ソフトウェア基礎論　３３－４
プログラミング言語　２３－４
（1989. 12. 8）

並列オブジェクト指向Lisp - POLの設計について

On the design of a Parallel Object-oriented Lisp ---POL

竹内彰一*1　柴山悦哉*2　安村通晃*3　高田敏弘*4　佐治信之*5
Akikazu Takeuchi Etsuya Shibayama Michiaki Yasumura Toshihiro Takada Nobuyuki Saji

*1三菱電機　*2東工大　*3日立製作所　*4NTT　*5日本電気
Mitsubishi Electric Corp. Tokyo Inst. of Tech. Hitachi Ltd. NTT NEC Corp.

あらまし　並列オブジェクト指向言語POL (Parallel Object-oriented Lisp) の設計について中間報告する。POLは、ISLisp(国際標準Lisp)の長期的課題に応えるものとして、情報処理学会SC22/Lisp-WG AdHoc 2の活動として設計を行っているものである。POLは並列オブジェクトとその間のメッセージ交換により並列/分散プログラムを記述するためのLispシステムである。

Abstract　Intermediate status of design of the language POL (Parallel Object-oriented Lisp) which incorporates features we consider desirable in the long range target of IS (International Standard) Lisp is reported. POL is a system on Lisp for describing parallel/distributed programs in terms of concurrent objects and message passing among them. This work is done as an activity of Japanese SC22/Lisp Wg AdHoc-2.

1 Introduction

Lisp has been an up-to-date professional AI language (or a family of languages) for many years. In order to hold this status, however, Lisp must be enriched to cope with new computational environments particularly in the following areas:

Network computing, distributed processing, concurrent computing, multi-media databases, computer graphics, animation systems, and so on.

For the purpose, we consider that future Lisp systems should support mechanisms for:

1. controlling parallel and distributed execution;
2. guaranteeing realtimeness;
3. modeling complex entities in a sophisticated manner.

Parallel Object-oriented Lisp (POL), which is a happy marriage of Lisp and parallel object-oriented computation model, is our current solution to a part of these requirements. With the notion of concurrent objects, POL supports mechanisms for description of distributed computation and those for natural modeling of entities in a wide variety of problem domains.

2 Design Issues of POL

In design of a Lisp language for distributed computing, we have to make several decisions concerning at least the following:

1. degree and granularity of parallelism
2. basic communication and synchronization mechanisms
3. code sharing mechanisms
4. persistence of data objects
5. compatibility with existing sequential Lisp systems

Note that appropriate solutions may depend not only on application areas but also on target machine architectures. This means that we have to figure out future computing environments and application areas of Lisp somehow.

2.1 Models for Underlying Architectures

One reason of the difficulties in design of a parallel language standard is that we do not have any single

standard parallel architecture. Instead, we have a wide range of parallel machines: SIMD machines and MIMD machines, shared memory machines and distributed memory machines, ethernet connected workstations and massive parallel multi-computers. This is not the case of sequential language design, in which we can assume the Von Neumann style architecture without any argument.

Of course, the details of underlying architectures are not our primary concern. Programming languages should provide more abstract views and conceal the architectural details. But still natures of underlying architectures often restrict high level abstractions since, for instance, with a SIMD machine we cannot expect efficient execution of MIMD style parallel programs.

Although universalities of programming languages are important and desirable (particularly when the languages are candidates for the international standard), we do not consider it feasible to design a language which is suitable for all sorts of parallel machine models. We have to choose appropriate range of target architecture models.

SIMD models versus MIMD models:

In our application problem domains (e.g., Distributed Artificial Intelligence), we need concurrent processes each of which does its own distinct work and which, as a whole, cooperate and solve a given problem. Since this sort of parallel computation is hard to be described on top of any SIMD model, we think that MIMD models are more appropriate for our purposes. Fortunately, in near future, various kinds of MIMD machines seem to be available.

Note that, if one's requirement were to support data parallelism (such as the one in Connection Machine Lisp) alone, a massive parallel SIMD model might be promising.

Single Address Space versus Multiple Address Space:

Although on a shared memory multi-processor system, we can easily implement a single address space language, it is hard (or at least inefficient) to maintain a single universal address space on a loosely coupled multi-computer system. It is obvious that shared memory models are more expressive, while a shared memory system with a large number of processors seems infeasible.

