数理解析研究所講究録
482巻 1983年 1-22

TOWARD THE DESIGN AND IMPLEMENTATION OF OBJECT ORIENTED ARCHITECTURE

Mario TOKORO

Department of E. E., Keio University

Yokohama 223 JAPAN

ABSTRACT

Object oriented programming languages and the computer architecture to support the reliable and efficient execution of programs written in these languages are important issues for providing better programming environment. The main purpose of this paper is to establish the foundation for the design and implementation of object oriented programming languages and object oriented architecture. First, various definitions for object in existing languages and systems are surveyed, and then a new definition of object is introduced. The sketch of a new object oriented programming language whose design is based on the definition is shown. Finally, issues in the design and implementation of an object oriented architecture which directly reflects both the definition of objects and the structure of the programming language is described.

1. INTRODUCTION

Recent requirements for high level programming languages can be summarized as follows:

(1) Easy to read/write programs,

(2) Flexible and yet efficient programming,

(3) Reliable execution, and

(4) Small software life-time cost.

Modular programming plays the most important role in satisfying these requirements, and object orientation, which is the ultimate form of data abstraction, is the paramount notion for achieving modular programming.

Briefly, object oriented languages/systems are languages/systems in which all the objectives of operations are called objects, and the operations permitted to operate on an object are described in or together with that object. Hydra [Wulf 75, Wulf 81], CLU [Liskov 79], and Smalltalk [Goldberg 76, Xerox 81] are systems/languages which adopted the notion of object orientation. Actor [Hewitt 73, Hewitt 77] is a computational model which is closely related to this notion. Although the notion of object orientation has become popular in recent years, detailed definitions for object and related terms differ much from one system to another. Thus, it would be worthwhile to survey difinitions of object and related terms in order to comprehend the reasons why such definitions are employed in their respective systems.

In order to establish the foundation for the design and implementation of object oriented programming languages and object oriented architecture, we propose a model of object and computation on objects. The definitions of object and related terms in this model are also described hereinafter. An object oriented programming language is used both to exemplify how the model is applied to object oriented programming languages and to give a sketch of our new object oriented language which employs this model. This language is featured by the decomposition of types into property and attribute, the notion of multiple representation in a class, and the notion of link of a name to another name.

As the level of abstraction in programming languages becomes higher, architectural support for more efficient and reliable execution of programs becomes indispensable. A few object oriented machines have been proposed [Snyder 79] or implemented [Giloi 78] [Rattner 80]. An architecture which adopts the object oriented model and executes programs written in the new object oriented language efficiently and reliably is outlined. Issues in the design and implementation of the object oriented architecture, including program structure, cache for object memory, the context switch/parameter passing mechanism, variable length operations, and garbage collection, are also described. Rationale for the new object oriented language and the architecture are discussed.

The discussion in this paper is based on the preliminary discussion in [Tokoro 82].

2. A SURVEY OF DEFINITIONS OF OBJECT

In this section, we survey definitions of object and related terms in

some existing systems.

2.1. Object in Hydra

Hydra [Wulf 81] is the kernel of an operating system. Object in Hydra is the analog of a variable in programming languages. An object is the abstraction of a typed cell. It has a value or state. More precisely, an object is defined as a 3-tuple:

(unique name, type, representation)

A unique name is a name that differs from any other object. A type defines the nature of the resource represented by the object in terms of the operations provided for the object. The representation of an object contains its actual information content.

In Hydra, capability is the other important element for its conceptural framework. Capability is the analog of a pointer in programming languages; the main differences are 1) that a capability contains a list of permitted access rights in addition to pointing to an object and 2) that a capability can be manipulated only by the kernel.

Representation may contain capabilities for other objects. That is to say, representation consists of a data part which contains simple variables and a C-list part which is a list of capabilities. Objects are named by path routed from the current Local Name Space (LNS) of a program. Fig. 1 shows objects in Hydra.

Objects in Hydra can be envisaged as the extension of the notion of storage resources in two directions: one is to incorporate type, i.e., the set of operations defined over the object, and the other is to include capabilities in an object. There is no notion of assignment at the object level. Assignement is performed by copying a value (but not an object) into a simple variable in the representation of an object. If we would like to incorporate assignment at the object level, we could ask the kernel to store in a slot of the current LNS the capability that points to an object. The slot is associated with the variable name to which the assignment has taken place. There is no type associated with a slot. Thus, we would do nothing with type checking for assignment at the object level.

2.2. Objects in CLU

There are two basic elements in CLU semantics [Liskov 79], object and variable. Objects are data entities that are created and manipulated by a program. Variables are just the names used in a program to denote objects.

Each object consists of a type and a value. A type defines a set of operations which create and manipulate objects of this type. An object may be created and manipulated only via the operations of its type. The operations of a type are defined by a module called cluster which describes the template (or skeleton) of the internal representation of objects created by that type and the procedures of the operations of that type. An object may refer to other objects or itself.

There are two categories of objects: mutable object and immutable object. A mutable object may change its state by certain operations without changing the identity of the object. Arrays and records are examples of mutable objects. If a mutable object m is shared by objects x and y, then a modification to m made via x will be visible from y. There are also copy operations for mutable objects.

