ICOT Technical Report: TR-308

TR-308

A'UM — Parallel Object-Oriented Language upon KL1

by
K. Yoshida and T. Chikayama

October, 1987

$\mathcal{A}'\mathcal{UM}$

– Parallel Object-Oriented Language upon KL1 –

September, 1987

Kaoru Yoshida and Takashi Chikayama

Institute for New Generation Computer Technology (ICOT)

Address: Mita-Kokusai Bldg. 21F., 1-4-28 Mita, Minato-ku, Tokyo 108 JAPAN,
Tel: 03(456)3193
Telex: ICOT J32964
CSNET: {yoshida, chikayama}%icot.jp@relay.cs.net
ARPA: {yoshida, chikayama}%icot.uucp@eddie.mit.edu

Abstract

This paper describes a parallel object-oriented programming language, $\mathcal{A}'\mathcal{UM}$ [1].

$\mathcal{A}'\mathcal{UM}$ has been designed as a user's language upon KL1 which is the kernel language of the parallel inference machine being developed at ICOT, with the aim of writing large application and system programs including the operating system for the machine.

The main feature of $\mathcal{A}'\mathcal{UM}$ is its high level abstractions based on pure parallel objects. In addition to this, $\mathcal{A}'\mathcal{UM}$ is characterized with its stream merging, name association, macro expansion, and modular programming support by class inheritance and method wrapping.

1 Introduction

In solving an application problem in parallel, the first to be explored is how to extract the maximum of parallelism from it.

Parallel logic programming languages have been paid special attention for their simple and atomic mechanism for communication and synchronization, which leaves no room for sequentiality. If a parallel construct is embedded in the atomic level, it is possible to solve the entire problem uniformly from its top to bottom.

In particular, GHC [Ueda85] is extremely simple. GHC realizes synchronization only with an additional construct called *guard*. Unification in the guard is restricted to instantiate the invoking goal and it requires no multiple environments for its execution.

Such a simple mechanism of the guard is desirable to the architecture, say at the execution level, for it makes the practical implementation feasible, especially considering a distributed implementation of it.

[1] $\mathcal{A}'\mathcal{UM}$ is a Japanese word, derived from a Sanskrit "ahum", which consists of $\mathcal{A}$ and $\mathcal{UM}$ and implies the beginning and the end, an open voice and a close voice, and expiration and inspiration.

Further giving some limitation on the guard to make its control more simple, a subset of GHC, called FGHC, has been adopted as the kernel language of the parallel inference machine [Goto86] being developed at ICOT.

In general, it is harder to solve a larger problem, even though it depends on the complexity of the problem itself. First of all, the most important is to design and test the problem systematically and hierarchically, and for promoting this process, abstraction and modularization are indispensable.

Through the development of the operating system SIMPOS [Yokoi84] in the object-oriented language ESP [Chikayama84] for the sequential inference machine PSI [Nakajima86], we have learned that abstractions of object-oriented programming, especially its uniform control of message passing and modularization support of multiple class inheritance and method combination, contribute greatly to the development of such a large system. Moreover, what contributed is not only the object-oriented programming language but also the programming and debugging environment to support it.

Compared to suquential systems, it is said to be much harder to develop parallel systems. Most of the difficulty in debugging a parallel program is to reconstruct the causal chain from actually happening events, say phenomina. Even if events are due to a single causal chain, when the whole bunch of events from small to big are acturally happening, they seem to be independent and at random. In other word, flatness of events makes it difficult, so the hierarchical and modular design and test of programs is much more needed for parallel systems.

Object-oriented programming has been explored also in parallel logic programming languages, especially in CP, in the framework of perpetual process. GHC belongs to the same language family as CP and this framework can be also applied to GHC. Written in object-orientd programming style, computation flow becomes so clear that readability, writability and understandability of program increase.

Instead, however, another problem potential to these languages is made obvious. In one word, as GHC is positioned as the kernel language, these langugages are too primitive and verbose to write large programs. As a result, most of the deadlocks are brought by stream breaking attributing to tiny bugs such as misnaming and mispositioning variables rather than by algorithmic ones.

It must be helpful if variety of programming and debugging tools are provided even in such a primitive level. Much more than that, what is fundamentally and directly needed for developping large systems is a high level language such that enables us to represent programs more simply and concisely and bring the program semantics into relief. Programming and debugging environment should be supported on this level or higher.

This paper describes a parallel object-oriented language, $\mathcal{A}'\mathcal{UM}$, which has been designed upon KL1, for ease of writing large application and system programs, mainly as a description language of the operating system PIMOS for the parallel inference machine.

The organization of the paper is as follows: Firstly, the object-oriented programming style in KL1, which originated $\mathcal{A}'\mathcal{UM}$, is shown. Secondly, after the main features of $\mathcal{A}'\mathcal{UM}$ are outlined, their detail are described with some examples. In addition, the implementation features of $\mathcal{A}'\mathcal{UM}$ onto KL1 are described. Finally, some discussions in comprison of $\mathcal{A}'\mathcal{UM}$ with other related works are given.

2 Programming in KL1

In this section, we summerize the programming in GHC which is the original language of KL1.

2.1 GHC and KL1

A GHC procedure is a set of guarded Horn clauses of the following form:

$$H \; :- \; G_1, \; \dots, \; G_m \; | \; B_1, \; \dots, \; B_n. \qquad (m > 0, \; n > 0)$$

where H, G_i's and B_i's are atomic formulas. H is called a *clause head*, G_i's *guard goals* and B_i's *body goals* respectively. The operator '|' is called a *commitment operator*, the left part before the operator a guard and the right part a body respectively.

Roughly speaking, the execution of a GHC procedure is explained as follows: When a procedure is invoked, all clauses defining the procedure can run in parallel, keeping the following suspension and commitment rules:

Suspension

- Unification invoked directly or indirectly in the guard of a clause C called by a goal G cannot instantiate the goal G.
- Unification invoked directly or indirectly in the body of a clause C cannot instantiate the guard of C until that clause is selected for commitment.

Commitment

If some of the clauses succeed in the execution of the guard part, one and the only one of them is nondeterministically selected. The selected clause continues execution of the body.

The kernel language of the parallel inference machine, called KL1, is a subset of GHC, called Flat GHC (or FGHC in short). FGHC is given a limitation that only system-defined (or built-in) predicates can be invoked in the guard but no user-defined predicates. This limitation makes FGHC free of nested guard.

Since the guard is resticted to instantiate the goal while the body is allowed after being committed, the guard is called a *passive part* and the body an *active part*, and unification in the guard is called *passive unification* and that in the body *active unification* respectively.

2.2 Object-Oriented Programming Style

Shapiro and Takeuchi [Shapiro83B] shows that CP [Shapiro83A] supports object-oriented programming style in the framework of *perpetual process* using stream communication.

Perpetual process is a causal chain of tail-recursive goals, regarding each goal as a process state at some stage. A clause waits for some particular event to hold. After commitment, it takes behaviors corresponding to the event, such as sending messages or modifying its internal states, and invokes an identical goal for the next stage.

