TM-0664

A'UM = Stream + Object + Relation

by
K. Yoshida & T. Chikayama

January, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

$\mathcal{A}'\mathcal{UM}$ = Stream + Object + Relation

Kaoru Yoshida Takashi Chikayama

Institute for New Generation Computer Technology
4-28, Mita-1, Minato-ku, Tokyo 108, Japan,
e-mail: {yoshida, chikayama}%icot.jp@relay.cs.net

Abstract

In this paper, we introduce a concurrent programming language $\mathcal{A}'\mathcal{UM}$ [1][Yoshida88], which has been designed aiming at high parallelism and high expressivity for the development of large scale software. $\mathcal{A}'\mathcal{UM}$ is characterized by three features: stream-based computation, object-oriented abstraction and relational representation.

1 Introduction

Our goal is to realize computer systems by which large scale or complicated problems can be solved quickly in the total amount of time required for the entire development process through designing, programming, debugging, maintenance, and extension. To achieve this goal, we would like to extract maximum parallelism from the problems and impose minimum overhead on the system architecture. Since a programming language lies between application problems given by the user or environments facing to the user and the system architecture designed by the implementator, it should be natural and flexible in its representation and efficient in its performance. In addition, the underlying computation model should be simple and uniform for ease of understanding, formalization and implementation.

$\mathcal{A}'\mathcal{UM}$ satisfies all these requirements by integrating the notions of streams, objects and relations.

What is a concurrent system? A concurrent system is a system in which more than one event can occur independently, i.e., in parallel or in any order. This situation can be simply modelled using partially ordered sets of events (*poset* for short), which may contain elements that have no precedence in order. Posets can be easily represented by streams.

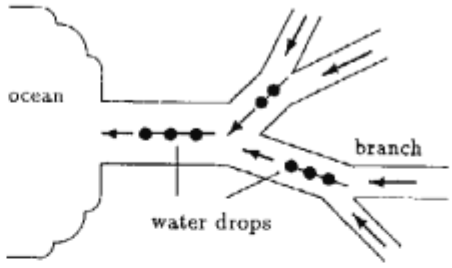
2 Stream Computation

In general, there are several kinds of streams, such as streams of water (rivers), streams of electricity (electric currents), and streams of cars (traffic). We visualize the notion of streams illustrating a river, which may be the Amazon flowing to the Atlantic Ocean.

What is a stream? Running in the river is water, which consists of drops of water. Drops connect one after another and finally make a continuous stream. Looking over the upper river area to the lower river area, every two branch streams converge into one stream so that a tree of streams is formed, whose root stream goes into the ocean. One drop from one branch stream may go to the ocean earlier or later than another in another branch stream. As the phrases: *upper river* and *lower river* show, the river has a direction of flow.

A stream also reminds us of connecting electric wires. For each wire, the end toward the anode is attached with a red tag and the other end from the cathode with a blue tag. We can make a long wire by connecting the red tag of one wire to the blue tag of another.

In $\mathcal{A}'\mathcal{UM}$, we deal with streams of messages. Let us regard the above ocean as an object and each drop of water as a message flowing into the object.



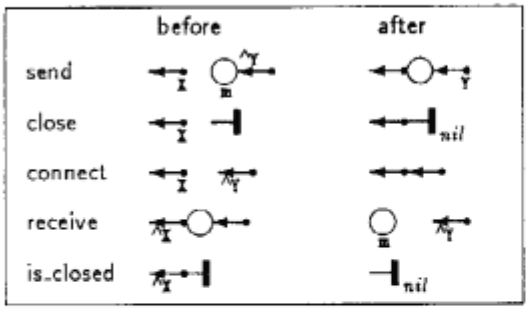
Stream computation: The direction of a stream is represented as an arrow with a complementary pair of terminals, *inlet* and *outlet*, which are respectively specified by a variable name preceded by ^ (e.g., `^X`) and just by a variable name (e.g., `X`).