On the other hand, immutable objects do not exhibit time-varying behavior. Examples of immutable objects are integers, booleans, characters, and strings. There are immutable arrays, called sequences, and immutable records called structure. Since immutable objects do not change with time, there is no notion of share or copy for immutable objects.

Invocation, which is the procedure call in CLU, is specified as:

type $ operation (parameters)

There is no subtype or derived type in CLU.

A type is declared for each variable in CLU programs. There are literals for naming the objects (i.e., constants) of built-in types. A variable can have type any to name an object of any type. There is an operation, force, which checks the type of the object named by the variable. CLU also has tagged discriminated union types oneof for an immutable labeled object and variant for a mutable labeled object. The tagcase statement is provided for decomposing oneof and variant objects. In assignment, the object which results from the execution of a right-hand side expression must have the same type as the variable to be assigned. There are no implicit type conversions.

Assignment y := z causes y to denote the object denoted by z (Fig. 2). The object is not copied; after the assignment is performed, the object will be shared by x and y. Assignment does not affect the state of any object. The declaration of a variable specifies the type of the objects which the variable may denote.

2.3. Actor Model

In actor model [Hewitt 73, Hewitt 78, Yonezawa 81], actor is the unified entity of procedures, storage resources, and data. An actor is activated when it receives a message.

A message conveys the requests of operations with or without data, replies to requests or results of operations. Replies and results can be passed to an actor other than the requesting actor.

There are two kinds of actors: pure actors and impure actors. A pure actor is immutable while impure actor is mutable. The simplest impure actor is called cell and accepts two kinds of messages: one is "contents:" to reference its content and the other is "update: <value>" to update the content. Thus, variables in programming languages are implemented by using cell actors. There is no notion of class or type for an actor.

Fig. 3 shows an actor named 3 which accepts a message "+ 4" and returns "7" and a cell actor named x which first accepts "update: 4", then accepts "content:", and returns "4".

2.4. Smalltalk

In Smalltalk [Goldberg 76, Xerox 81], there are two basic elements: object and variable. An object is a package of information and a description of its manipulation. A variable is a name in a program which refers to objects. An operation for an object is designated in terms of a message to the object. A message contains selectors (i.e. the operation names) and parameters.

Class is a module which defines the operations to create an object and the operations to operate on objects created by the class. An object which is created by a class is called its instance. A class is itself an object. A class may contain its own variables (i.e., the state of the class). That is to say, a class is more than a template.

A class inherits methods (i.e. operations) directly from one (and only one) other class. In such a case, the class from which methods are inherited is called its super class. A super class itself may be subsumed under another super class again. There is a special class called OBJECT, which is the ultimate super class of any classes.

Smalltalk's objects are independent of the notion of storage resources. Particularly, there is no explicit declaration for the internal representation of objects.

In Smalltalk, there is no class (type) declaration for variables. Therefore, it is not easy for a compiler to determine classes and methods at compile-time. This impedes fast execution. It may possibly lead to less reliable execution, since no (static nor dynamic) type checking is performed for an assignment. There is research work on type inference at compile-time done without changing the language construct [Suzuki 80] and on the incorporation of explicit type declarations into variable declarationsin order to increase reliability and efficiency of execution [Borning 81].

2.5. Discussion

Programming languages should be independent of the restrictions imposed by the existing machine architecture, especially from the structure and management schemes of storage resources. After we establish the firm foundation as to what high level programming languages should be, performance issues for execution of programs should be discussed along with the design and evaluation of new computer architecture.

We think the notion of objects employed in Smalltalk seems to be very appropriate in the sense that it is independent of storage resources. The notion of class and the inheritance mechanism of Smalltalk also seems to be natural. We also think that the notion of variables in CLU is powerful in the sense that a variable specifies the type of objects which can be denoted by the variable.

In CLU, a mutable object is effectively used when it is shared by objects which communicate through it. Communication, however, can also be achieved by sharing a variable, if such a mechanism is provided. Then, we need not distinguish mutable objects from immutable objects. A mutable object in CLU also contributes to reducing the number of memory claims and reclamation. This will, however, not always be true in our programming language that is proposed later, since objects of a class can vary in size. An efficient memory claim/reclamation scheme should be provided by object oriented memory architecture.

In contrast with CLU, the actor model employs a cell object which functions as a variable. A cell object can receive messages from two or more objects. This is equivalent to sharing in CLU.

3. THE MODEL OF OBJECT AND COMPUTATION

In this section we present the model of object and computation with definitions. Example programs are also shown to explain the model as well as

to sketch our new object oriented language which employs this model.

3.1. Object and Name

Let us define name as an identifier which denotes objects. A name corresponds to a variable in CLU. Assignment is defined as the association of an object to a name.

We have decided to construct our model with object and name. This decision has been made through the following reasoning:

(1) The semantic structure of an information object is decomposed into two disjoint subfunctions, the access/visibility control and the entity which is the subject of operations. Name takes charge of access/visibility control and object takes charge of the entity to be operated on.

(2) History sensitiveness is achieved only by a name. Thus, a state change occurs only when an assignment is executed, and is explicit.

(3) Sharing of an object is specified only through a name. There is no implicit sharing of an object. (An object may be shared in the implementation, but it is not known or manipulated by programmers.) For this purpose, we introduce link which relays a reference to a name to the referencee of the name. Thus, sharing is controlled by name which is the access/visibility control function. Therefore, an object can remain as an entity to be operated on.