Most of the object-oriented programming languages such as [Goldberg83] define an object as an passive entity which accepts a particular set of messages. Regarding each procedure as an object and each message as an event, objects can be represented naturally in logic programming.

Communication is performed by sharing the same entity, say communication media. In this object-oriented programming style, communication media is visible; The list construct is used as a message stream where the car part means a message and the cdr part a succeeding stream respectively.

This framework can be applied not only to CP, but also to GHC and KL1 since both of them are based on the commitment rule and support the list construct and tail recursion as well.

For example, a stack is defined in the object-oriented programming style in KL1 as follows:

Example 1 *Stack in KL1*

```
stack(S) :- true | stack(S, []).
stack([pop(X)|S], []) :- true |
      stack(S, []).
stack([pop(X)|S], Top) :- Top \= [] |
      Top = [get_data(X)|Xs1], stack(S, Top).
stack([push(X)|S], Top) :- true |
      element(Element, X, Top), stack(S, Element).
stack([], Top) :- true | Top = [].

element([get_data(X)|S], Data, Next) :- true |
      X = Data, element(S, Data, Next).
element([set_data(X)|S], Data, Next) :- true |
      element(S, X, Next).
element([get_next(X)|S], Data, Next) :- true |
      X = Next, element(S, Data, Next).
element([set_next(X)|S], Data, Next) :- true |
      element(S, Data, Next).
element([], _, Next) :- true | Next = [].
```

This program can be read as follows: A'`stack` object whose top instantiated with `[]` is created at first. At each stage, the `stack` may receive either of the messages push, pop\1 and `[]`. For the message pop\1, it instantiates the top element and recurs. For the message push\1, it creates an `element` object and recurs. When it receives the nil message `[]`, it terminates. An `element` is similar execpt that it receives either of the messages, `get_data\1`, `set_data\1`, `get_next\1`, `set_next\1` and `[]`.

A stack keeps the current top of the stack as an internal state and an element does a data and a link to the next element. These internal states are represented using local variables; `Top`, `Data` and `Next`, each of which appears in the fixed position, and their new variables are carried by the tail goal for the next stage.

In comparison with other object-oriented languges and their implementations, one of the most noticeable points this style shows is that the semantics of *updating internal states* is logically pure, say free of any side effect. A chain of logical variables placed in some fixed position is recognized as an identical internal state.

As easily seen from such a small example, however, programming in KL1 is too primitive and verbose. It does not only loose readability, writability or understandability of programs, but also is apt to bring a lot of careless bugs such as misnaming or mispassing variables to the next stage, that should occur stream breaking and as a result fall into deadlock.

To represent the programmer's intention more simply, concisely and directly, high level abstraction is needed.

3 Parallel Object-Oriented Language $\mathcal{A}'\mathcal{UM}$

We propose a parallel object-oriented programming language, called $\mathcal{A}'\mathcal{UM}$.

$\mathcal{A}'\mathcal{UM}$ has been designed as a user's language which is compiled into KL1, forcusing its description targets to procedural ones such as system programs including the operating system PIMOS of the parallel inference machine.

$\mathcal{A}'\mathcal{UM}$ is independent of KL1. KL1 programs cannot be contained together in $\mathcal{A}'\mathcal{UM}$ programs.

$\mathcal{A}'\mathcal{UM}$ is caracterized with the following features:

Pure Parallel Object $\mathcal{A}'\mathcal{UM}$ is a pure parallel object-oriented language. All the existing entities in $\mathcal{A}'\mathcal{UM}$ are objects each of which belongs to some class.

A object in $\mathcal{A}'\mathcal{UM}$ is a perpetual process; it repeats the cycle of receiving a message, after that, sending messages to itself or other objects in responce and updating internal associated with names in the side-effect free way. Message passing to objects is the basic execution mechanism to execute an $\mathcal{A}'\mathcal{UM}$ program with.

Stream Merging The external interface to an object, in other word, what is regarded as the object itself from the outside, is a directional stream. Streams are connected from one terminal to another so that the direction should be consistent. In case that three or more terminals specifying an identical stream are specified, or every time a slot is refered, a merger is implicitly inserted.

In $\mathcal{A}'\mathcal{UM}$, non-determinicity can exist only in stream merger. From twigs through stem finally to the target object, messages are sent via mergers. Since stream merging has no logical meaning exept for sending messages to the target object, merger is embedded inside the language, not availble on surface.

Name Association In $\mathcal{A}'\mathcal{UM}$, any object is associated with a name. Updating an object is not giving any side effect on it, but is creating a new version of object and associating it with the name, that is, changing the association.

Macro Expansion Linguistically or syntactically, an $\mathcal{A}'\mathcal{UM}$ program is composed of macro expressions. A macro expression is evaluated to be a target object with a sequence of abstract instructions expanded in the way of functional programming. For instance, a message passing expression represents a new stream after a message is sent, and an arithmetic expression does a new object created as the computation result. With this feature, $\mathcal{A}'\mathcal{UM}$ programs can be written simply and compactly.

Class Inheritance An $\mathcal{A}'\mathcal{UM}$ class can inherit multiple classes. Class inheritance expands method space applicable for an instance, but not brings forth any other instances of the super classes.

Method Wrapping Methods, each of which defines the behavior in response to a received message, are visible or sharable from the outside of the class in the form of *capsule*. Capsules are to define rules of method combination and make it possible to modularize methods and capsules incrementally along the inheritance tree.

The above example of `stack` can be written in $\mathcal{A}'\mathcal{UM}$ as follows:

Example 2 *Stack in $\mathcal{A}'\mathcal{UM}$*

```
class  stack
    slot top.
    :initiate ->
        !top = nil.
    :pop(Data) ->
        (!top \= nil) [
            :true ->
                !top :get_data(^Data) :get_next(^Next),
                !top = Next.
            :false -> .
        ] .
    :push(^Data) ->
        #element :new(^Element),
        !top = Element :set_data(Data) :set_next(!top) .
    : -> !top : ..
end.

class  element
    slot data, next.
    :set_data(^Data) -> !data = Data.
    :get_data(!data) -> .
    :set_next(^Next) -> !next = Next.
    :get_next(!next) -> .
    : -> !next : ..
end.
```

A stack is created by sending a message **:new\1** and is sent messages in some class as follows:

```
#stack :new(^Stack0),
Stack0 :push(1) :pop(^A) :pop(^B) :push(2) :pop(^C),
```

Some other program examples in $\mathcal{A}'\mathcal{UM}$ are shown in Appendix A.

4 Class and Object

4.1 Class

Syntax

$<$ *class definition* $>$ ::=
  class $<$ *class name* $>$
    $<$ *superclass definition* $>$
    $<$ *slot definition* $>$
    { $<$ *method definition* $>$ }
  end '.'

$<$ *superclass definition* $>$ ::=
  super $<$ *superclass name* $>$ { ',' $<$ *superclass name* $>$ } '.'