We therefore should cope with multiple address space. Fortunately, Lisp does not depend so much on a mechanism mapping addresses to their contents. Instead, what we need is a mechanism which chases references. With an appropriate abstraction mechanism such as the one provided by the notion of parallel object-orientation, the number of remote references can be kept small (at lease much smaller than the number of the addresses) and a reasonable implementation on loosely coupled multi-computer systems may become possible. Anyhow, a large number of concurrently accessible global addresses will cause trouble in programming, and thus we need a good abstraction mechanism.

Fine-grained Computers versus Coarse-grained Computers:

In fact, a wide variety of parallel computers, which support fine, medium, and coarse grained parallelism are available (or will be available near future). Currently, many organizations have work-station systems connected via local area networks and they support coarse grained parallel computing, in which processes (or threads) provided by operating systems are units of parallelism. Massive parallel multi-computers based on message passing will be available in near future. Since both styles can be popular and widely used, our target Lisp language should support a wide range of granularities (or it is preferable to support variable grained parallelism).

The framework of parallel object-orientation can be applicable to any grain sizes1)[1]. However, we are wondering whether it is possible for programmers to effectively control fine grained parallelism with a reasonable amount of efforts. Perhaps, we should consider a way to support variable grained parallelism, i.e., a way to describe programs which can be executed efficiently both on network-connected work-stations and massive parallel multi-computers.

[1]One problem is that, to cope with fine grained parallelism, the dispatch mechanism of arriving messages must be simple and light.

According to the observation above, we have designed POL so that it is suitable for MIMD, multiple address space with a wide range of grain sizes. Note that, however, shared memory machines are acceptable since, in general, any parallel languages which have (or can have) implementations on top of distributed memory machines can be implemented more easily on shared memory machines.

2.2 Basic Synchronization Mechanisms

Until now, a lot of mechanisms for synchronization and communications are proposed and employed by parallel computing models and languages such as MultiLisp [Hals84], Qlisp [Gabr84], CSP [Hoar78], CCS [Miln80], Actor [Agha87], Concurrent Prolog [Shap83], and Linda [Carr89]. Multi-Lisp and Qlisp are parallel Lisps developed on top of shared memory machines. Without drastic modifications of their language semantics, it is difficult to transport them on a distributed computing environment. However, the "future" mechanism originated by Multilisp is notable.

CSP, CCS and the Actor model are well-suited for distributed memory models. Synchronization and communications of CSP and CCS are based on synchronous operations and those of the Actor model is based on asynchronous message passing. Both synchronous and asynchronous communications have advantages and disadvantages: with asynchronous communications we may exploit more parallelism, while possibly distorting program structures. As a result, POL supports both synchronous and asynchronous communications primitives (even though we consider that asynchronous models, which are more effective but dangerous, are a good bet). Note that the "future" mechanism might support safer asynchronous interactions.

In Concurrent Prolog and other parallel logic programming languages including PARLOG [Clar86] and GHC [Ueda85], logical variables are the means for process interactions. We consider that logical variables are useful and attracting, but we have not yet successfully integrated them into the language POL. One more interesting point is that these languages can, in some degree, support variable grained parallelism because the application orders of the unification operation, which is the most fundamental one, are not significant, i.e., it is commutative.

Linda is the name of a family of languages which support the tuple-space communication mechanism. We do not want this mechanism to be the only synchronization/communication primitive, but with introduction of the notion of hierarchical decomposition we may accept it as "one-of-them."

2.3 Inheritance and Delegation

When building a large scale software, we need a sophisticated code sharing mechanism. Currently, for the purposes we have two alternatives, namely, inheritance and delegation. Inheritance is a popular mechanism and supported by most object-oriented languages including Smalltalk-80 [Gold83], Flavors [Flav86], CLOS [CLOS88], and C++ [Stro86]. Even though there are criticisms, we agree that an inheritance mechanism has been a necessary aid for building a large program. In contrast, an explicit language support for delegation is rare: SELF [Cham89] is one exception. We do not know any concrete example in which a delegation mechanism helps to build a large scale software, but we consider that a delegation mechanism has similar advantages.