An object which has no internal name space functions as an immutable object. An object which has its internal name space may function as a mutable object.

3.2. Reconsideration of Type

In most of the strongly-typed languages such as CLU and ADA [ADA 80], class or type is specified for variable, although type is defined as a set of objects on which a set of operations is defined. To specify a class for a variable implies two purposes: one is to give information to compilers for determining the class of an operation in advance of execution, and the other is to check whether the assignment of an object to a variable is valid. This situation is consistent unless we introduce subtype and/or derived type. When we introduce them, it becomes very difficult to infer the type of an operation from the types of the variables [Feldman 79]. To make the situation worse, there are cases in which programs are correctly executed even though the range of the operation does not cover the domain of a variable

(but all the values associated with the variable). Such cases often occur, as shown in the following:

```
x: integer;   y: even integer;   y := 2 * x;
```

Obviously, what is done first is to perform the multiply operation, and then to perform the assignment of the result to the variable with an appropriate check. The type of the multiply operation is determined as integer, unless the type of constant 2 is explicitly specified as even integer. Even in the case that constant 2 is explicitly specified as even integer, we do not usually define another multiply operator with the range of even integers. Thus, the assignment fails in compilation, though the statement is tautologically correct.

In order to get rid of this situation, there are three alternatives:

(1) to abandon strong typing,

(2) to define all the operations with the domain(s) and type(s) of operand(s) and the range(s) and type(s) of the results, or

(3) to introduce the notions of property and attribute.

The third alternative, which is proposed also by [Feldman 79] and [Williams 79], is most attractive. Property is attached to name and attribute is attached to object. Property is the assertion for a name, which is described in the form of a proposition.

Attribute corresponds to a specified type or class. Each proposition of property should specify at least an attribute. Thus, the class of an operation is determined by the operands' properties at compile-time, and if it cannot, by the operands' attributes at run-time. Property is validated at compile-time if it is possible, or it is translated into object codes to verify when an association of an object to the name occurs at execution-time.

Properties of names are declared as shown in the following example.

```
i:  property {i: integer};
j:  property {j: integer; 0 < j < 100 };
k:  property {k: integer; k mod 2 = 0};
m:  property {m: integer; m > k};
x:  property {x: anyof (real, integer);
      CLASS$class(x) = real & x > 3.5 |
      CLASS$class(x) = integer & x <= 0 };
```

In an assertion proposition, it is permitted to refer to other names and to invoke operations if necessary. CLASS$class(x) is the operation of the most primitive class CLASS which will return the class of x.

In practice, it might be more amenable to programmers to declare property as*:

<name>: <attribute> [in <representation>]
[property <assertion proposition>];

where <assertion proposition> describes more detailed property which cannot be expressed in attribute or representation. Representation is discussed below. If the same proposition is used for many names, it would be beneficial to use compile-time facility to define property texts.

3.3. Reconsideration of Representation

In most abstract data type languages, an operation is strongly related to the representation of values. That is to say, one representation is associated with and used in a class. An operation, however, should simply be the mapping of values to values and be independent of their representation. That is to say, we would like to unify semantically equivalent classes into one class, regardless of the difference of representation.

In order to express a value, we need a bit-string which is just long enough to express the value. Integer numbers of any size should be operated on in a unified manner and the result should be generated in appropriate sizes. Binary integers and decimal integers should automatically be adjusted in the operations. Floating numbers of any size and formats should also be operated on in a unified manner. The orthogonal and polar representation for a complex number should be used interchangeably. The list representation, dope vector representation, and linear representation for an array should also be used interchangeably. In most cases, representation should not be seen by programmers. For primitive representation such as length or radix, computer architecture should provide the facilities for bit-strings interpretation/generation/conversion in accordance with the description of representation of an object. For more complex representation, programming languages should provide a method of describing the interpretation/generation/conversion of representations.

Thus, we introduce multiple representation in a class. That is to say, a class can have multiple representations with their operations definitions,

* A clause surrounded by "[" and "]" may be omitted.

and the transformation rules between representation if possible. An example of a multiple-representation class is shown as follows:

```
class complex is
  create_xy,create_polar,add,sub,mul,div,
  x_cordinate,y_cordinate,abs,angle;
rep structure {r, i: real} as orthogonal;
rep structure {r, theta: real} as polar;

procedure create_xy (x, y: real)
      returns( orthogonal cvt)
  return rep${ r: x, i: y } as orthogonal
end create_xy
     .
     .
procedure add (x: orthogonal cvt, y: cvt)
      returns (orthogonal cvt)
  repcase y of
    orthogonal: return complex$create_xy (x.r
      + y.r, x.i + y.i);
    polar: return complex$create_xy( x.r +
      y.r * real$cos(theta),
      x.i + y.r * real$sin(theta) );
end repcase
end add

procedure add (x: polar cvt, y: cvt)        returns (polar cvt)
     .
     .
end add
     .
     .
end complex
```

Note that we do not need to have different classes for orthogonal and polar complex numbers, although the same effect could be expressed by tagcase for the oneof type in CLU. Representation can neatly describe units, for example, the class for length such as in meter, centimeter, inch, mile, and so forth. Representation is attached to object in addition to attribute.