$<$ *slot definition* $>$ ::=
  slot $<$ *slot name* $>$ { ',' $<$ *slot name* $>$ } '.'

Each $\mathcal{A}'\mathcal{UM}$ object is an instance which belongs to some class.

Class is an index to the module which defines the attributes and functions of its instance objects. In other word, each of the attributes and methods is indexed with its own class name and is applied with the class name.

Each class can inherit multiple classes. By inheriting a class, the set of attributes and functions applicable for an instance object is expanded, but no other instances of the super classes are created.

Class is treated as an immutable object which will be mentioned later, but belongs to no other class; there is no notion of meta class.

4.2 Object

Each $\mathcal{A}'\mathcal{UM}$ object is characterized with the following attributes:

Original Class which the object belongs to and is created from.

For an instance object, the original class is constant through life.

Current Class which defines a method which is applied for the received message.

For an instance object, the current class is variable depending on the received message. When a method of some class is applied, the instance object is said to be *under* the class.

At the initiation and every time the object recurs, the current class is set to be the original class.

External Interface Streams which the object offers to the outside to let them send messages to itself through. One or more interface streams can be offered, each of which is assigned a different priority and priority-merged into the internal input stream **self**. From the outside, the given interface stream is regarded as the target object itself.

Internal Input Stream (Self) through which the object receives messages. The internal input stream is accessible with the name `$self`.

Slots which are associated slot values with their identifier.

Slot value is a message stream to some object.

Each slot is uniquely identified with the original class name and its slot name; any slot is visible only in its own class which defines the slot. Slots defined in some class have nothing to do with those defined in the super or inferior classes, even if they have the same slot name.

When an object is created, a global stream named `$system` is given. This global stream is for raising a message to the underlying operating system through. In the conceptual model, this global stream may be treated as one of the slots.

Supers which is the entire inheitance tree composed of all the super classes that the original class of the object inherits directly and indirectly. The inheritance tree is constructed from the `super` definition in the left-first depth-first order.

For an instance object, the supers are constant through its life.

Delegates which is the rest of the inheritance tree appearing later than the current class. For each class, the first of its delegates is accessible with the name $super.

As well as the current class, the delegates are variable depending on the received message.

At the initiation and every time the object recurs, delegates are initialized with the supers.

Example 3 *Object Attributes*

Given the following class definitions:

```
class c21
    super c11, c12.
    slot  s.
    :ma -> !s :ma.
end.

class c3
    super c21, c22.
    slot  s.
    :mb -> !s :mb.
end.
```

From some other class, an instance of the class c3 is created as:

```
#c3 :new(^C3),
C3 :ma :mb
```

1. For an instance of the class c3, the following two attributes; **original class** and **supers** are constant through its life:

Original Class:	c3
Supers:	c21 → c11 → c12 → c22

2. The slot s in the class c21 is independent of that in the class c21.
3. During execution, the following two attributes; **current class** and **delegates** are changeable: When executing a method for the message :mb,

Current Class:	c3
Delegates:	c21 → c11 → c12 → c22

   When executing a method for the message :ma,

Current Class:	c21
Delegates	c11 → c12 → c22

4.3 Object Life

An $\mathcal{A}'\mathcal{UM}$ object is a perpetual process whose life is drawn as follows:

Creation When a message :new\1 (or :new_with_priority\2) is sent to a class, an instance object of the class is created, a message :initiate is sent to the object, and an interface stream (or a set of interface streams) to the new object after the message :initiate is sent is returned.

Initiation Whenever an instance is created, it is implicitly sent a message :initiate. The method for the message :initiate can be overwritten, which is predefined as:

```
:initiate -> .
```

Generation Including the internal states such as **self**, slots and **system**, any object is associated with a name. Updating an object is not giving any side effect on it, but creating a new version of object and associating it with the name.

The term while the same version of object can be associated with the name is called *a generation*, and chaging the name association is called *generation descending*.

For one generation, either connecting streams, sending a message, delegating a message or creating a volatile object can be taken as a behavior.

Cycle After receiving an external message, an object behaves decending one generation to another. A sequence of generations derived from receiving one external message is called a *cycle*. A script of the cycle for one external message is called a *method*.

Termination Terminating the object life is decending no more generation.

When to terminate can be defined freely by the users; when receiving not only the nil message (just `':'`) but also any use-defined message. The method for the nil message can be overwritten, which is predefined as:

```
    : -> $slots : ..
```

Example 4 *Object Life*

```
:push(^Data) ->
    #element :new(^Element),                          % generation-1 %
    !top = Element:set_data(Data):set_next(!top).     % generation-2 %
                                                      % recur %
: -> !top : ..                                        % terminate %
```

4.4 Mutable and Immutable Objects

A'UM objects are categorized into two; *mutable objects* and *immutable objects*, depending on whether they have changeable internal states or not.

Class objects are immutable. The instances of some primitive classes such as `true`, `false`, `integer`, `vector` and `string`, are immutable.

Both of the mutable and immutable objects are treated completely in the same way in terms of message passing. For example, messages are sent to a string `"abcde"` and an integer `1` as well as to mutable objects such as `Stack` and `Element`.

Example 5 *Mutable and Immutable Objects*

```
Stack:push(1)
Element:set_data(Data)
"abcde":element(3, ^X)
1:add(2, ^X)
```

5 Other Basic Notions

5.1 Name Association

A'UM objects are associated with names. Names are categorized into two; *temporary names* and *permanent names*. The term while an identical object can be associated with the same name is called *name scope.*

Permanent Names Among permanent names are system-defined names such as `$self`, `$system` and `$super`, and user-defined slot names.

The name scope of a permanent name is within one generation.

Temporary Names Variables are temporary names. Among variables are parameter variables which are carried in messages and temporary variables which are generated in the cycle.

The name scope of a temporary name is within one cycle.

5.2 Stream Merging

In the leading sections, we described the internal view of objects. Here in this section, we profile objects from the outside.

The external interface to an object is directional streams. In other word, if a stream to the object is given, the stream can be regarded as the object itself from the outside.

In $\mathcal{A'UM}$, non-determinicity is absorbed in stream merger. From twigs through stems to the target object, messages are sent via mergers. Since stream merging has no logical meaning except for sending messages to the target object, mergers are embedded in the language, not availble on surface.

A stream merger is inserted in the following two cases:

- when the input terminal of a temporary name (or variable) occurs multiple times in one cycle.
- when any permenanent name except `$self` occurs multiple times in one generation.

5.2.1 Variable Mode

In order to specify the direction of stream, variable occurences have their terminal *modes*, either of *input* or *output*.

- Variables have only one occurrence with `'^'`, called an output terminal, and one or more occurrence without `'^'`, called input terminals.
- An object is somewhere ahead of the output terminal (with `'^'`).
- A stream is connected to the output terminal (with `'^'`).
- Messages can be sent to the input terminals (without `'^'`).
- All the messages sent to the input terminals are merged and sent to the target object ahead of the output terminal.