Stream computation is production and consumption of streams, whose basic operations are:

- sending a message (`m`) to an outlet (`X`) when the outlet (`Y`) of the following stream is given;
- closing an outlet (`X`);
- connecting an inlet (`^Y`) to an outlet (`X`);
- receiving a message (`m`) from an inlet (`^X`) when the inlet (`^Y`) of the following stream is given;
- detecting that an inlet (`^X`) is closed.

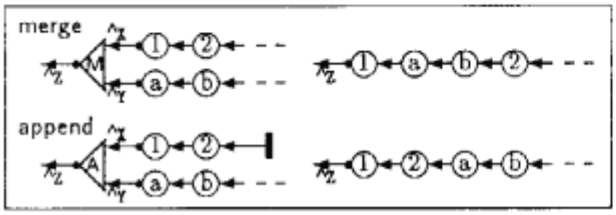
[1] $\mathcal{A}'\mathcal{UM}$ is a Japanese word, derived from the Sanskrit "ahum" consisting of *Ah* and *Um*, which implies the beginning and the end, an open voice and a close voice, and expiration and inspiration. This name was given to symbolize stream communication which is the basic notion of this language.

Thus, stream computation is explained using three terms, *inlet*, *outlet* and *nil*, where ***nil*** is the state left after closing an outlet, indicating that the stream is not longer accessible.

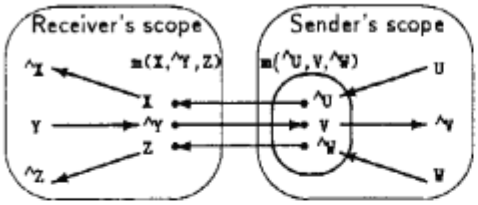


There are two kinds of binary operations for building a stream tree:

- merging messages from two streams (`^X` and `^Y`) nondeterministically;
- appending messages from one stream (`^Y`) to any of the messages from the other stream (`^X`).



What is a message? A message contains a message name and a tuple of stream terminals, each of which is either an inlet or an outlet, and is identified by the message name, the number of the terminals, and their directions from the receiver's viewpoint. For example, message `m(X,^Y,Z)` has a name `m` and contains three terminals: one inlet `^Y` and two inlets `X` and `Z`, so it is identified as `m(-+-)`.



Each message works as a stream connector which connects streams given as formal parameters in the sender's scope with streams given as actual parameters in the receiver's scope. Since two streams can be connected when the inlet of one and the outlet of the other are given, the sender and the receiver should specify complementary directions for each parameter.

3 Object Model

What is an object? An object (or process) is the abstraction of iterative computation. When being created, an object is given one stream, called the *interface stream*, which works as an interface to the outside. Every time an object detects an event on the interface stream, which is either receiving a message from the stream or detecting that the stream is closed, the object takes some actions according to the event and simultaneously descends to the next cycle of iteration. Taking each cycle as one generation of an object, an object is a chain of generations. For some generation of an object, the object *itself* means the next generation.

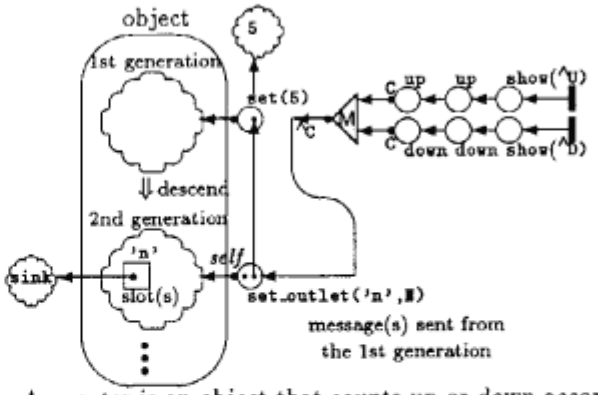
Each generation is a collection of stream terminals which include the inlet of its interface stream from which to receive a message and other stream terminals which represent the internal states of the object.

From the sender's viewpoint, each stream (outlet) toward an object can be looked upon as the object itself. *Making acquaintance with an object* is obtaining a stream (outlet) toward the object. *Introducing an aquaintance* (A) *to another* (B) is splitting the stream (outlet) toward A into two and passing one of the two streams to B.