3.4. On the Visibility and View of Objects

Access to an object should be regulated for the purpose of protection. Capability, as used in Hydra, is one of the methods of regulating access. However, we do not feel secure in using capability, since a given capability is valid forever and is transferable. Thus, we introduce the notion of scope. Scope resembles the access list method of protection [Denning 76] in the sense that all the information for protection is kept in the accessed entity, and the permission to access is determined by the accessed entity.

It should be noted, however, that scope is not specified for an object, but is specified for a name. Scope specifies the visibility of a name with respect to time and usage. Like a virtual circuit in a communication network, permission is given to an accessing entity, but can be revoked at any time. The permission is not transferable to others.

As for usage, there are three different modes: evaluate enable/disable, associate enable/disable, and link enable/disable. If evaluation is enable for a name which denotes an object, then the evaluation of the object returns the object as a whole. If evaluation for a name which denotes a procedure in the object is also enable, then the procedure can be evaluated. Associate enable is equivalent to write enable in many operating systems. In our object oriented programming language, variable must have at least associate enable for the surrounding context (i.e., local variable). Link enable is used to specify that a name can be shared (pointed) by more than one other name.

Scope can specify permission for specific accessing entities. Thus, we can regulate access for each accessing entity at various modes.

It is sometimes strongly demanded that different views [Goldstein 80] be provided for the same object. View is achieved by the use of link and scope.

3.5. The Model

In this section, we will summarize the definitions of object and related terms discussed above.

The relations between an access and the accessed object are classified into the following four hierarchical categories:

Scope: Scope defines the set of permitted accessing methods between an access entity and the name to be accessed. That is to say, scope specifies for a name the authorized accessing entities and access methods. The access methods include evaluate and associate. This corresponds to

capability. Thus, scope is used to regulate access to shared information.

Property: Property defines the relation between a name and the set of objects which can be associated with the name. Thus, property is the assertion of a name, which is validated at each association. Thus, property describes an assertion proposition which is evaluated on association. An assertion proposition may specify property and representation, may refer to other names, and may invoke operations.

Attribute: Attribute defines the relation between an object and the set of operations to be performed on the object. That is to say, attribute corresponds to specifying class, and is used to check the eligibility of an operation to be performed.

Representation: Representation defines the relation between an object and its physical representation. For example, representation specifies radices and sizes for an integer and real number, the orthogonal or polar representation for a complex number, mapping methods for an array, and the length of a string. Representation can be transformed dynamically at the execution-time to meet the operation to be performed*.

Thus, object, class, and name are formally presented as follows:

Object: Object is a package of information and the description of its manipulation. An object is represented as a 3-tuple:

(<attribute>,<representation>,<bits>)

An object can be envisaged as an environment (or context), where <attribute> specifies a class and <representation> and <bits> are local data in the environment.

Class: Class is a module which defines the operations to create an object and the operations to operate on objects created by the class. An object which is created by a class is called its instance. A class is itself an object. A class may contain its own variables.

Name: Name is an identifier that denotes an object (i.e., the current object or nil). A name is represented by a quartuple:

(<identifier>,<scope>,
<property>,<object pointer>)

* and the property of name to be associated, if necessary.

Objects can be referenced only through names.

3.6. Operations

We define three basic operations for the interpretation of programs, association, evaluation, and link. Association, which is used to assign a value to a variable, performs the association of a name with an object. This basic operation is expressed in the following form:

assoc (<name>,<object>)

Association frees the old object that has been associated with a name from the name and then associates a new object with the name. In advance to this association, the eligibility of the access right of association is checked first according to the scope of the name, and then the eligibility of association of the object to the name is checked according to the attribute (and representation, if necessary,) of the object and property of the name.

The evaluation of a name or operation on objects returns an object. Evaluation is expressed in one of the following forms:

```
eval (<name>)
eval (<class name>$<procedure name>
                    <parameter list>)
```

where <parameter list> represents the list of names. The eligibility of evaluation of the name or operation is checked in advance according to the scope of the name or operation. Evaluation of <name> simply returns the object denoted by the name. Evaluation of a procedure is performed as follows: after the necessary checking is performed, the context of the <procedure name> of the <class name> is created; after the parameters are passed, the procedure body is executed. The returned object has attribute and representation, but can have neither property nor scope until it is associated with name. Object is, therefore, defined as the entity as a whole that is associated with name.

Thus, for example, the statement z := x + y is compiled as either of the followings:

```
assoc( z, eval( number$add( x, y ) ) )
```

which is the object program when the compiler knows that the class of x is type number, or

assoc(z, eval(CLASS$class(x)$add(x, y)))

when the compiler does not know the class of x.

Link makes a link from one name to another. This basic operation is expressed in the following form:

link(<from name>, <to name>)

Assume this operation has been performed. Then, the evaluation of <from name> keeps performing this operation along with this link until it reaches the ultimate object. Checking of the eligibility of access is done at every name it hauls. Association of an object to <from name> associates the object to the ultimate name of <from name>.

link(<name>, undefined)

unlinks <name> and denotes undefined.

