Managing all the pages uniformly, the buddy algorithm is simply implemented. The EusLisp memory manager, including the garbage collector, consists of only 250 lines of C code.

6. Foreign Language and Window System Interfaces

6.1. Foreign Language Interface

The more heavily Lisp is used in practical applications, the greater the need for integration with other software written in other languages. Particularly on UNIX workstations, if we could make use of existing numerical libraries and graphics packages for window systems originally written in C or FORTRAN, the development of robotics applications in Lisp would be greatly accelerated. EusLisp extends Common Lisp to provide a flexible foreign language interface†. The ability to define subclasses for built-in data types, and the immovable characteristics of data objects in the heap have eased this extension.

Before discussing the foreign language interface, we describe how ordinary Lisp programs are compiled, loaded, and called in EusLisp. The EusLisp compiler transforms a Lisp program into C, and then invokes *cc* (the C Compiler) of UNIX to get an object file of *a.out* format. Each Lisp function is transformed into a static C function with three parameters. A caller lays evaluated forms on the Lisp stack, and passes the number of actual arguments, a pointer to the arguments, and a pointer to an environment, to a compiled function. Every Lisp function always returns a pointer as the result. Toplevel forms such as *defun* and *defmethod* are compiled and located in the *_eusmain* function, which is executed immediately after loading. The *_eusmain* function makes an instance of the *compiled-code*

† In this section, however, we only describe the interface between EusLisp and C, which is most frequently used on UNIX, for simplicity.

class for each *defun*, and puts it in the *function* slot of the appropriate *symbol* object.

There are two major differences between C code generated by the EusLisp compiler and other C programs: how arguments are passed, and the existence of an initializer (_*eusmain*) for function symbols.

6.2. Loading of Foreign Modules

None of the objects in EusLisp migrate once they are allocated in the heap. Neither does object code, which is contained in instances of the *byte-vector* class. Therefore, object code, which needs complicated relocation when moved, can be allocated in the heap in the same manner as strings.

Loading is done by the *load-foreign* macro. The loader forks UNIX's linkage editor *ld* with *-A* option to solve address references incrementally. Since the foreign language module does not define _*eusmain*, no initialization or creation of function symbols is done here. *Load-foreign* returns an instance of the *foreign-module* class, which is a subclass of the *compiled-code* class, and has a symbol table to remember all the address symbols appearing in the object module.

Entries for foreign functions are explicitly given by the *defforeign* macro. First, *defforeign* searches for the address of a particular function in the symbol table of a *foreign-module*, and then makes an instance of the class *foreign-code*, which is a subclass of the class *compiled-code*. *Foreign-code* extends *compiled-code* to have the *parameter-type* and *result-type* slots for data type conversion. Following is a simple example of *load-foreign* and *defforeign*.

```
/* a C function in "sync.c" */
float sync(x) double x;
{ extern double sin();
  return(sin(x)/x);}

eus$ cc -c sync.c
eus$ (setf m (load-foreign "sync.o"))
eus$ (defforeign sync m "_sync" (:float) :float)
eus$ (sync pi)  --> 0.0
```

The *Defun-c-callable* macro defines a Lisp function that is called by C. This bidirectional interface is needed to make use of window systems which invoke C functions asynchronously when mouse events occur†. *Defun-c-callable* makes an instance of the *foreign-symbol* class which inherits *symbol*. This *foreign-symbol* holds a small code piece which provides an individual entry point for each c-callable Lisp function to accept calls from C, together with information for data type conversion.

As shown above, a few subclasses for *compiled-code and symbol* are effectively used to accomplish the foreign language interface of EusLisp. Even after this extension, all the functions and type discriminating predicates except *funcall* could remain the same as before. Lacking subclassing facilities for built-in classes, we would have been obliged to use the property list of a symbol and to add separate data types, necessitating many changes to the Lisp kernel and a slowdown in *funcall*.