Example 6 *Variable Mode*

```
:consult(^A, ^B, ^C) :-
    A :try(X),
    B :try(X),
    C :select(^X).
```

is translated to:

```
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, consult(A, B, C), NewSelf) |
    send(A, try(X1)),
    send(B, try(X2)),
    send(C, select(X3)),
    merge(X1, X2, X3),
    object(Original, NewSelf, Slots, Original, Supers, Supers).
```

5.2.2 Slot Access

Referring

If a slot is referred, it opens a stream and a new generation of the slot, both of which are merged into the current generation of the slot.

```
X = !some_name
```

is translated to:

```
send(NewSelf, get_slot({Class, some_name}, X), Self)
```

Updating

Slots are updated when they are specified as the destination of stream connection or message sending. Updating a slot is to update the association table so that it can associate the specified new value with the slot name, when the old value is closed.

```
!some_name = Value
```

is translated to:

```
send(NewSelf, set_slot({Class, some_name}, Value), Self)
```

5.3 Macro Expansion

Another feature that characterizes $\mathcal{A}'\mathcal{UM}$ from the lingustic point of view is that $\mathcal{A}'\mathcal{UM}$ syntax is based on macro expansion. With this feature, $\mathcal{A}'\mathcal{UM}$ programs can be written compactly and clearly.

An $\mathcal{A}'\mathcal{UM}$ program is composed of *macro expressions* each of which is evaluated to be a target object with a sequence of abstract instructions expanded in the way of functional programming.

For example, sending a message to a variable is an expression which is evaluated to the new variable after the message is sent, so it can be specified wherever expressions are available.

Example 7 *Macro Expansion*

```
:create(A0:initialize(^InitialList), InitialList) ->
     #a :new(^A0).
```

is equivalent to:

```
:create(A, InitialList) ->
     #a :new(^A0),
     ^A = A0 :initialize(^InitialList).
```

6 Method

Syntax

< *method definition* > ::=
    ':' < *message* > '->' < *cycle* > < *terminator* >

< *cycle* > ::=
    < *generation* > { ',' < *generation* > }

A script of the cycle for one external message is called a *method*. One cycle consists of generations and for each generation either of the following behaviors can be defined:

- Connecting streams
- Sending a message
- Delegating a message
- Creating a volatile object

6.1 Stream Connection

Syntax

< *connection* > ::=
    < *output terminal* > '=' < *expression* >

< *output terminal* > ::=
    < *output variable* > | < *slot* >

< *output variable* > ::=
    '^' < *variable name* >

< *slot* > ::=
    '!' < *slot name* >

Connection is to connect an input terminal to an output terminal.

An expression on the right side is evaluated to a target value with a sequence of abstract instructions expanded and the target value is connected to the output terminal.

For the *output variable*, it means a new generation of variable. For the *slot*, it means the next generation of slot, say updating the slot. The semantics of < *slot* > is different depending on where it appears; referring slot in an expression on the right and updating slot on the left.

6.2 Message Sending

Syntax

< *message sending expression* > ::=
    < *destination* > { ':' < *message* > } < *last message* >

< *destination* > ::=
{} | < *input variable* > | < *slot* > | `$system`

< *last message* > ::=
{ ':' < *message* > | ':' }

< *input variable* > ::= < *variable name* >

A message sending expression is evaluated to be a new object after the message has been sent. By repeating this evaluation, a sequence of message can be sent to an object.

The message sending expression can be specified wherever expressions are allowed, for example, as a parameter of another message or as the source (the right part) of an assignment.

The meaning of message sending depends on what is specified as the destination as follows:

{} *(default)*

prepends the message sequence to the current **self**. Sending a message to **self** is evaluated to be new **self** after the message is prepended.

```
            :m(P)
represents  Self1
where       send(Self1, m(P), Self0)
```

< *input variable* >

appends the message sequence to the input variable. Sending a message to an input variable is evaluated to be a new variable after the message is appended.

```
            X :m(P)
represents  X1
where       send(X, m(P), X1)
```

Closing a stream with the nil message ':',

```
            X :
represents  nil
where       close(X)
```

< *slot* >

appends the message sequence to the slot and updates the slot. Sending a message to a slot is evaluated to be the new slot value after the message is sent.

```
            !s :m(X)
represents  Slot1
where       send(Self1, slot(s, Slot0, Slot1), CurrentSelf),
            send(Slot0, m(X), Slot1)
```

`$system`

raises the message sequence to the system stream. Raising a message is evaluated to be the new system stream after the message is raised.

```
            $system :m(X)
represents  Slot1
where       send(Self1, slot('$system, System0, System1), CurrentSelf),
            raise(System0, m(X), System1)
```

6.3 Message Delegation

Syntax

> < *delegation* > ::=
>   < *delegate* > '<-' { ':' < *message* > } < *last message* >
>
> < *delegate* > ::=
> $super | '#'< *superclass name* >

If a class inherits super classes, a sequence of messages can be *delegated* to any of the super classes.

In $\mathcal{A}'\mathcal{UM}$, class is an index to categorize the method space applicable for an instance with. Class inheritance is to expand the method space, but not to bring forth any other intances of super classes. Therefore, delegating a message to some desired super class means specifying *under* which class the message should be received. Instead of the target message, an indirect message delegate\2 which is enclosed the target message and the desired super class in it is sent to the instance to itself under the direct super class. For specifying which class to delegate, there are the following two ways:

Direct Super Class
With $super specified, messages are delegated to the object itself under the direct super.

```
:m(^X) ->
    $super <- :mm(X).
```

is translated to:

```
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, m(X), NewSelf),
    Delegates = [Super|Delegate1] |
    send(Self1, delegate(Super, mm(X)), NewSelf),
    object(Super, Self1, Slots, Original, Supers, Delegates1).
```

Specific Super Class
By specifying a the class name, messages can be delegated to the object itself under any of its super classes.

```
:m(^X) ->
    #some_super <- :mm(X).
```

is translated to:

```
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, m(X), NewSelf),
    Delegates = [Super|Delegates1] |
    send(Self1, delegate(some_super, mm(X)), NewSelf),
    object(Super, Self1, Slots, Original, Supers, Delegates1).
```

When a super class receives the message delegate(Class, Message), the class either sends the target message Message under itself or send to its further super class depending on whether the specified class Class is itself or not.

```
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, delegate(Class, Message), NewSelf),
    Current == Class |
    send(Self1, Message, NewSelf),
    object(Current, Self1, Slots, Original, Supers, Delegates).
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, delegate(Class, Message), NewSelf),
    Current \= Class,
    Delegates = [Super|Delegates1]  |
    send(Self1, Message, NewSelf),
    object(Super, Self, Slots, Original, Supers, Delegates1).
```

6.3.1 Default Message

When a method to receive some message is not defined in some class, the received message is delegated to itself under the direct super class.

```
:$default ->
    $super <- :$default.
```

is translated to:

```
object(Current, Self, Slots, Original, Supers, Delegates) :-
    receive(Self, Message, NewSelf),
    Delegates = [Super|Delegates1],
    % Message is not defined in Current % |
    send(Self1, delegate(Super, Message), NewSelf),
    object(Super, Self1, Slots, Original, Supers, Delegates1).
```

6.4 Volatile Object Creation

Syntax

< *volatile object creation* > ::=
    < *immutable volatile object deinition* >
    < *mutable volatile object definition* >

< *immutable volatile object defintion* > ::=
    < *volatile object* > '['
        { < *method definition* > }
    ']'

< *mutable volatile object definiiton* > ::=
    < *volatile object* > '{'
        < *slot definition* >
        { < *method definition* > }
    '}'

< *volatile object* > ::=
    < *input variable* > | < *output variable* > | < *expression* >

6.4.1 Volatile Object

In $\mathcal{A'UM}$, for handling conditions, one additional notion of *volatile object* is introduced.