4. THE OBJECT ORIENTED ARCHITECTURE

4.1. Proposal

In order to perform a computation, we need a program and information. Thus, the internal structure of a computer can be envisaged as the pair of an operational unit and an informational unit. The operational unit controls and executes operations, while the informational unit preserves and supplies information. In conventional architecture, the operational unit and the informational unit corresponds to CPU and the memory, respectively. The interface between CPU and the memory is performed in terms of the physical memory address. The memory returns a string of bits to the CPU. In the object-oriented architecture proposed here, names are used for this interface. The informational unit returns objects.

The names, which interface the operational unit and the informational unit in the object oriented architecture, are similar to the names in the capability based architecture [England 72] [Myers 81] [Houdek 81] [IBM 80] in the sense that they are unique in the system. In order to make distinction between the local information environment of a program segment and the global information environment of the system, the object oriented architecture provides the local name space and the global name space. Fig. 6 shows the conceptual level description of a procedure being executed on the object oriented architecture.

During the evaluation of a class procedure of an instance, the instance

logically contains the procedure object. Thus, even though different instances belong to the same class, these instances can be evaluated (executed) simultaneously. However, multiple procedures of an instance cannot be evaluated simultaneously.

The functions of the informational unit are to regulate all the access to objects as well as preserve objects. The informational unit supports the virtual memory system. The informational unit is also in charge of doing garbage collection.

The four principal functions of the operational unit are the following: 1) to check whether or not the attributes of objects are eligible for an operation to be performed, 2) to find an appropriate resolution for the representation of objects, 3) to perform the operation on the objects, 4) to request the informational unit to access/preserve the object with a name. The operational unit also controls the sequencing of instructions and process activities.

4.2. Issues

Issues in the design and implementation of the object oriented architecture are described here.

(1) Attributes

An attribute can be either primitive or structured. Examples of primitive attributes are boolean, character, integer, fraction, real, and complex. A structure object supplies a space for names and has an internal addressing mechanism, so that an element of the structure object is accessed via an element name. An element of a structure object may be a structure or a primitive object. Some typical structure attributes are array, string, record, stack, and list. Nil is defined here as the value of an object whose representation is not defined. Undefined is defined as the value which indicates that the object name is not associated with any object.

(2) Program Representation and Procedure Invocation

A procedure object is represented by a tree of segments, each of which represents a block in a block level. There are no branch instructions that specify memory addresses. Control branchs to the top of a segment and returns from the bottom of the segment without exception, and thus structured programming is attained at the architectural level.

A procedure is invoked by evaluating the name of the procedure with actual parameters. Thus a procedure invocation can be envisaged as the

passing of a message to the procedure object. The scope of the name of the procedure is checked as to whether it is executable (evaluate enable to the caller) or not. If it is executable, the eligibility of the association of the actual parameters with the formal parameter names is checked according to the scope and property of each formal parameter name. Then, the local name table is created in accordance with the procedure template, and the actual parameters are associated with the formal parameters. The local name table is freed when the execution of the procedure is completed. Therefore, a procedure object cannot be history sensitive.

(3) Name Table Structure

As seen in commercialized object oriented computers such as Intel 432 [Rattner 80] [Intel 81] and IBM System 38 [Houdek 81] [IBM 80], fast address transformation via a chain of tables is the most important issue for higher performance. Fast hashing hardware such as [Goto 77] and enough table lookaside buffers are indispensable.

Placing a simple object to the pointer part of an entry of address transformation tables should also be considered. In our object oriented architecture, basic operations are performed in variable size fashion. Therefore, we place into the pointer part a simple object whose length is less than or equal to that of the pointer part. If the length of an object is larger, the object is created in the heap area of the system. Since the length of most of the primitive objects is less than the length of the object pointer, this strategy will result in a fast execution speed with variable length operations.

(4) Maximal Utilization of Cache

Since it is expected that the instruction and data access patterns differ greatly from each other in the object oriented environment, to provide separate caches for instruction and data sounds reasonable and effective. The instruction cache would be similar to those which are used in existing machines. However, the data cache would be peculiar to the object oriented architecture.

The data cache should be designed to provide for the very fast creation of objects. The object oriented architecture would try first to create an object on the data cache. If the cache has no free space, one or more objects which have copies in main memory and/or which are expected not to be referenced in the near future are rolled out. Thus, the cache acts as a fast local memory. The cache does not have to be a partial copy of the main

memory addresss space, but can provide a separate address space. A separate garbage collection (possibly with a different scheme) from main memory might be effective for the data cache.

(5) Fast Context Switching and Parameter Passing

It is well known that a procedure call instruction appears on the aberage every 10 instructions in the object codes of a high level language program. In object oriented language programs, the frequency of the apparance is expected to be higher. Thus, fast context switching/parameter passing mechanisms are of particular significance. Since our object oriented architecture is not a general register machine, fast context switching/parameter passing mechanisms by optimized general register usage such as those employed in IBM 801 [Radin 82] or RISC [Patterson 81] cannot be employed. Object oriented context switching/parameter passing mechanisms are being investigated. The use of multiple stacks with stack caches have been of special interest.

(6) Variable Length Operations

One of the purposes of our object oriented architecture is to provide variable length operations by hardware. Variable size operations for binary and decimal integers, boolean, and character strings will be implemented by hardware. Variable size floating-point operations by means of recurring rationals [Yoshida 82] are being considered for hardware implementation.