In $\mathcal{A}'\mathcal{UM}$, there is nothing but objects which communicate with each other via streams. Primitive data such as integers, atoms and booleans, and classes are also objects. For example, if an integer such as `5` is specified, it means an outlet toward integer `5`, to which a message such as `add(Adder, ^Sum)` can be sent.

Example (counter): We explain the $\mathcal{A}'\mathcal{UM}$ object model with a simple example, *counter*, which is defined as follows:

```
class   counter.
    out  n.                 % outlet slot definition
 :up   -> !n + 1 = !n. % X+Y => S ; X:add(Y,^S)
 :down -> !n - 1 = !n. % X-Y => D ; X:sub(Y,^D)
 :set(^N) -> N = !n.    % $self:set_outlet(n,N)
 :show(N) -> !n = ^N.   % $self:get_outlet(n,^N)
end.
class   test_counter.
 :testM(U, D) ->
    #counter:set(5) = ^C, C:up:up:show(^U),
                          C:down:down:show(^D).
 :testA(U, D) ->
    #counter:set(5) = ^C, C$1:up:up:show(^U),
                          C$2:down:down:show(^D).
end.
```



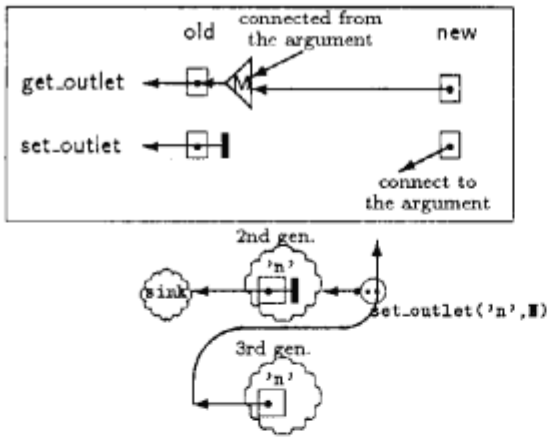
A counter is an object that counts up or down according to each received message. We try two kinds of test using the counter: `testM` and `testA`. Both methods are: (1) create an instance of class `counter`; (2) send message `set(5)` to the instance; (3) split the stream remaining after sending the `set` message into two; (4) send two up messages and one `show(^U)` message in a row to one branch; (5) send two `down` messages and one `show(^D)` message also in a row to the other branch. The only difference between the two test methods is that, in `testM`, messages from the two branches

are merged, while, in `testA`, messages are appended according to the given suffixes: `$1` prior to `$2`. Therefore, the counting results, `^U` and `^D` of `testM`, should be respectively some of `{5,6,7}` and some of `{3,4,5}`, while those of `testA` should be exactly 7 and 3.

The picture shows the situation just after the counter has received message `set(5)` in `testM`. For `testA`, an appending joint takes the place of the merging joint.

Relational model: Among those actions that each generation can take are stream operations, creation of objects, and generation descent. One of the characteristic of $\mathcal{A}'\mathcal{UM}$ is its declarativity. A program contains no constraints on the execution sequence on these actions, as follows:
(1) Parallel actions: The actions can be executed in parallel and their arguments can be connected in parallel.
(2) Causality: All that relates one action to another is causality that is the relationship of their arguments.
(3) Asynchronous communication: Objects do not wait in sending messages. They wait only when receiving messages at the beginning of each generation.

Slot access: An object may have *slots*, each of which holds a stream terminal associated with a name. There are two kinds of slots, *inlet slots* and *outlet slots*, according to the direction of the terminal that they hold. Slot access is done by sending a particular message to the object *itself*. The action taken for each outlet slot access messages and, based on the action, what will happen in the second generation of the counter object are shown below.



Class inheritance: $\mathcal{A}'\mathcal{UM}$ supports multiple class inheritance for the purpose of minimizing the total amount of program code. By inheriting classes, the applicable method space for an instance is expanded, but no extra instance is created for any of the super classes.