Volatile objects are those which are created in a method and terminate in it. This notion is derived from that the ordinary objects are already condition handlers in terms that receive a particular set of message and behave differently according to the received message. Volatile objects are different from the ordinary objects only in that they are not given any class name. Applying this notion, the definition of object can be nested.

6.4.2 Basic

The volatile object field means an external interface to the volatile object, toward which messages are flown. This is the basic.

Output Terminal If an output terminal (with ^) is specified in the volatile object field, there must be one or more input terminals which are merged into the output terminal. Messages are flown from the input terminals to the output terminal.

```
^T [
  :p(X, Y) -> P.
  :q(U, V, W) -> Q.
]
```

is translated:

```
(
 receive(T, p(X, Y), _) | P ;
 receive(T, q(U, V, W), _) | Q
)
```

Expression Representing Input Terminal If an expression which is evaluated to be an input terminal (without ^), the input terminal is connected to an output terminal in the above.

6.4.3 Extension

If an input terminal (without ^) is specified for the volatile object field, it implies there already exists some output terminal into which messages should be flown. The semantics is extended with the notion of *reflection*.

Input Terminal If an input terminal (without ^) is specified, a message `:who_are_you(Who)` is sent to the input terminal. An volatile object is created so that it should take the output terminal `Who` as its external input stream. For each message from `Who`, a method is defined.

```
T [
    :p -> P.
    :q -> Q.
]
```

is translated to:

```
send (T, who_are_you(Who)),
(
 receive(Who, p, _) | P ;
 receive(Who, q, _) | Q
)
```

Expression Representing Output Terminal If an expression which is evaluated to be an output terminal (with ^), the output terminal is connected to an input terminal in the above.

6.4.4 Applications

Pattern Matching Message `:who_are_you(Who)` transforms an immutable object to a message stream which contains the frozen image of the object as a message. Using this mechanism, pattern matching can be represented.

Example 8 *Pattern Matching*

```
T [
  :1 -> P1.
  :2 -> P2.
  :3 -> P3.
]
```

If-then-else Construct If an conditional expression, which is evaluated to be either `true` or `false` object, is specified in the volatile object field, it means the if-then-else construct.

```
X == Y [
    :true -> Then.
    :false -> Else.
]
```

is translated to:

```
send(X, equal(Y, Result)),
send(Result, who_are_you(Who)),
(
 receive(Who, true, _)  | Then;
 receive(Who, false, _) | Else
)
```

6.4.5 Mutable and Immutable Volatile Objects

Volatile objects are categorized into two; immutable volatile objects and mutable volatile objects.

We explain the difference between them with the following example of number generator:

Example 9 *Number Generator* generates a sequence of integers i in the range $n \leq i < m$.

```
class  numbers
    slot  max, n, numbers.
    :initialize(IL) ->
        ^IL {                              % mutable volatile object %
            :set_max(^M) -> !max = M.
            :set_n(^N) -> !n = N.
            :set_numbers(^Ns) -> !numbers = Ns.
        }.
    :do ->
        (!max > !n) [                       % immutable volatile object %
            :true -> !n :add(1, ^N1),
                     !numbers :n(N1),
                     !n = N1,
                     :do.
            :false -> :terminate.
        ].
    :terminate -> ..
end.

    #numbers :new(^Numbers),
    Numbers :initialize(^IL) :do,
    IL :set_max(M) :set_n(0) :set_numbers(Ns) :
```

Immutable Volatile Object The immutable volatile object may neither have any internal state nor recur after receiving one message, that is, it is supposed to terminate at once after receiving the message.

The name scope in an immutable volatile object is the same as that in the outside object. Temporary names such as parameter and temporary variables used in the outside object are also visible in the immutable volatile object.

In the above example, when messge `:do` is sent, an immutable object is created for the result of the condition `(!max > !n)`. The volatile object accepts either message`:true` or `:false`. When it receives message `:true`, the volatile object sends message `:add\2` to slot `!n` and message `:n\1` to slot `!numbers`, updates slot `!n`, sends message `:do` to the outside object and then terminates. For message `:false`, the volatile object sends message `:terminate` to the outside object and then terminates.

Mutable Volatile Object The mutable volatile object may have internal states and recur in the same way as the oridinary mutable objects do.

The name scope in a mutable volatile object is one level *deeper* than that in the outside object. Temporary names used in the outside object are not visible in the mutable volatile object.

In the above example, when message `:initialize\1` is sent, a mutable volatile object is created for output terminal `^IL`. The volatile object accepts either message `:set_max\1`, `:set_n\1` or `:set_numbers\1` and sets the corresponding slot and recurs until the nil message is sent, where the method for the nil message is predefined.

7 Modularization

To support modularization of programs, KL1 provides the functions of class inheritance and method wrapping.

7.1 Class Inheritance

A class can inherit multiple classes. The inheritance tree composed of super classes is constructed traversing the super definition in the left-first depth-first order.

Class inheritance in $\mathcal{A}'\mathcal{UM}$ expands the set of accessible slots and applicable methods for an instance, but brings forth no other instances of the super classes.

In the leading sections, we mentioned several features related with class inheritance which are summarized as follows:

1. Each slot is identified with the combination of its own class name and slot name.
2. Each instance has a current class name as an internal state which is variable depending on the currently received message, in addition to the original class name which is constant through life.
3. A message is delegated to the instance *itself* by enclosing it with the delegate class in the `delegate` message.

7.2 Method Wrapping

Syntax

< *message* > ::=
    < *message pattern* > < *capsule info* >

< *capsule info* > ::=
    '@' < *capsule name* > | '&' < *component method name* >

In addition to the above features, a notion of *method wrapping* is introduced for controlling the visibility of methods. Method wrapping is to encapsulate methods. The following two constucts are used to support it:

Capsule Capsules are visible not only in the class but also from its outside. Calsules are used to define rules of method combination and make it possible to modularize methods and capsules incrementally along the inheritance tree. Those methods which are not specified any capsule infomation are called *entry capsules.*

Component Method Component methods are visible only in their class and used to constitute capsules.