(7) Support for Dynamic Checking

We have decided on the policy that in order to generate efficient object codes the compiler checks the eligibility of access and operation as much as it can at compile-time, and leaves what it cannot check at compile-time for dynamic checking at execution time. We employ tags [Iliffe 68] [Feustel 72] for frequently used properties, attributes, and representations. Thus, we need high speed tag manipulation hardware.

(8) Garbage Collection

In object oriented architecture efficient garbage collection schemes must be adopted. This is especially important in our architecture, since objects are not only dynamically created and freed but also the sizes of the bit strings vary in concert with their values. The sizes are indicated in their representation specification. Thus, we might have to incorporate compaction as well as garbage collection.

We employ the combination of the reference counter method and the

marking method for garbage collection. In most cases, the reference counter method is effective and efficient. For the cases in which links make a cycle and in which a reference counter overflows, the marking method is effective. The employment of a garbage collection processor is being considered in the informational unit. It will run parallel with the other part of the informational unit.

5. CONCLUSION

The object oriented language described in this paper provides us with powerful and yet flexible descriptivity. The decomposition of types into property and attribute, the notion of multiple representation in a class, and the notion of link of one name to another are most novel features of the language.

The object oriented architecture that directly supports this language brings us the following advantages:

(1) Information can be independent of programs and programming languages. This independence enables sharing of information among programs written in different programming languages.

(2) Programs become independent of the types, structures, and representation of data. This enables a procedure to be used with parameters of various properties as long as the algorithm is the same.

(3) Increased software reliability and debugging capability are achieved by the dynamic checking of scope, property, and attributes.

The issues in the design and implementation of this architecture are described for the rialization of versatile, efficient, and fast VLSI object oriented computers. Since the VLSI technology has already brought us more than a million transistors on a chip, author believes that we will have the object oriented architecture in a microcomputer in the very near future.

The final decision for the design of the object oriented language is being made by the object oriented programming/processing system (OOPS!) group at Keio University. The design of the architecture is also being performed along with performance evaluation.

ACKNOWLEDGEMENT

The author is grateful to Professor E. I. Organic of the University of Utah, Dr. J. R. Hamstra of Sperry Univac, and Dr. Dennis Frairy of Texas

Instrument for their invaluable comments. The author is also thankful to Mr. Takashi Takizuka of KDD and Miss Kaoru Yoshida and Mr. Yutaka Ishikawa of Keio University for their earnest and keen discussions.

REFERENCES

[IBM 80] "IBM System/38 Technical Developments (G58060237)," IBM, 1980.

[ADA 80] Reference Manual for the Ada Programming Language, United States Department of Defense, 1980.

[Borning 81] A.H. Borning and D.H.H. Ingalls, "A Type Declaration and Inference System for Smalltalk," Proc. of Principle of Programming Languages, December 1981.

[Denning 76] P.J. Denning, "Fault Tolerant Operating Systems," Computing Surveys, Vol. 8, No. 4, pp.359-389, 1976.

[England 72] D.M. England, "Architectural Feature of System 250," Infotech State of the Art Report 14: Operating Systems, Berkshire, Infotech, England, pp.395-428, 1972.

[Feldman 79] J.A. Feldman, "High Level Programming for Distributed Computing," Communications of ACM, Vol.22, No.6, pp.353-368, 1979.

[Feustel 72] E.A. Feustel, "The Rice Research Computer - A tagged Architecture," AFIPS SJCC, pp.369-377, 1972.

[Giloi 78] W.K. Giloi and H.K. Berg, "Data Structure Architecture - A Major Operational Principle", Proc. of 5th Annual Symposium on Computer Architecture, 1978.

[Goldberg 76] A. Goldberg and A. Kay, ed., "Smalltalk-72 Instruction Manual," Xerox Palo Alto Research Center, March 1976.

[Goldstein 81] Goldstein, Ira and Bobrow, Daniel, "An Experimental Description-Based Programming Environment: Four Reports," Xerox PARC Research Report, CSL-81-3, March 1981.

[Goto 77] Goto, E, Ida, T, and Gunji, T, "Parallel Hashing Algorithms," Information Processing Letters, Vol. 6, No. 1, 1977.

[Hewitt 73] Hewitt, C., et al., "A Universal Modular Actor Formalism for Artificial Intelligence," Proc. of IJCAI, pp.235-245, 1973.

[Hewitt 77] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," J. of Artificial Intelligence, Vol. 8, 1977.

[Houdek 81] Houdek, M.E, Soltis, F.G., and Hoffman, R.L, "IBM System/38 Support for Capability-Based Addressing," Proc. of the 8th Int'l Symp. on Computer Architecture, pp.341-348, May 1981.

[Iliffe 68] J.K. Iliffe, "Basic Machine Principles," American Elsevier, New York, 1968.

[Intel 81] "Intel APX432 General Data Processor Architecture Reference

Mannual," Intel, Aloha, Orengon, 1981.

[Liskov 79] B. Liskov, et al., "CLU Reference Manual," TR-225, Laboratory for Computer Science, MIT, Oct. 1979.

[Patterson 81] Patterson, D.A., and Sequin, C.H., "RISC-I: A Reduced Instruction Set VLSI Computer," Proc. of the 8th Int'l Symp. on Computer Architecture, pp.443-457, May 1981.