6.3. Data Passing between C and EusLisp

Data representation must be converted back and forth between C and Lisp. When a foreign function is called, *funcall* transforms Lisp data to C's representation. Immediate data such as integers and floats are easily converted by simple bit operations. Among many structured objects, bit, byte, integer, and float vectors can be passed to C, since their internal representations are same as C. Thereafter, the vector elements can be shared

† Asynchronous calls to EusLisp functions are often harmful, since the memory management may be severely damaged. In some window systems, however, such a call can occur only during the read system call, before which we can guarantee the consistency of all the pointers.

between C and EusLisp. These interpretations are taken by looking up type information stored in *foreign-code* objects. Even if the type information is omitted, automatic conversion can be performed, because all EusLisp data is tagged to indicate its type. On the other hand, the result type of a foreign function cannot be omitted. When a vector is returned by C, we can choose whether its contents should be copied to an object allocated in the EusLisp memory space, or it should be addressed by an instance of the *foreign-string* class. *Foreign-string* is a subclass of *string*, but manages only the size and address of a string outside of EusLisp memory space.

A C call to Lisp is caught by the code piece in a *foreign-symbol*, which transfers to a common data conversion routine with the information on the types of the parameters and the result. Looking up this information, the conversion routine dumps all the arguments sent from C onto the Lisp stack. After the function specified in the *foreign-symbol* is applied to these arguments, its value is converted to the representation of C again. Since the function body itself behaves in quite the same manner as other EusLisp functions, it is easily debugged without actually calling it from C.

6.4. Window System Interface

The foreign language interface has been applied to the Xwindow interface. After loading the Xwindow library, *Xlib*, using *load-foreign*, and defining as many as 400 *Xlib* functions by *defforeign*, Xwindow programming can be accomplished in EusLisp. Namely, on top of these Xwindow primitives, data structures for *window, pixmap, graphic-context, font,* etc. are defined in classes with their specific behaviors.

In contrast with CLX which implements all the X protocols in Common Lisp, and our previous trial to build an interface between EusLisp and SunView in C, this approach is easier, more efficient and more robust to window system upgrades.

7. Performance Evaluation

To confirm the efficiency of the strategies discussed above, the performance of type discriminating primitives and memory management was measured. Furthermore, the total performance as a Lisp was evaluated by using Gabriel's benchmarks[9]. Every program is compiled and executed on a Sun3/60. For comparison, we have chosen KCL[10] which is a typical implementation of Common Lisp available on a variety of machines. Both languages compile Lisp source via the C language. Timing is measured with *get-internal-run-time* function, paying fair attention to the load average and the amount of real memory. For the looping in KCL, loop counters are folded at 1,000 to use only preallocated fixnum objects.

7.1. Type Discrimination

The most primitive type discriminating functions in Common Lisp are intrinsic predicates such as *consp, symbolp,* and so on. For user-defined structures, type discriminating predicates with the name of the structure concatenated with the "-P" suffix are automatically defined. EusLisp built-in primitives such as *consp* and *symbolp* return T for subclasses of cons and symbol respectively. *Derivedp* is a general type checking predicate used for user-defined classes. As described in the fourth section, the message sending mechanism of object-oriented languages has the equivalent capability with the type checking primitive. The times required for the processing *symbolp, structure-p, derivedp* and *send* are listed in **Table-1**.

Symbolp is compiled in line both in EusLisp and in KCL. For the built-in types, testing for equality with the tag in the object referenced by a pointer is enough for KCL, while EusLisp first confirms that the word is a pointer and not a immediate number, then performs type checking with two integer comparison instructions, rendering EusLisp approximately three times slower than KCL. If a compound range check instruction like CHK2 of the M68020 processor were emitted by C, EusLisp could be close to KCL. Since *derivedp* is not compiled in line and type checking for the second argument is required, it runs rather slowly relative to *symbolp*. For the test of structure-p, we defined three levels of structures, *a* (top), *b* (middle) and *c* (bottom) using *:include* options, and applied *a-p, b-p* and *c-p* predicates to the instance of *c*. They turn out to be several dozen times slower than *symbolp*. But while EusLisp *derivedp* runs in a constant time for every object, however deeply its classes' inheritance may nest, KCL becomes slower as the level of nesting increases. *Send* is further slower than *derivedp* since it needs a method search and extra bindings for *self* and *class*, but the processing time remains constant as long as the cache hits regardless of the levels of inheritance.