7.2.1 Default Capsule Environment

The following capsule environment is provided to each method for deault:

```
:$message ->                        % entry capsule %
    :$message@before,
    :$message@primry,
    :$message@after.
:$message@before ->                 % before capsule %
    :$message&before,
    $super <- :$message@before.
:$message@primary ->                % primary capsule %
    :$message&primary.
:$message@after ->                  % after capsule %
```

```
        $super <- :$message@after,
        :$message&after.
    :$message&before -> .                % before component method %
    :$message&primary ->                 % primary component method %
        $super <- :$message@primary.
    :$message&after -> .                 % after component method %
```

8 Abstract Machine

An $\mathcal{A}'\mathcal{UM}$ program is translated to a sequence of abstract instructions. A system which can execute the abstract instuction set is called an $\mathcal{A}'\mathcal{UM}$ *abstract machine.*

8.1 Abstract Instruction Set

The entire set of abstract instructions are listed as follows:

receive(Self, Message, NewSelf)
: receives a message from the internal input stream and opens the new input stream after the message is received.

closed(Object)
: receives a nil message from the internal input stream.

send(Object, Message, NewObject)
: sends a message to an object stream and opens the new object stream after the message is sent.

send(Object, Message)
: sends a message to an object and closes a new object stream. It is equivalant to:

```
send(Object, Message, NewObject),
close(NewObject)
```

close(Object)
: closes an object stream.

raise(System, Message, NewSystem)
: raises a message to the underlying machine and opens the new system stream after the message is raised.

merge(X, Y, Z)
: merges two streams X and Y, into a stream Z. In KL1, it is translated to:

priority_merge(X, Y, Z)
: merges two streams X and Y, into a stream Z, monitoring X prior to Y.

new(ClassName, N, Externals)
: creates an instance object of the specified class and opens a stream to the vector consisting of N external input streams which are priority-merged into the internal input stream to the object.

new(ClassName, External)
: creates an instance object of the specified class, and opens an external input stream which is connected to the internal input stream of the object.

object(CurrentClass, Self, Slots, OriginalClass, Supers, Delegates)
: continues its execution to the next cycle, that is, recurs.

8.2 Primitive Messages

$\mathcal{A}'\mathcal{UM}$ defines the following set of primitive messages which are implicitly sent to objects by the compiler:

get_slot(^SlotId, Value)
: refers a slot with the slot identifier; opens a stream and a new generation of the slot both of which are merged into the current generation of the slot as follows:

```
object(Current, Self, Slots, System, Original, Supers, Delegates) :-
    receive(Self, get_slot(SlotId, Value), NewSelf) |
    send(Slots, associate(SlotId, OldSlot, NewSlot, NewSlots)),
    merge(Value, NewSlot, OldSlot),
    object(Original, NewSelf, NewSlots, System, Original, Supers, Supers).
```

set_slot(^SlotId, ^Value)
: updates a slot with the slot identifier; associates the specified value with the slot identifier and closes the old generation of slot value as follows:

```
object(Current, Self, Slots, System, Original, Supers, Delegates) :-
    receive(Self, set_slot(SlotId, Value), NewSelf) |
    send(Slots, associate(SlotId, OldSlot, Value, NewSlots)),
    close(OldSlot),
    object(Original, NewSelf, NewSlots, System, Original, Supers, Supers).
```

slot(^SlotId, ^OldValue, NewValue)
: refers a slot with the slot identifier and updates it with the new value.

```
object(Current, Self, Slots, System, Original, Supers, Delegates) :-
    receive(Self, slot(SlotId, OldValue, NewValue), NewSelf) |
    send(Slots, associate(SlotId, OldValue, NewValue, NewSlots)),
    object(Original, NewSelf, NewSlots, System, Original, Supers, Supers).
```

who_are_you(Who)
: when this message is sent to an immutable object, it transforms the object into a message stream *Who*. If this message is sent to to a mutable object, *Who* is closed.

delegate(^Class, Message)
: delegates a message which should be sent to the object itself under the class.

8.3 Primitive Classes

$\mathcal{A}'\mathcal{UM}$ provides the following primitive classes:

Immutable class **vector**

```
Vector1 = Vector:vector_element(Position, ^Element)
Vector1 = Vector:set_vector_element(Position, ^Element)
```

Immutable class **associative_table**

```
Table1 = Table:associate(Key, Old, ^New, ^NewTable)
```

Immutable class **string**

```
String1 = String:string_element(Position, ^Element)
String1 = String:set_string_element(Position, ^Element)
```

Immutable class **integer**

```
Integer1 = Integer:add(Integer2, ^Integer3)
Integer1 = Integer:subtract(Integer2, ^Integer3)
Integer1 = Integer:multiply(Integer2, ^Integer3)
Integer1 = Integer:divide(Integer2, ^Integer3)
Integer1 = Integer:mod(Integer2, ^Integer3)
Integer1 = Integer:eq(Integer2, ^TrueOrFalse)
Integer1 = Integer:not(Integer2, ^TrueOrFalse)
Integer1 = Integer:gt(Integer2, ^TrueOrFalse)
Integer1 = Integer:ge(Integer2, ^TrueOrFalse)
Integer1 = Integer:lt(Integer2, ^TrueOrFalse)
Integer1 = Integer:le(Integer2, ^TrueOrFalse)
```

Immutable class **true**

Immutable class **false**

9 Implementing $\mathcal{A}'\mathcal{UM}$ onto KL1

9.1 Message Sending

$\mathcal{A}'\mathcal{UM}$ has been firstly designed on top of KL1 and a message stream is implemented as a list.

For example, the expression of message sending

```
^NewX = X :add(Y, Z)
```

which is represented as:

```
send(X, add(Y, Z), NewX)
```

is translated in KL1 to:

```
X = [add(Y, Z)|NewX]
```

As mentioned before, in $\mathcal{A}'\mathcal{UM}$, both of the mutable and immutable objects are treated uniformly in the same way in message passing.

9.2 Unification Failure Handling

In the above example, let X be an integer 1 and Y be 2. Then the following unification must be made true.

```
1 = [add(2, 3)|1]
```

In order to realize it, some extensions have been introduced into KL1.

In the original KL1 language, such a unification normally *fails*. For a certain goal and all subgoals of the goal, a predicate for handling such failure can be specified, which is called in stead of simple failure. It is called the *unification failure handler*. The unification failure handler receives two original arguments, of the unification. If the unification was between two structures and the unification failed for certain elements of them, then these elements are passed as the argument of the unification failure handler. The execution of the unification handler takes place of the execution of the unification itself.

