ICOT Technical Report: TR-0831

TR-0831

Programming in KL1 and AYA

by
K. Susaki & T. Chikayama

February, 1993

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

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Programming in KL1 and AYA

Kasumi Susaki Takashi Chikayama
Institute for New Generation Computer Technology
Mita Kokusai Bldg.21F
4-28, Mita 1-chome, Minato-ku, Tokyo 108 JAPAN
Phone: +81-3-3456-3193, Fax: +81-3-3456-1618
susaki@icot.or.jp

1 Introduction

KL1[1] is a concurrent logic programming language based on GHC[1] with extensions to write practical large scale systems. PIMOS[2], the operating system of Parallel Inference Machine is fully written in KL1 and is being used to run many large scale application programs written in KL1.

To write programs in KL1, the process oriented programming technique [3] is widely applicable and useful. Using this technique, a program is constructed with processes exchanging messages each other. Because a program is described as an aggregation of relatively independent processes, high level modularity of programs can be kept and parallelism can be controlled easily. PIMOS is designed with this model to prevent the bottle neck of central control; all the devices and other management functions are distributed.

In spite of these merits, processes are no more than a programming style and there is no support by the language itself. It causes many difficulties. For example, arguments of a KL1 goal that keep the states of a process should be written each time even in case of no modification. Adding a new state variable to the process often requires a large number of modifications to the program. It is also difficult to introduce a new function to the debugger for this style.

As a solution to these problems, a new language AYA has been designed upon KL1 that supports explicit descriptions of processes and communication between them.

In the following sections, the process oriented programming technique and its deficiencies are explained and one solution, AYA, is proposed.

AYA runs on PIMOS after translation to KL1. The first version of its compiler has just started to run on PIMOS.

2 Process oriented programming in KL1

The process oriented programming style is widely applicable to concurrent logic programming languages including KL1.

Processes are realized as tail recursive goals and communicate with each other using variables shared among them. Binding a value to such variables and inspecting them is the method of communication among them. Internal states of processes are kept in their non-shared variables and both shared and non-shared variables are kept as their arguments.

When a shared variable gets some value, the process waiting for it resumes its execution. The synchronization mechanism is realized by the guard of KL1 language.

Here is a short example of a KL1 program written in the process oriented way. This program describes a 'counter' process.

```
counter(In):- true |
        counting(In,0).
counting([],State):- true | true.
counting([up|In],State):- true |
        New := State + 1,
        counting(In,New).
counting([down|In],State):- true |
        New := State - 1,
        counting(In,New).
counting([show(Value)|In],State):- true |
        Value = State,
        counting(In,State).
```

This process has two arguments: one for communication and another for internal memory. The communication is done by a message stream implemented as a list. The process accepts three kinds of messages, up, down, show(Current) which increments and decrements its internal memory and

obtains the current value, respectively. When the stream is closed, the process terminates execution.

PIMOS is also written in this programming style and has the following characteristics.

- I/O devices and their management mechanisms are modeled as processes. They are working each other exchanging messages via streams.
- Not only the streams for normal communication but also simple shared variables are used for asynchronous communication for interrupts.
- A device in one of several states, initiation, regular transaction, aborting and so on, and changes its state among them. Sometimes different states have a different set of state variables.

Ease in maintenance and extension is important.

Presently the program of PIMOS contains predicates with many arguments and there are many similar predicates with only slight differences in their state variables. It causes difficulties in maintaining and extending the programs, as follows:

- Errors in positions and names of arguments are often made.
- Adding or removing arguments has to be done for all predicates with common state variables.
- New variable names are required for each state change.
- Because the tail recursive goal has no difference from other body goals, the process structure is not explicit, making it harder to understand.
- Though programs are written in this style, the debugger only shows execution of predicates rather than processes because it has no information about processes.

AYA, a language upon KL1, is one solution to these problems.

3 AYA

To describe a process which has multiple communication paths and changes its state, AYA has several dedicated features.

A process is a component of this language and a class is a unit of describing a process. A class can have several 'scenes' that correspond to states of a process.

In addition to these, sockets are introduced to reduce the verbosity required to maintain the generation of arguments and variables. Brief notation of stream communication is also introduced. In this section, these characteristics of AYA will be described.

The following example is the previous 'counter' program rewritten in AYA.

```
class counter(In)
    with +in := In , +state := 0.
input in.
 :up -> @state := ~(@state + 1).
 :down -> @state := ~(@state -1).
 :show(Value) -> Value <- @state.
 :/ -> continue \\ .
end class.
```

From 'class' to 'end class' is a process definition. 'counter(In)' after the keyword 'class' gives the name of this class and the parameter that is passed when invokes this process.

'with' is also a keyword that starts socket definitions. This process defines two sockets; 'in' is for communication and 'state' is used as internal memory. The parameter 'In' and the constant '0' are set to each socket as initial values. In this definition, 'In' is a shared variable and is used for communication between processes. In AYA, such a variable is called a 'line'.