7.2. Memory Management

Keeping the heap size of both processes almost the same, the memory allocation times for a cons, a string of length 10, and vectors of length 3, 10 and 30 were measured. EusLisp exhibits better performance than KCL for every kind of vector as well as for cons. The results in **Table-2** are calculated from the time for the creation of 10,000 pieces of objects and garbage-collection time is included. One reason why EusLisp is superior to KCL is that GC happens less frequently than in KCL. EusLisp can allocate the full heap to cons cells and up to 60% of the heap to vectors, for a **gc-merge** parameter of 0.4. On the other hand, as KCL divides the heap into type related pages, cons can only use a half of the heap and vectors can only use 20% or 30% of the heap depending upon the allocation parameters. This is a handicap of KCL, and the tuning of the allocation parameters seems to affect severely the performance of BiBoP method. In EusLisp, the time for garbage collection is almost proportional to the size of the heap, with approximately 1.5 Mbyte of heap marked and swept in 1 sec.

7.3. Total Performance

The results of Gabriel's Benchmarks are depicted in **Table-3**, where *tak* is affected mainly by the performance of the recursive function call, *Boyer* by list manipulation, *Derivative* by making conses, *FFT* by floating-point arithmetic, *Puzzle* by array access and *Qsort* by sequence access. EusLisp can bear comparison with KCL. The poorer performance of EusLisp in *Puzzle* and *Derivative* arises from the insufficient optimization of the compiler: currently, EusLisp cannot substitute tail recursive calls with iterations. Higher performance in *FFT* comes from the immediate representation of floating which saves time and space for making numbers.

7.4. Lisp and the Geometric Modeler

The ultimate goal of EusLisp is to develop a programming system for robots, and the geometric modeler has an important place there. The geometric modeler is software which allows one to define the shapes of three dimensional objects through the composition of primitive bodies, translate and rotate them, compute and retrieve mass properties, and draw them on graphics displays. Implementation of the geometric modeler in object-oriented Lisp has many advantages over the traditional implementation in Fortran: (1) modeling elements can be described with the objects that can abstract their behavior with their topological relations represented by lists, (2) models can be extended to have extra properties through the class inheritance, (3) storing and transferring models is easily done by the use of S-expressions, (4) no memory management troubles and no capacity restriction are imposed, and (5) there is no need to build a special command language.

By employing wide-range immediate numbers and incorporating built-in functions for vector and matrix computation, EusLisp demonstrates an efficient implementation of a geometric modeler. The inheritance structure of modeling elements is illustrated in **Fig.9**. Each body is approximated by a polyhedron which is defined by a set of planar surfaces (Brep: Boundary representation). EusLisp defines functions for primitive body creation, set operation between bodies, convex-hull generation, computation of the volume and centroid, hidden-line elimination for drawings, and so on.

To show the viability of EusLisp for practical applications like a geometric modeler, the time required for shape composition and hidden-line elimination for the image of the three-dimensional model illustrated in **Fig.10** is compared with a Fortran version modeler, Solver[17] (**Table-4**). We have chosen Solver for the comparison since its source code is available and it is widely used in the robotics research. Almost the same functionality of Solver which is implemented in 20,000 lines of Fortran code, is described in 2,000 lines of EusLisp, developed by only one person in one month. The conciseness depends not only on Lisp's basic features such as automatic garbage collection and symbol and list processing, but also on the geometric data types and functions built into EusLisp.