If integers should understand add messages, the unification failure handler should have clause such as the following:

```
handler(Int, [add(Addend, Sum)|Rest]) :-
    integer(Int), integer(Addend) |
    add(Int, Addend, Sum), Int = Rest.
```

A program piece:

```
^NewX = X :add(1, Y) :add(2, Z)
```

is translated to:

```
X = [add(1, Y), add(2, Z)|NewX]
```

If X is 3 and the unification handler is defined as given above, the execution will be as follows:

1. The unification fails.
2. The unification failure handler is invoked with two arguments, 3 and `[add(1, Y), add(2, Z)|NewX]`.
3. The head unification and guard tests succeed making `Int = 3`, `Addend = 1`, `Sum = Y` and `Rest = [add(2, Z)|NewX]`.
4. The goal `add(Int, Addend, Sum)`, i.e., `add(3, 1, Y)` is executed instantiating `Y` to 4 (in parallel with the following steps).
5. The unification `Int = Rest`, i.e., `3 = [add(2, Z)|NewX]` is executed and fails.
6. The unification failure handler is invoked with two arguments, 3 and `[add(2, Z)|NewX]`.
7. The head unification and guard tests succeed making `Int = 3`, `Addend = 2`, `Sum = Z` and `Rest = NewX`.
8. The goal `add(Int, Addend, Sum)`, i.e., `add(3, 2, Z)` is executed instantiating `Z` to 5 (in parallel with the following steps).
9. The unification `Int = Rest`, i.e., `3 = NewX` is executed and `NewX` becomes 3.

The above explained mechanism is realized by defining the unification failure handler appropriate for execution of $\mathcal{A}'\mathcal{UM}$. Users who prefer simple KL1 language can define his unification failure handler which simply fails, keeping the original semantics of KL1.

The use of the same mechanism of unification failure handling is not restricted for $\mathcal{A}'\mathcal{UM}$ implementation. Implementation of other higher level languages may also require some extension of unification.

10 Related Works

In comparison of $\mathcal{A}'\mathcal{UM}$ with other related works, Vulcan [Kahn86] is the closest in approach. Vulcan is designed as a preprocessor on top of CP and is based on perpetual processes connected via streams. Vulcan supports a variety of functions as $\mathcal{A}'\mathcal{UM}$ does, but both are different from each other as follows: Firstly, unlike in $\mathcal{A}'\mathcal{UM}$, name space is flat in Vulcan. Temporary and parameter variables are treated in the same way as the variable representing the internal states of objects and it is hard for users to grasp the transition of each internal state. On the way of class inheritance, these are different. Vulcan supports two ways of inheritance; mehod copy and delegation, while $\mathcal{A}'\mathcal{UM}$ does only delegation. For developing large systems, the amount of copied method cannot be ignored. Most of the difference is that $\mathcal{A}'\mathcal{UM}$ is an independent language rather than a preprocessor and supports message sending as a primitive instruction, while Vulcan is a preprocessor.

Mandala [Furukawa84] was also designed on CP as Vulcan. Mandala supports the association of objects with their names, but globally through the name server, which must bring a bottleneck in performance. In $\mathcal{A}'\mathcal{UM}$, the name association is solved in each object, say local, not bringing such a problem. Another difference is that message receiving in $\mathcal{A}'\mathcal{UM}$ is based on one-at-a-time principle. Until all the behaviors for an external message are taken, no other external messages are received. Mandala allows multiple messages to be received, so it makes its implementation difficult.

These languages are exploring to realize object-oriented programming with clean semantics. As another approach toward object-oriented programming with clean semantics based on side-effect free foundation, FOOPS [Goguen86] should be listed even though it is in function programming. FOOPS distinguishes objects from abstract data types and the basic construct is much more restrictive and complicated.

11 Current and Future Works

We are now on the stage of experimenting the implementation of $\mathcal{A}'\mathcal{UM}$ compiler into KL1. We will write a variety of sample programs to investigate the expressive power of $\mathcal{A}'\mathcal{UM}$ and start writing the operating system PIMOS in this version.

In the future, we are planning to explore the better implementation such that primitive objects should work more effectively. The development of programming and debugging enviroment will be another work.

References

[Chikayama84] Takashi Chikayama: *ESP Reference Manual*, Technical Report TR-044, ICOT 1984

[Yokoi84] Toshio Yokoi: *Sequential Inference Machine: SIM -Its Programming and Operating System*, Proc. of FGCS'84, Tokyo 1984

[Nakajima86] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto and M. Mitsui: *Evaluation of PSI Micro-Interpreter*, Proc. of IEEE COMPCON-spring'86, March 1986

[Goto86] Atsushi Goto, Shunichi Uchida: *Toward a High Pergormance Parallel Inference Machine - The Intermideate Stage Plan of PIM*, Technical Report TR-201, ICOT 1986

[Ueda85] Kazunori Ueda: *Guarded Horn Clauses*, Technical Report TR-103, ICOT, 1985

[Shapiro83A] Ehud Shapiro: *A Subset of Concurrent Prolog and Its Interpreter*, Technical Report TR-003, ICOT, 1983

[Shapiro83B] Ehud Shapiro and Akikazu Takeuchi: *Object Oriented Programming in Concurrent Prolog*, New Generation Computing, 1 (1983), OHMSHA Ltd. and Springer-Verlag

[Furukawa84] Koichi Furukawa, Akikaze Takeuchi, Susumu Kunifuji, Hideki Yasukawa, Masaru Ohki and Kazunori Ueda: *Mandala: A Logic Based Knowledge Programming System*, Proc. of the International Conference on FGCS 1984

[Goldberg83] Adele Goldberg and David Robson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesly, Reading, 1983

[Kahn86] Kenneth Kahn, Eric D. Tribble, Marks S. Miller and Daniel G. Bobrow: *Vulcan: Logical Concurrent Objects*, Technical Report, Xerox Palo Alto Research Center, 1986 (its preliminary version appears as *Objects in Concurrent Logic Programming Languages"* in the proceeding of the ACM OOPSLA'86)

[Goguen86] Joseph A. Goguen, Jose Meseguer: *Extensions and Foundations of Object-Oriented Programming*, Internal Memo, SRI & CSLI, 1986

Appendix A. Examples in $\mathcal{A}'\mathcal{UM}$

Example 10 *Prime Number Generator*

```
    #numbers :new(^N),
    N :do(100, 2, Ns),
    #prime :new(^P),
    ^Ns = P :start(2, Ps)

class  numbers                   % Another Solution of Number Generator %
    :do(^M, ^N, ^Ns) ->
        (M > N) [
           :true ->
               N :add(1, ^N1),
               :do(M, N1, Ns:n(N1)).
           :false ->
               Ns :,
               :terminate.
        ].
    :terminate -> ..
end.

class  prime
    slot  flag, value, next, report.
    :initiate ->
        !flag = 0.
    :start(^V, ^Ps) ->
        !value = V,
        !report = Ps :n(!value).
    :n(^N) ->
        (N / !value == 0) [
           :true -> .             % N is a multiple of the value. %
           :false ->
               (!flag == 0) [
                  :true ->
                      !flag = 1,
                      #prime :new(^P),
                      !next = P :set_value(N, !report).
                  :false ->
                      !next :n(N).
               ].
        ].
end.
```