`:up`, `:down`, `:show(State)` are stream type messages. The last message `:/` is a 'close message' and it tells the end of this stream. Each message has a corresponding procedure called a 'method'.

A process repeats its execution when no assignment of next scene is given. To terminate the execution of a process, it is necessary to assign the language defined scene 'termination' and it is written '\\'. The 'counter' process will terminate when the close message arrives.

3.1 Class and scene

Every 'class' has a 'scene' called the 'initial scene'. The class definition is equivalent to the definition of this scene and is accessed by its class name. Inside this scene, arbitrary numbers of scenes are defined. Scenes can be nested at arbitrary levels.

The relation between a class and its scenes are showen in the Figure 1.

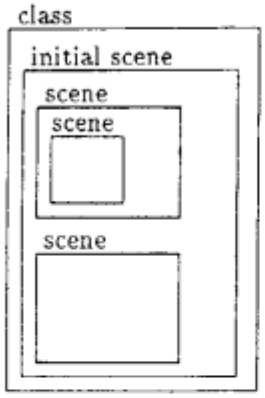

Figure 1: Class and scenes

The process 'calculator' is a short example that has three scenes inside it.

```
class calculator(In)
    with +in := In
        \\ waiting.

scene waiting.
input in.
 :/ -> \\ .
 :sum -> continue \\ adding.
 :product -> continue \\ multiplying.
end scene.

scene adding
    with +sum := 0.
input in.
 :result(Result) ->
        Result <- @sum
        \\ waiting.
 :N -> integer(N) |
        @sum := ~(@sum + N).
end scene.

scene multiplying
    with +product := 1.
input in.
 :result(Result) ->
        Result <- @product
        \\ waiting.
 :N -> integer(N) |
        @product := ~(@product * N).
end scene.
end class.
```

The class 'calculator' contains three scenes: 'waiting', 'adding' and 'multipling'. After initiation, it changes its state to 'waiting'. In this scene, the process waits for the message `:sum` or `:product` and moves to the corresponding scene. In adding, it sums up the integers that arrive until `:result(Result)` is received, at which point it returns the result to the argument of `:result(Result)` message. Then the process changes its scene to 'waiting' and waits for the message again. In multipyling, it multiplies the integers that arrive.

A class is composed of scenes in this way. Each scene corresponds to a state of the process. The variations of acceptable messages are defined independently by each scene.

3.2 Socket

3.2.1 Socket

Sockets are holders of processes. There are two types of socket, input and output. A socket for input is used for waiting arrival of messages. Instantiation is done to the socket for output. Input and output are called the modes of sockets.

When an input socket and an output socket share a variable, binding a value to the output socket and getting the value from the input socket realize communication. In AYA, a variable is called a 'line' and the data communicated is called a 'message'. Any data can be used as a message. Lines in sockets are accessed by their socket names.

Sockets should be declared at the top of scene definitions after the keyword 'with'. The socket is declared with its name, mode and optional initial value. Without the initial value, '[ ]' is set to input sockets and '_', an anonymous variable, is set to output sockets.

A socket can be accessed from inside the declared scene. When the scene has internal scenes, sockets can also be accessed from them and scenes can share them.

In the calculator example, a socket 'in' is declared with its initial value at class definition and is accessed from its internal scenes.

'sum' is declared at the scene 'adding' and 'product' is declared at the scene 'multiplying'. They also have initial values. They are accessed only from the declared scenes.

Sockets can be updated using the operator ':='. In the previous 'calculator' example, the arrived integer 'N' is added to the value of socket 'sum'.

```
@sum := @sum + 1
```

Updating and reading operations are repetitively used for a socket to realize internal memory.

The value of output socket is bound to [ ] automatically when it is abandoned. This is done to avoid unexpected deadlock.

3.2.2 Term

Instantiated terms are also regarded as lines that already have values. Terms also have modes. Terms that can be kept in input sockets are input terms and so on.

Lines that have the same name within a method are identical lines.

3.2.3 Unification

The operation of connecting two lines is called 'unification' and is written as follows.

```
Out <- In
```

The left hand side of <- should be an output term and the right hand side should be an input term.

Message sending via a line kept in a socket is also done by unification. Sending a message via sockets and referring to the value of sockets are also written as unifications.

```
@out <- end
Amount <- @amount
```

3.3 Method definition

Operations to be taken on message arrival to a socket are defined by a combination of the socket name and the message. This is called a 'method definition'.

A method is defined in four parts, event, condition, action and next scene assignment. When a socket is declared as a 'base socket', message arrivals at this socket is waited as an event.