Indeed, the definitions of "inheritance" and "delegation" vary from language to language. In order to continue the discussion, now, we present our views of them focusing their differences. Table 1 shows four dimensions.

	Inheritance	Delegation
Parents	Statically determined	Dynamically determined
Shared entities	Methods of the parents	parents themselves
Copied entities	Instance variables of the parents	
User's view	Implementations are inherited	Behaviors are inherited

Table 1: Inheritance and Delegation

Static Parents and Dynamic Parents:

From the table, it is obvious that a delegation mechanism is suitable for the situation in which the child-parent relationship may be modified dynamically during execution. In contrast, if the relationship is statically determined, inheritance may be more efficient.

Sharing and Copying:

With a delegation mechanism, more than one object can share the same parent object. Thus, this mechanism seems a means to control the degree of sharing. Note that in a distributed environment it is very important to control the degree of sharing explicitly. On the other hand, owing to concurrent accesses to a parent object, we need some concurrency control mechanism for the object. When a delegation mechanism forwards messages in an implicit manner, serious problems may occur.

In case of inheritance, no problem concerning shared objects may occur since what an inheritance mechanism can do at best is to share immutable methods [2].

If class variables (in Smalltalk-80) or shared slots (in CLOS) were allowed to be inherited, a serious problem might occur. Inheritance does not provide a way to share objects during execution but to share source codes prior to execution.

Inheriting Implementation and Behavior:

In design of the hierarchy among concurrent objects, which model entities in a given problem domain, we probably classify the objects based on externally observable "similarity."On the other hand, when implementing these objects, we probably require a good mechanism for reuse of implementation details.

From this observation, we consider that we need two sorts of classification hierarchies: one for external behaviors and the other for internal representations. Delegation and inheritance seem to play these two roles, respectively, since an inheritance mechanism reuses the (details of) internal structures of parent classes, whereas a delegation mechanism just reuses the external behaviors of parent objects.

Consequently, inheritance and delegation are not necessarily competitors but we think that their cooperation will grow the modeling and expressive powers of programming languages. Currently we think delegation is harder to be used in distributed environment though it has more attractive characteristics.

3 Objects

POL has three kinds of objects, active, monitored and unmonitored objects. Monitored and unmonitored objects are called passive objects. An active object has its own local state, script, message queues and single execution thread. Such an object receives messages and processes them one by one in a sequential manner. It is a unit of parallelism. In contrast, a passive object has no thread. An unmonitored object is a private datum of just one active object, the owner of the passive object. When an active object sends unmmonitored objects to other active objects, what is actually sent must be their copies. The notion of a unmonitored object corresponds to the notion of objects in sequential object system such as CLOS and Flavors. In contrast with unmonitored objects, monitored objects can be accessed by any active object. However their accesses are controlled in mutually exclusive manner, that is, only one method call can be processed at one time.

POL is basically a classless language. An object can be defined and created by a **defobject** form. However, for convenience, a **defclass** form is also provided, and its instances are created by a **make-instance** form. The syntax of *defobject* and *defclass* forms is shown below:

(**defclass** *class-name* (*super-class*...)
  ((*queue-name option*...)...)
  ((*instance-variable option*...))
  option...)

[2]If class variables (in Smalltalk-80) or shared slots (in CLOS) were allowed to be inherited, a serious problem might occur. Even without inheritance, extensive use of class variables or writable shared slots would cause trouble on a distributed environment.

(**defobject** *object-name*
((*queue-name option...*)...)
((*instance-variable option...*)...)
option...)

The syntax and semantics of **defobject** and **defclass** forms are similar to those of CLOS [CLOS88]. However the following difference should be noted:

defobject	A **defobject** form is a short hand for defining a unique active object.
Multiple queues	An active object can have more than one message queue. This will be explained in detail below.
No class slot	Note that no classes can contain any shared slot since a shared slot cannot be implemented in any reasonably efficient manner on a distributed computer system at least under the today's technology.

POL has three important built-in classes **active-object**, **monitored object** and **unmonitored object**. Except for objects defined by a **defobject** form, each active object is an instance of a class which inherits the built-in class **active-object**. Each monitored object is an instance of a class which inherits **monitored object**. Such generic methods shared by all kinds of objects as *copy* which makes a copy of the target object are also being considered.