Volatile object: Each generation (1) takes actions according to the event and (2) descends to the next generation. (1) means conditioning and (2) means looping. Objects are condition handlers by nature. If one class were defined for each condition, however, a number of classes would have to be defined. $\mathcal{A}'\mathcal{UM}$ allows temporary classes to be defined within a method, which are called *volatile classes* but have the same framework as external classes.

4 Linguistic Support

As a concurrent system is modelled using a poset of events, a concurrent program is an event graph. Here, how to make it easy to draw a graph is the most important issue. Stream programming is direct, but it has the disadvantage that even one failure to close a stream or to connect a stream might cause a deadlock which will make the entire system fail. $\mathcal{A}'\mathcal{UM}$ provides linguistic support for safe and easy programming.

Functional grammar: A method consists of two parts: the *passive part* to define an event by which the object is activated, and the *active part* to define a set of actions which the object should take according to the event.

$<Method>$::=$<Event>$ '|'[$<Actions>${',' $<Actions>$}]
$<Actions>$::=$<Nil>$

The event and actions are defined by functional expressions, each of which is called *inlet expression*, *outlet expression* or *nil expression*, according to the representation result. Any complicated graph can be drawn by constructing these expressions. For example, the message-sending expression represents the outlet of the following stream, to which another message can be sent. Thus, `C:up:up:show(^U)` represents the outlet of the stream following the message `show(^U)`.

relation	*expression*	*result*
`receive(^I,m,Y)`	':' $<Message>$ '=' $<Out>$ `:m = Y`	$<Nil>$
`is_closed(^I)`	'::'	$<Nil>$
`send(I,m,^Y)`	$<Out>$ ':' $<Message>$ `I : m`	$<Out>$ `Y`
`close(I)`	$<Out>$ '::' `I ::`	$<Nil>$
`merge(^I,^Y,Z)`	$<Out>$ '=' $<In>$ `Z = ^I`	$<Out>$ `Y`
`append(^I,^Y,Z)`	$<In>$ '\' $<In>$ `^I \ ^Y`	$<In>$ `^Z`
`connect(I,^Y)`	$<Out>$ '=' $<In>$ '::' `I = ^Y ::`	$<Nil>$
`descend(^I,5)`	'<==' $<In>$ `<== ^I`	$<Nil>$

(1) Right-to-left rule: The above pictures are drawn so messages flow *from right to left* toward the leftmost object, i.e., time should go by from left to right, so that messages further left are received by the object earlier than ones further right. The expressions are designed to keep this *right-to-left* manner, so that we can write a program like drawing a picture.
(2) Derivation of nils: The grammar promotes the completion of computation. It basically allows only *nil expressions* to be specified as the top-level expressions in the $<Actions>$ field, so that no outlets can be left open and no inlets be left unconnected. This grammar is extended to be more flexible as explained later.

Macros: For ease of writing compact programs, several kinds of macro expressions are supported, among which the following three kinds of macros are used in the example:

(1) Arithmetic operation macros:
This type of macro means an outlet toward the operation result, e.g., `X+Y` means `S` for the sum `^S` of `X:add(Y,^S)`.

(2) Instance creation macro (`'#'`<*ClassName*>):
which means an outlet toward an instance which is created by sending an instance-creation message to the class object, e.g., `#counter` means `New` for the instance `^New` of `##counter:new(^New)`.

(3) Outlet slot access macro (`'!'`<*SlotName*>):
The meaning of this macro depends on the field: slot-reference in the < *Out* > field, and slot-updating in the < *In* > field. For example, methods `set(^N)` and `show(N)` are equivalent to:

```
:set(^N) -> :set_outlet(n, N).
:show(N) -> :get_outlet(n, ^N).
```

Note that, for those programmers who are used to be programming in procedural languages like C, it might seem strange that `!n = ^N` implies slot-reference and `N = !n` slot-updating, but they are consistent on the *right-to-left* rule.
Suppose that the counter receives messages `set(5)` and up. An outlet toward integer 5 is set into a new slot named n, then it is referred to and an addition message is sent to the referred outlet as follows:

```
5 = !n, !n = ^N, N:add(1, ^S),
```

Thus, the addition message will flow from right to left toward integer 5.