Here is an example from the previous 'calculator' program.

```
input in.
 :result(Result) ->
        Result <- @sum
        \\ waiting.
 :N -> integer(N) |
        @sum := @sum + N.
```

The first line is a declaration of the basic socket, which is the default socket for message arrival. `:result(Result)` and `:N` represent the arrival of these messages at the basic socket that is called an event. In case of `:N`, N should be an integer and should be examined in the condition part. When both an event and conditions are satisfied, the procedure written in the action part is executed. In this example, '`Result <- @sum` ' and '`@sum := @sum + N`' are actions. A process can change its scene after its execution, in case of :result(Result), the process changes its scene to 'waiting'.

An event and conditions can be omitted. In case of omitting an event, conditions are examined directly. When both are omitted, the action part is executed directly. Put only the keyword 'continue' when there is nothing to execute in the action part. If there is no declaration of next scene, the current scene is repeated. To terminate the process, the language defined scene 'terminate' should be assigned as the next scene.

A socket can be accessed arbitrarily many times in a method and is updated in the order of appearances.

3.4 Stream notation

To send a sequence of messages, stream communication implemented by a list structure is frequently used. In this way, a list cell is a unit of sending or receiving; a message is put in the car part and the variable that is used for the next communication is put in the cdr part. To terminate the communication, [ ] is used as the close message.

To support this technique, AYA provides special notation.

The following is a rewrite of a method that appeared in the previous example 'counter' using a list cell notation.

```
[up|In] -> @state := ~(@state +1 ),
        @in := In.
```

In the stream notation, ':' is prefixed to a message to indicate it and :/ indicates the termination of stream called the 'close message'. The above method can be written in the following way using this notation.

```
:up -> @state := ~(@state+1).
```

The cdr is set to the socket automatically in this syntax.

Message sending is written as follows and the cdr is automatically set to the socket in this case also.

```
@out <<= :up
```

Sending a sequence of messages can be written in the following way.

```
@out <<= :up:down:show(Current)
```

In this case, three messages are sent and the cdr is set to the socket 'out'. Messages can also be sent to an input socket to be received it later. This has the effect of specifying a continuation method.

```
@in <<= :up
```

The message :up is received next and is executed. To close a stream, a unification with a close message should be done.

```
@out <- :/
```

4 Implementation

AYA runs on PIMOS after compilation to KL1. The compiled program is similar to the programs written in process oriented programming technique.

Sockets become arguments and lines become variables in KL1 predicates. The synchronization mechanism of KL1 is used for waiting messages. Process invocations and scene changes become body goals.

Unification failures and deadlocks are bugs hard to debug in KL1 programs. In AYA, sockets and variables are examined during compilation using mode information making it possible to find many of such bugs.

Moreover, the consistency between the argument modes of next scene declarations and process invocations can also be examined.

5 Future work

5.1 Syntax revision

As repetition of the same scene is the default when no assignment of the next scene is made, a careless omission of it may easily cause infinite loops.

The stream notation, especially the difference between unification and stream type message sending, is hard to understand. In addition getting and using cdr explicitly is not possible with this syntax.

Though the mode system has been introduced to find bugs, the usage of sockets as internal memories are difficult to understand.

Furthermore, after experiencing programming, modification of syntax for better readability should be required.

5.2 Inheritance

The inheritance mechanism is necessary to share program codes. Though it is desirable to share the method definitions of multiple classes, it is difficult to specify such sharing because classes can have several scenes and scenes also have several input sockets. Translation of KL1 might also become difficult.

One solution under consideration is defining a table of methods for each socket and sharing it. A table can be inherited from other tables. The basic scheme has already been designed but the syntax is yet to be designed.

5.3 Debugging environment

The debugging environment specifically designed for AYA has not been prepared yet and programs have to be debugged as the compiled KL1 program. For ease of programming in AYA, it is necessary to provide a debugging environment that utilizes the concepts of processes and sockets directly.

6 Conclusion

For ease of programming in KL1, a new higher level language AYA has been designed and implemented. The compiler to KL1 has just started to run on PIMOS.

Vulcan[4], FLENG++[5], Polka[6] are research with similar objectives. AYA differs from these in the following characteristics.

- The concept of 'scene'.
- Communication based on variables, not confined to streams.
- No syntactical differences between sockets for communication and sockets for internal memory.

References

[1] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 1990.

[2] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference on Fifth Generation Computing Systems 1988*, Tokyo, Japan, 1988.

[3] E.Y. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing, OHMSHA Ltd. and Springer-Verlag*, 1(1):25–48, 1983.

[4] K. Kahn et al. Vulcan: Logical Concurrent Objects. In *Research Directions in Object-Oriented Programming*, Cambridge, Massachusetts, 1987. MIT Press.

[5] H. Nakamura et al. Object oriented language FLENG++ on concurrent logic programming language FLENG. In *WOOC'89 (in Japanese)*, 1989.

[6] A. Davison. Polka : A Parlog Object Oriented Language. Ph.d thesis, Imperial College, 1989.