An active object can have multiple message queues. Each message queue is given a name by the **queue** declaration. If there is no queue declaration, one queue is allocated and has the same name as the object name. Although the total number of queues is determined statically by the definition, the actual queue entities can change during the computation.

A *queue* is the first class data object. A queue is a monitored object keeping a sequence of objects together with at least following operations:

(**send** *Queue Object*)	putting *Object* to the end of *Queue*
(**get** *Queue*)	getting an object from the head of *Queue*
(**append** *Queue-1 Queue-2*)	appending two queues, *Queue-1* and *Queue-2*

A monitored object which satisfies the minimum requirement above is called *a primary queue*. We can define several kinds of message queues, for example a queue handling normal messages and express messages, using the inheritance mechanism as usual. Message queues associated with an active object must be of this class or a sub-class of this class, and are shared by the owner (the receiver) and sender of messages, while their accesses to a queue is controlled in mutually exclusive manner.

Each queue has the state indicating that it is either owned by an active object (precisely speaking it is acting as a message queue of an active object) or free. If a queue is owned by an active object, operations to the queue are restricted depending on the executor. For example, objects except the owner can not get while the owner can do every operation (*Capability*). There are two basic operations which establish and cancel the ownership relation between an active object and a queue.

(**own** *Queue* {*QueueName*})	An object calling this method makes *Queue* its message queue. *QueueName* is optional. If it is omitted the name *primary* is used. *QueueName* is needed when an object has more than one queue.
(**release**{*QueueName*})	The queue specified by *QueueName* is released. The returned value is the identifier of the queue.

A queue owned by an object *O* with the queue name *Q* can be referred to as the following form: *O.Q*. When an object refers to its own queue with the name *Q*, the form, *self.Q*, can be used.

Multiple-queue has three advantages. First it distributes accesses to one message queue over multiple queues and reduce access traffic to each message queue. The effect will be significant since message queues are realized as monitored objects to which *read* and *write* accesses are controlled in a mutually exclusive manner. Secondly it enables partial reorganization of message queues. Since a

queue can be manipulated independently we can, for example, replace part of queues by new queues. The third advantage is related to description of behavior of an object. Multiple-queue is a way to separate messages from different objects. In other words, it can prevent messages sent from different objects from being merged into one queue. Therefore by assigning a distinct queue to an object with a special role we can write some behavior elegantly. For instance, suppose an object *O* has a superviser *S* which sends control messages such as *suspend*, *resume* and *terminate* to *O*. These messages control processing of messages sent from an object *O'*. The behavior of *O* can be easily described by defining two queues *Q1* and *Q2*, *Q1* from *O'* and *Q2* from *S*. The description of *O*'s behavior is:

If there is a message in Q2 then process it. Otherwise process a message in Q1.

Note that the third advantage is not a unique feature of the multiple-queue. In fact, the same effect can be obtained even in the single queue case by defining a queue with multiple local queues. Such a queue has a keyword index of local queues and read/write access to a queue must be associated with a key in order to be interpreted as read/write access to a local queue with the identical key.

4 Methods

In POL, the behavior of objects is defined in **defmethod** and **defaction** forms:

(**defmethod** (*method-name object-name* {qualifier})
  (*formal-parameter...*
    {**:reply-to** *formal-parameter*}
    {**:from** *formal-parameter*}
    {**:constraint** *expression*})
  form...)
(**defaction** (*object-name* {*qualifier*})
  (*formal-parameter...*
    {**:reply-to** *formal-parameter*}
    {**:from** *formal-parameter*}
    {**:constraint** *expression*})
  form...)

Defmethod is used for passive objects and active objects with single queue. **Defaction** is used only for active objects with more than one queue. The syntax and semantics of **defmethod** are similar to those of new Flavors [Flav86] rather than CLOS since POL does not support multi-methods. The **defmethod** form defines a method named *method-name* for an object specified by *object-name* if it is a classless object, or for instances of the class specified by *object-name*. **Defaction** has the similar semantics except that it does not have *method-name*. When *object-name* inherits active-object, these forms can have an optional formal parameter following the keyword **:reply-to**. Upon invocation of a method or an action (i.e. upon message acceptance), this *formal-parameter* is bound to the *reply-destination* (the destination to which the reply of a message should be sent back) of the accepted message. Method combinations and action combinations are being considered. Hereafter invocation of a method and that of an action are called *method call* and *action call*, respectively.