[Radin 82] Radin, G, "The 801 Minicomputer," Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems, pp.39-47, March 1982.

[Rattner 80] J. Rattner and G. Cox, "Object-Based Computer Architecture," Computer Architecture News, Vol. 8, No. 6, 1980.

[Snyder 79] A. Snyder, "A Machine Architecture to Support an Object-Oriented Language," MIT TR-209, Laboratory for Computer Science, MIT, Marach 1979.

[Suzuki 80] Suzuki, N., "Inferring Types in Smalltalk," Proc. of Principle of Programming Languages, December 1980.

[Tokoro 82] Tokoro, M and Takizuka, T., "On the Semantic Structure of Information --- A Proposal of the Abstract Storage Architecture," Proc. of the 9th Int'l Symp. on Computer Architecture, pp.211-217, April 1982.

[Williams 79] Williams, G., "Program Checking", Sigplan Notice, Vol. 14, No. 9, pp.13-25, 1979.

[Wulf 75] Wulf, W.A., et al., "Overview of the Hydra Operating System," Proc. of the 5th Symposium on Operating System Principles, pp.122-131, Nov. 1975.

[Wulf 81] W.A. Wulf, R. Levin, and S.P. Harbison, "HYDRA/C.mmp: An Experimental Computer System," McGraw-Hill, New York, 1981.

[Xerox 81] Xerox Learing Research Group, "The Smalltalk-80 System," Special Issue on Smalltalk-80 System, Byte, Vol. 6, No. 8, August 1981.

[Yonezawa 81] Yonezawa, A., "Unified model of Algorithm Representation," in "Algorithm Representation," Iwanami Book Co., (in Japanese,) 1981.

[Yoshida 82] Yoshida, K., "A Research on Error Free Machines -- A proposal of New Floating-Point Arithmetic based on Recurring Rationals," Computer Science Report, Arithmetic-82-1, Department of E. E., Keio University, March 1982.