Example 11 *Bat and Penguin*

```
class  bat
      super  flier, mammal.
  :is_bird(Ans) ->
      ^Ans = yes.       % If you always answer 'yes', you must be a bat! %
end.

class  penguin
     super  bird.
  :flies(Ans) ->
      ^Ans = no.
end.

class  bird
     super flier, with_feather, vertebrata.
  :is_bird(Ans) ->
      ^Ans = yes.
  :is_mammal(Ans) ->
      ^Ans = no.
end.

class  flier
     super  with_wing.
  :flies(Ans) ->
      ^Ans = yes.
end.

class  with_wing
  :has_wings(Ans) ->
      ^Ans = yes.
end.

class  with_feather
  :has_feather(Ans) ->
      ^Ans = yes.
end.

class  mammal
     super  vertbrata.
  :is_mammal(Ans) ->
      ^Ans = yes.
  :is_bird(Ans) ->
      ^Ans = no.
  :flies(Ans) ->
      ^Ans = no.
  :is_self_regulating_temperature(Ans) ->
      ^Ans = yes.
end.

class  vertebrata
  :is_vertebrate(Ans) ->
      ^Ans = yes.
end.
```

Example 12 *Slot Machine*

```
      #slot_machine :new(^S),
      S :insert(100) :insert(50) :get(120, ^Ticket, ^Change)

class  slot machine
     slot  n10, n50, n100, n500.
  :initiate ->
      !n10 = 0,
      !n50 = 0,
      !n100 = 0,
      !n500 = 0.
  :insert(^Coin) ->
      Coin [
         :10 -> !n10 = !n10 + 1.
         :50 -> !n50 = !n50 + 1.
         :100 -> !n100 = !n100 + 1.
         :500 -> !n500 = !n500 + 1.
      ].
  :get(^Price, Ticket, Change) ->
      ^Change0 = 10 * !n10 + 50 * !n50 + 100 * !n100 + 500 * !n500 - Price,
      (Change0 >= 0) [
         :true ->
             ^Change = Change0,
             ^Ticket = issued,
             :initiate.
         :false ->
             ^Change = 0,
             ^Ticket = unissued.
      ].
  :cancel(Change) ->
      ^Change = 10 * !n10 + 50 * !n50 + 100 * !n100 + 500 * !n500,
      :intiate.
end.
```

Example 13 *File Copy*

```
    #copier:new(^Copier),
    Copier:copy("a", "b"),

class  copier
     slot  inpath, outpath, infile, outfile.
  :copy(InPath, OutPath) ->
       !inpath = InPath,
       !outpath = OutPath,
       $system :get_file_manager(^FileManager),
       FileManager :create_stdin_file(!inpath, ^StdinFile, ^Status1)
           :create_stdout_file(!outpath, ^StdoutFile, ^Status2) ,
       !infile = StdinFile,
       !outfile = StdoutFile,
       :copy.
  :copy ->
       !infile :getc(^Char),
       (Char == eof) [
          :true ->
               !infile :,
               !outfile :,
               :terminate.
          :false ->
               !outfile :putc(Char),
               :copy.
       ].
  :terminate -> ..
end.

class  file_manager
  :create_stdin_file(^PathName, StdinFile, Status) ->
       #standard_input_file :new(^StdinFile),
       StdinFile :initialize(PathName, ^Status),
  :create_stdout_file(^PathName, StdoutFile, Status) ->
       #standard_output_file :new(^StdoutFile),
       StdoutFile :initialize(PathName, ^Status).
  :directory(^PathName, Directory, Status) ->
       #directory :new(^Directory),
       Directory :initialize(^PathName, ^Status).
end.

class  standard_input_file
     super  as_standard_file, input_file.
  :getc(Char) ->
       :get_position(^Position),
       :get_buffer_size(^BufferSize),
       (Position >= BufferSize) [
           :true ->
                :read_block(BufferSize, ^Buffer)
                :set_position(0).
       ],
       :get_position(^Position1),
       (Position1 >= BufferSize) [
           :true ->
                ^Char = eof.
           :false ->
```

```
                 :get_buffer(^Buffer)
                 :set_buffer(Buffer :string_element(Position1, ^Char))
                 :increment_position.
        ].
end.

class  standard_output_file
     super  as_standard_file, output_file.
  :putc(^Char) ->
       :get_position(^Position)
       :get_buffer_size(^BufferSize),
       (Position >= BufferSize) [
           :true ->
                :write_block(BufferSize, ^Buffer)
                :set_position(0).
           :false -> .
       ],
       :get_buffer(^Buffer)
       :set_buffer(Buffer)
       :set_string_element(Position, Char)
       :increment_position.
end.

class  as_standard_file
     slot  position, buffer_size, buffer.
  :intialize(^Pathname, Status)&before ->
       !position = 0,
       !buffer_size = 992 * 4,
       #string :new(^Buffer, [size(992 * 4)]),
       !buffer = Buffer .
  :get_position(Position) -> ^Position = !position.
  :set_position(^Position) -> !position = Position.
  :incremenet_position ->
      !position = !position + 1.
  :get_buffer(!buffer) -> .
  :set_buffer(^Buffer) -> !buffer = Buffer.
  :get_buffer_size(BufferSize) -> ^BufferSize = !buffer_size.
  :set_buffer_size(^BufferSize) -> !buffer_size = BufferSize.
end.

class  input_file
     super  as_file.
  :read_block(^Length, Buffer) ->
       !file_device :read(Length, ^Buffer, ^Status),
       Status [
          :normal ->
               :proceed_pointer(Length) ..
          :error(^ErrorReason) ->
               ErrorReason [
                  :eof -> % eof handling %.
               ].
       ].
  :seek(Position, Status) ->
       !file_device :seek(Position, ^Status),
       Status [
          :normal ->
          :error(^ErrorReason) ->
```

```
                ErrorReason [
                   :eof -> :close.
                ].
       ].
  :end_of_file(YorN) ->
       :get_pointer(^Pointer),
       (Pointer == eof) [
           :true -> ^YorN = yes.
           :false -> ^YorN = no.
       ].
  :close -> ..
end.

class  output_file
     super  as_file.
  :write_block(^Length, ^Buffer, Status) ->
       !file_device :write(Length, Buffer, ^Status),
       Status [
          :normal -> :proceed_pointer(Length).
          :error(^ErrorReason) -> :close.
       ].
  :close -> ..
end.

class  as_file
     super  as_retrievable_object.
     slot  file_device, pointer.
  :initialize(^PathName0, Status0)&after ->
       :get_pathname(^PathName),
       $system :open_binary_file(PathName, ^FileDev, ^Status),
       !file_device = FileDev.
  :get_file_device(!file_device) ->.
  :set_file_device(^FileDev) -> !file_device = FileDev.
  :get_pointer(!pointer) -> .
  :set_pointer(^Pointer) -> !pointer = Pointer.
  :proceed_pointer(^Delta) -> !pointer = !pointer + Delta.
  : ->
      !file_device : .
end.

class  as_retrievable_object
     slot  pathname.
  :initialize(^PathName, Status)&before ->
       !pathname = PathName,
       :get_status(^Status).
  :get_pathname(!pathname) -> .
  :set_pathname(^PathName) -> !pathname = PathName .
end.

class  file_device
     :read(^Length, ^Buffer, Status) ->  % request FEP to read %
     :write(^Length, ^Buffer, Status) -> % request FEP to write %
end.
```