A method call whose first argument is a passive object is evaluated as in new Flavors. Whereas, a method call whose first argument is an active object is interpreted as a message passing form. Assume, for instance, that some active object, say O, executes the following method call:

(*method-name O' arguments ...*)

where *O'* is an active object. *O* sends to *O'* a message which contains method-name, arguments, and some other extra information such as the reply destination. This message is asynchronously received by *O'* and first put into its message queue. By the message passing, the current execution of *O* is not blocked and *O* continues the execution. Just when *O* becomes necessary to get the evaluation value of the method call, *O* is blocked until the reply to the message passing is arrived. In parallel, *O'* processes messages in its message queue one by one and will eventually encounter the message sent by *O* (if *O'* neither enters infinite loop nor signals an error) and, at that time, *O'* invokes its method specified by method-name with the arguments enclosed in the message. In the body of the method, the reply to the message may possibly be sent back by a reply form:

(**reply** *form*) or (**reply-to** *form reply-destination*)

where the evaluation value of the form can be multiple values. Notice that, in the body of the

method, the reply destination to the message can be forwarded to another active object, which is, in this case, responsible to send back the reply. Upon evaluation of a method call whose first argument is an active object, a future structure is automatically generated, by default, and specified as the default reply destination. POL will support forms which explicitly generate future structures and those which explicitly specify reply destinations of message passing.

Formal parameter part in **defaction** specifies message patterns of multiple queues where a queue name is used as a keyword and is followed by message pattern. A sequence of a pair of a queue name and message pattern is translated into a sequence of **get** and comparison procedures by the underlying mechanism. An active object with multiple queues checks actions in nondeterministic way. When an object receives a message, it searches for candidate actions in which its message pattern matches with the current situation and its *constraint*. If there are several candidates, then one of them is arbitrarily selected. In this sense constraint acts like a guard. Note that get operation involved in actions not selected are all undone.

In order to send a message which may consitute a part of message pattern calling some action, the message must be sent to a queue first. There are two ways to specify the queue address.

(**send** *Qid message*)	where *Qid* is an identifier of the queue.
(**send** *Oid.Qname message*)	where *Oid* and *Qname* are an identifier of the object and the name of the queue, respectively.

These two forms differ in the effect when the target object has changed the actual queue entity associated with the queue name *Qname*. The former sends messages to the old queue while the latter sends to the new queue.

In POL, the concept of an object is broken into queues and the others. In other words, queues are extracted out from the concept of an object. Together with the fact that queues are external view of an object, this will enable the replacement of objects without being known by other objects. The implication of this is illustrated below using several examples.

(1) Safe mechanism for object recompilation

When an existing object is defined, we could safely replace the
object by a new one in the following way:

1. suspend the object soon after it processed a message
2. install a new object and copy state information to it.
3. have the old object release the queue and have the new own it.

Note that the above scenario needs some help by another object (may be a meta object).

(2) Low cost mechanism for (restricted) reflective object-oriented computation

Since a message queue can be directly manipulated and a pair of states and script can be changed safely as stated above, some sort of reflective computation can be efficiently implemented.

(3) Reverse delegation

A queue is regarded as a job queue to be processed by an object. Now an object can throw its job queue to another object. Usually delegation is realized by installing a back-end object processing those requests which can not be processed by the object. In reverse delegation, instead of a back-end object a front-end object is introduced in the following way:

1. suspend the object and get it to release the queue.
2. create a front-end object and have it own the queue.
3. create a new queue to which the front-end object sends requests which it can not process.
4. have the object own the new queue.

(4) Optimizing message flow

Sometimes messages are forwarded to an object via several objects even when intermediate objects do nothing on them. But now we can make short-cut without disturbing both the sender and the receiver. Suppose that there is message flow from *A* to *B* via *C*. *C* is forwarding message from *A* to *B*. Hence it is not needed. Assume that:

1. *C* has a queue *Qc* for messages from *A*,

2. *B* has a queue *Qb* for messages from *C*,
3. *A* always sends messages to *C* by
   (**send** *Qc message*), and
4. *C* always sends messages to *B* by
   (**send** *Qb message*).