8. Conclusion

The concept and implementation of Euslisp, based on object-orientation, have been presented. Definition of the built-in data types of Common Lisp as standard classes yields the possibility of incremental extension of system functions through class inheritance, and facilitates the description of basic Lisp functions. High-speed discrimination of tree-structured types and a flexible memory management method, both of which are significant to the efficient implementation of object-oriented Lisp, have been described. As a Lisp, EusLisp demonstrates performance comparable to KCL, and high performance and improved description ability when compared with Fortran in a geometric modeler application. EusLisp is a practical programming system which realizes a subset of Common Lisp facilities and has been used for robotics research at several sites. Currently, implementations for Sun3, Sun4, Vax, News, and Sanyo are completed and are available to the public along with source code.

Acknowledgment

The authors sincerely express their gratitude to Dr. T. Yuba, Director of the Intelligent Systems Division, Electrotechnical Laboratory, Dr. M. Kakikura, Chief of the Autonomous Systems Section, and Dr. K. Takase, Chief of the Intelligent Machine Behavior Section, for their encouragement in this research work. Also, special thanks go to Dr. S. Tomura of the Computer Language Section, to Dr. M. Pasquier, visiting researcher from LIFIA, and to Mr. S. Peters from J.P.L. as well as to all the EusLisp users for their instructive technical suggestions.

References

[1] Steele, G. L. Jr., Common Lisp the Language, Digital-Press, (1984).

[2] Bobrow, D. G., et al., Common Lisp Object System Specification, X3J13 Document 88-002R, (1988).

[3] Keene, S. E., Object-Oriented Programming in Common Lisp, Addison-Wesley, (1988).

[4] Kempf, J., Design for High Performance Dynamic Generic Dispatch in the Common Lisp Object System, Proceedings of CLOS Users and Implementors Workshop, (1988).

[5] Bobrow, D. G. et al., CommonLoops: Merging Lisp and Object-Oriented Programming, Proceedings of OOPSLA'86, (1986).

[6] Bromley, H. and Lamson, R., LISP LORE: A Guide to Programming the Lisp Machine, 2nd Ed., Kluwer Academic Publishers, (1987).

[7] Cranson, B. and Thomas, R., A Simplified Recombination Scheme for the Fibonacci Buddy System, Communication of the ACM, Vol.18, No.6, (1975).

[8] Peterson, J. L. and Norman, T. A., Buddy Systems, Communication of the ACM Vol.20, No.6, (1977).

[9] Gabriel, R. P., Performance and Evaluation of Lisp Systems, MIT Press, (1985).

[10] Steele, G. L. Jr., Data Representations in PDP-10 MacLisp, Proceedings of 1977 MACSYMA Users Conference, NASA Scientific and Technical Information Office, Washington, D. C., July, (1977).

[11] Cohen, J., Garbage Collection of Linked Data Structures, ACM Computing Surveys, Vol.13, No.3, pp.341-367, (1981).

[12] Yuasa,T. and Hagiya, M., Kyoto Common Lisp Report, Teikoku Insatsu, (1985).

[13] Matsui, T. and Inaba, M., EusLisp: an Object-Based Implementation of Lisp and its Application to Solid Modeling, IPSJ, SIGSYM, 89-SYM-50-2, (1989), in Japanese.

[14] Matsui, T., An Object-Based Robot Programming System EUSLISP, Electrotechnical Laboratory Research Memo., ETL-RM-87-06E, (1987).

[15] Matsui, T., Tsukamoto, M. and Ogasawara, T., An Object Oriented Programming Facilities LEO on ETALisp, Bulletin of Electrotechnical Laboratory, Vol.49, No.7, (1985).

[16] Matsui, T. and Tsukamoto, M., An Integrated Robot Teleoperation Method Using Multi-Media Display, Preprint of the Fifth International Symposium on Robotics Research, pp.156-163, Tokyo, (1989).

[17] Koshikawa, T. and Shirai, Y., A 3-D modeler for Vision Research, Proceedings of ICAR'85, (1985).