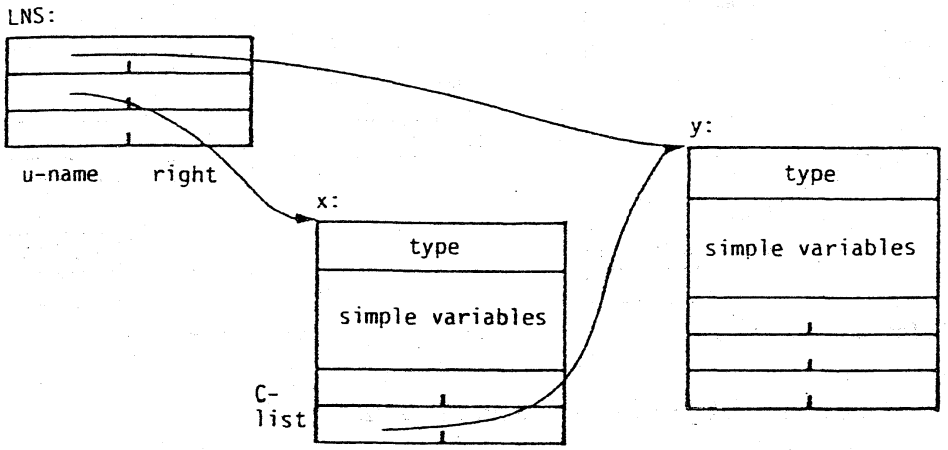

Fig. 1 Objects in HYDRA

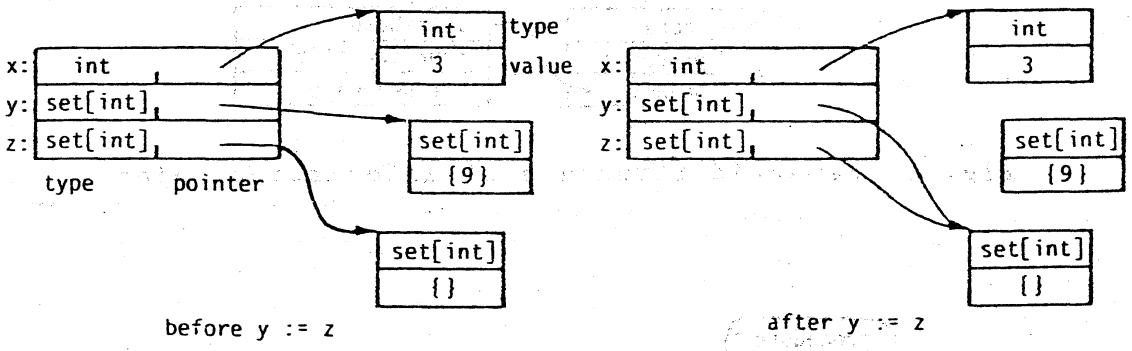

Fig. 2 Objects in CLU

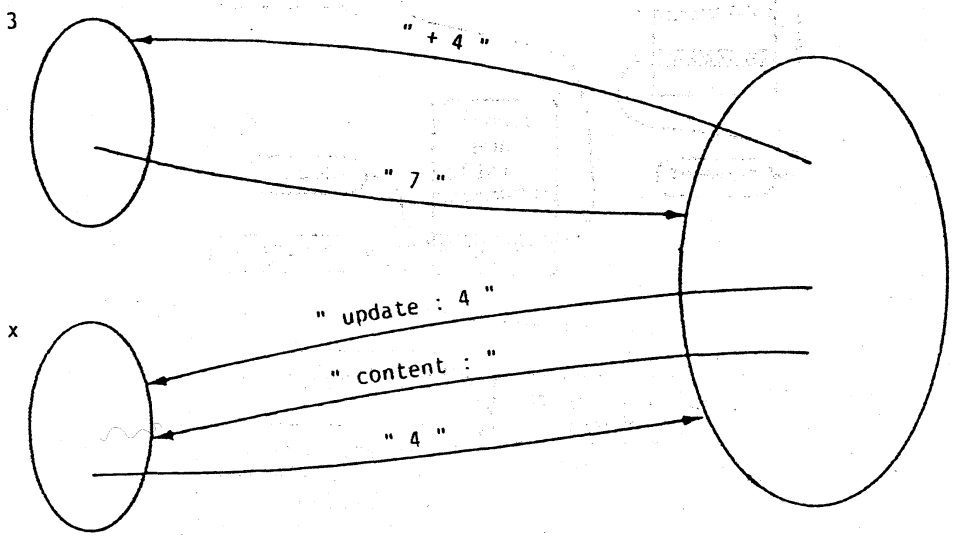

Fig. 3 Actors

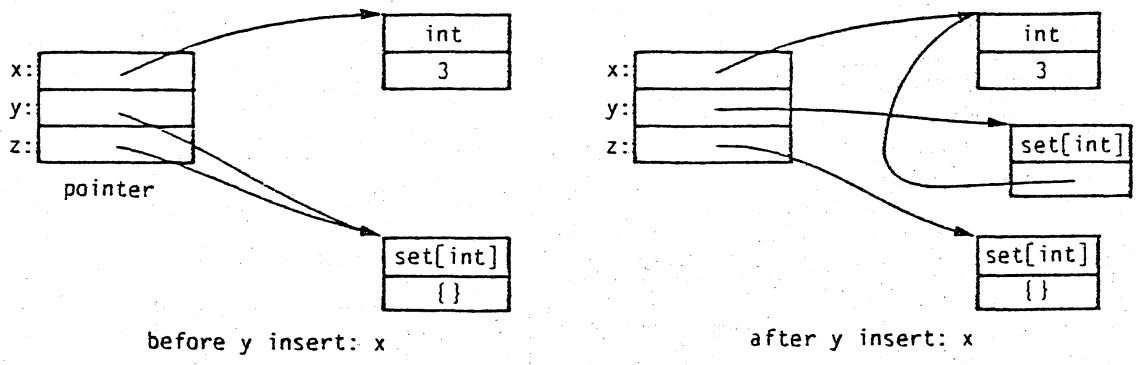

Fig. 4 Objects in Smalltalk

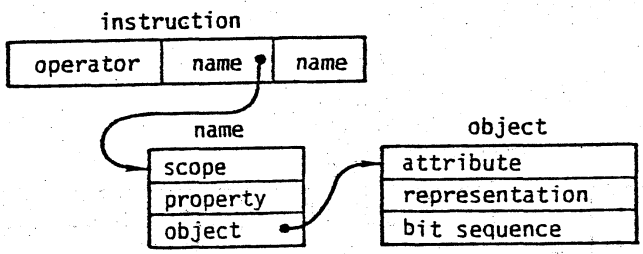

Fig. 5 Semantic Structure of Information Object

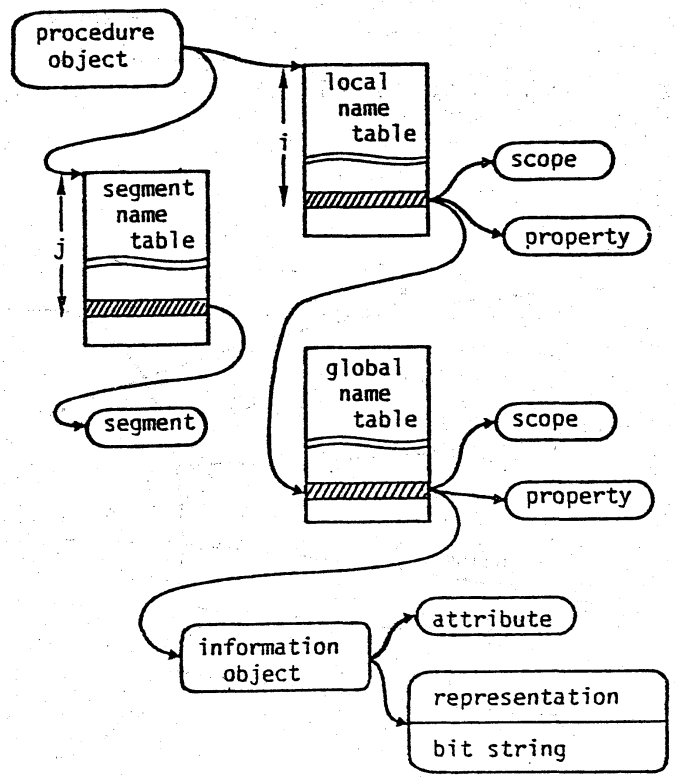

Fig. 6 Run-Time Environment