Then we can eliminate *C* from the message route by the following *C*'s actions:

1. (**release** *self.fromA*)
2. (**append** Q_b *Qc*)

The effect of the above action is that *Qc* is concatenated at the tail of *Qb*. After this action, *A*'s messages are directly sent to *Qb* (new *Qb*). Note that, by the append, the following two operations become obsolete, which can be detected by an error handler:

- (**send** *Qb something*)
- (**get** *Qc something*)

These operation should be handled as exceptions.

5 Input Output

Here we describe how I/O features are realized in POL. In POL, streams are implemented as active objects and the read/write request to them are performed.

5.1 Special variable

Lisp systems often provide the following special variables to control input and output operations (Table 2).

In POL, the roles of the special variables in the categories A and B which controls I/O functions are played by the instance variables of stream objects. In contrast, the roles of the special variables in the category C which designate I/O streams are played by namespace objects, details of which will be described later.

5.2 Stream objects

An output stream is an object which has **print-escape**, **print-pretty** and so on as instance variables and has at least a method "write".

```
(defcalss output-stream (stream)
   ((*print-escape* :init-form t
         :accessor stream-print-escape) ... ))
(defmethod (write output-stream)
   (object &key ...) ...)
```

Other methods such as *princ, print, format* and so on can be defined by using *write*. For example, *princ* might be defined as follows:

```
(defmethod (princ output-stream) (object)
         (write self object :escape nil))
```

In this case, it is just nuisance to specify a stream as the first argument. Suppose that the method called *print-self* were defined at the top level class, and any other class could inherit this definition and also could define its own print-self method, then the definition of *write* would be something like:

```
(defmethod (write output-stream)
   (object &key ...)
   (print-self object self))
```

5.3 Namespace databases

Though the special variables in the categories A and B in Table 2 can be confined as local variables of a stream, the other special variables like *standard-

A. Input control	B. Output control	C.Stream designation
read-base	*print-escape*	*standard-input*
read-suppress	*print-pretty*	*standard-output*
readtable	*print-circle*	*error-output*
read-default-float-format	*print-base*	*query-io*
	print-radix	*debug-io*
	print-case	*terminal-io*
	print-gensym	*trace-output*
	print-level	
	print-length	
	print-array	

Table 2 Special variables for I/O

output* may not be. Such information should be managed by shared data bases which are shared by several active objects. For this purpose, we introduce *namespace objects* and the concept of *packs*. A namespace object, which serves its clients as a name server, keeps symbol-value pairs. A client of a namespace is an object which has a reference to a namespace object and may send to it requests for inquiring the associated values of symbols and those for registering new symbol-value pairs.

A pack is nothing but a collection of objects which share a single name-space object. This means that all the objects in a pack share the same standard-input stream, standard-output stream, and so on. Conceptually, objects in a pack cooperate with one another and aim for one particular goal. The reader can consider that the granularity for a pack in POL is analogous to the one of a process in a traditional operating system such as Unix. Currently, POL does not support at least directly pack declarations. The *pack* mechanism is supported as follows: A pack is born when a new namespace object is created and its first client specifies the namespace as its name server; The name server of an object is specified implicitly or explicitly upon birth of the object (by an invocation of a make-instance form or by use of copy-object primitive); If the name server of an object is not specified explicitly, the default name server, the name server of the object creating the new object, is used. An object may not have any name server. For instance, a global shared object such as a window manager or a printer spooler might not belong to any single pack but is shared by a number of packs since a printer spooler might have to display its error message on the error-output stream of the requester. For such shared objects, POL supports a mechanism that every message implicitly contain the reference to the name server of the sender object. In this fashion, a shared object can use the namespace object of the requester object.

6 Comparison

CLOS (Common Lisp Object System) is an object-oriented system which integrates types and classes and it gracefully supports user definable generic functions [CLOS88]. Unfortunately CLOS does not support concurrency.