(4) Succession macro (< *Event* >`'->'`): The meaning is: (i) create a new generation; (ii) send to it the messages from the current generation, prior to those following the received message in the interface stream. For example, the above `set(^N)` method is equivalent to:

```
:set(^N) = Rest |
     <== ^Self, Self:set_outlet(n, N) = ^Rest.
```

Variable as a stream tree: For example, when making a call from Tokyo to someone in London, our intention is not to know how to connect telephone lines from Tokyo to London, but to talk with the person. Stream programming is like this. Merging and appending expressions using stream variables make us pay attention to how to build a stream tree toward an object rather than what to do with the object.

The meaning of a variable is extended from a stream to a stream tree so that a stream tree can be directly represented only by variable occurrences. For one inlet (e.g., `^C`), outlets without suffixes (e.g., `C`) make a merging tree, and outlets with suffixes (e.g., `C$1` and `C$2`) make an appending tree.

Implicit completion of streams: It is permissible to specify a non-nil expression in the < *Actions* > field or to leave inlets open or outlets unconnected, but instead, these inlets or outlets are implicitly closed or connected to sink objects which should absorb messages. In the example, all the top-level expressions are outlet expressions, so they are implicitly closed.

Therefore, method `testM` is expanded and completed as follows:

```
:testM(U, D) -> ##counter:new(^New)::,     %1
                New:set(5) = ^C ::,        %2
                C = ^C1 = ^C2 ::,          %3
                C1:up:up:show(^U)::,       %4
                C2:down:down:show(^D)::.   %5
```

For `testA`, line 3 is replaced by `C = (^C1 \ ^C2) ::.`

5 Related Works

What influenced most and has a close relationship with $\mathcal{A}'\mathcal{UM}$ is the family of concurrent logic languages (CLLs) and the Actor model.

$\mathcal{A}'\mathcal{UM}$ vs. CLLs: The declarative property of $\mathcal{A}'\mathcal{UM}$ is inherited from CLLs. Declarativity is one of the greatest advantages we can obtain in concurrent programming for extracting maximum parallelism and providing high expressivity, since it frees the programmer's mind from concern about the execution sequence.

Since an object-oriented programming style for CLLs was proposed, in which an object is represented as a sequence of tail recursive goals, each of which has an interface stream from which to receive a message and carries arguments as its internal states, there have been several object-oriented approaches studied based on this style. Attempting to program in this style revealed several substantial problems, mainly due to the verbosity of CLLs, that motivated us to design $\mathcal{A}'\mathcal{UM}$. $\mathcal{A}'\mathcal{UM}$ differs from these object-oriented approaches to CLLs in two points:
(1) there is no general (bi-directional) unification, that simplifies the system architecture;
(2) abstraction is infinite, i.e., primitive data can be seen as objects in the program, that promotes generic programming.

$\mathcal{A}'\mathcal{UM}$ vs. Actor: The object-oriented abstraction of $\mathcal{A}'\mathcal{UM}$ is based on the Actor model. $\mathcal{A}'\mathcal{UM}$ differs from the Actor model mainly in that the arrival order of messages can be explicitly represented using streams.

6 Current and Future Work

We have implemented a prototype system for $\mathcal{A}'\mathcal{UM}$ onto a concurrent logic language, KL1, which is inherently a subset of GHC. In this system, the parallel computation and communication mechanism is dependent on that of KL1, which is heavy for $\mathcal{A}'\mathcal{UM}$. We plan to develop an independent system which provides an adequate parallel and communication mechanism for $\mathcal{A}'\mathcal{UM}$ by itself.

References

[Yoshida88] Kaoru Yoshida and Takashi Chikayama: *$\mathcal{A}'\mathcal{UM}$ – A Stream-Based Concurrent Object-Oriented Language* –, Proc. of the International Conference on Fifth Generation Computer Systems '88, ICOT, November 1988