The object-oriented model on which POL is based includes a mechanism which coordinates multiple processes executed in parallel. POL and CLOS are designed for orthogonal directions. CLOS is designed mainly for integrating an existing Lisp system and an object-oriented system with the emphasis on generic programming. One the other hand, POL is designed mainly for describing distributed systems whereas some part of POL is designed with some influences by CLOS. The semantics concerning classes, inheritances, etc. and the syntax of POL are strongly influenced by new Flavors and CLOS. However, its semantics concerning concurrency is similar to the one of ABCL/1 [Yone86].

There are some proposals of parallel Lisps [Hals84], [Gabr84]. They are primary designed for multi-processor and extended from Lisp by adding parallel constructs, such as *futures* in MultiLisp [Hals84] or *qlet* and *qlambda* in Qlisp [Gabr84]. They do not have data abstraction mechanism. On the other hand, a parallel object-oriented language, such as ABCL/1 [Yone86] is also proposed which is designed based on object-oriented model for describing both multi-processing and distributed processing. But it does not take care large built-in functions, since it is not an extension of Lisp.

7 Conclusion

Here we proposed a parallel object-oriented Lisp called POL, which is suitable for describing wide spectrum programs of distributed processing. In POL, message passing and receiving of an object can be done in parallel with its intrinsic execution. POL has dynamic multiple queues which will be used for describing flexible message handlings. POL also supports namespace databases which can be used for accessing large shared library in consistent way. This report is just an intermediate report, but we hope it will contribute to make a better long range object-oriented Lisp standard.

Acknowledgements

We would like to acknowledge Prof. Itoh, the chair of Japanese SC22/LispWG, for his continual support to our activity. We would also like to acknowledge Dr.Yuasa and other members of the LispWG for their helpfull comments to this report.

References

[CLOS88] ANSI-X3J13, Common Lisp Object System Specification, 88-002R, June, 1988.

[Yone86] Yonezawa, A., Briot, J.-P., Shibayama, E., An Object-oriented Concurrent Programming Language in ABCL/1, Proc. of OOPSLA'86, pp. 258-268, 1986.

[Agha87] Agha, G., Actors: A Model of Concurrent Computation in Distributed Systems, The MIT Press, 1987.

[Hoar78] Hoare, C.A.R., Communicating Sequential Processes, Comm. ACM, Vol. 21 No. 8, pp. 666-677, Aug. 1978.

[Miln80] Milner, R., A Calculus of Communicating Systems, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Stee84] Steele, G., Jr., Common Lisp: the language, Digital Press, 1984.

[Lieb86] Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, Proc. of OOPSLA'86, pp. 214-223, Nov., 1986.

[Flav86] Symbolics Inc., Symbolics Flavors, Sept. 1986.

[Hals84] Halstead, R. H., Jr., Implementation of MultiLisp: Lisp on a Multi-processor, Proc. of the 1984 ACM Symp. on Lisp and Functional Programming, pp. 9-17, Aug., 1984.

[Gabr84] Gabriel, R. P., McCarthy, J., Queue-based Multi-processing Lisp, Proc. of the 1984 ACM Symp. on Lisp and Functional Programming, pp. 25-43, Aug., 1984.

[Gold83] Golberg, A., Robson, D., Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Ueda85] Ueda, K., Guarded Horn Clauses, ICOT TR 103, also in Logic Prog. Cof., LNCS 221, Springer-Verlag, 1985.

[Clar86] Clark, K., Gregory, S., PARLOG: Parallel Program in Logic, ACM Trans. Prog. Lang. Syst., Vol. 8, No. 1, pp. 1-49, 1986.

[Shap83] Shapiro, E., A Subset of Concurrent Prolog and its Interpreter, ICOT TR-003, Jan. 1983.

[Stro86] Stroustrup, B., The C++ Programming Language, Addison-Wesley, 1986.

[Carr89] Carriero, N., Gelernter, D. Linda in Context, Comm. ACM, Vol. 32, No. 4, pp. 444-458, Apr., 1989.

[Cham89] Chambers, C., Ungar, D., Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming language, Proc. of the SIGPLAN '89 Conf. on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 7, pp. 146-160, July, 1989.

[Dijk72] Dijkstra, E. W., Hierarchical Ordering of Sequential Processes, in C.A.R. Hoare, and R.H. Perrot eds., Operating Systems Techniques, Academic Press, pp. 72-93, 1972.