TR-652

KL1上の並列プロセス指向言語　AYA

寿崎　かすみ、近山　隆

May, 1991

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

KL1 上の並列プロセス指向言語 $\mathcal{AYA}$

寿崎 かすみ　近山 隆
(財) 新世代コンピュータ技術開発機構
三田国際ビル 21 階
東京都港区三田 1 丁目 4-28
03-3456-3193, susaki@icot.or.jp

概要

プロセス指向プログラミングは Shapiro and Takeuchi [3] の論文で提唱された手法で, KL1 のような並列論理型言語で幅広く使用されている.

'プロセス' の概念を用いたモデル化はプログラムを構成するうえでのエレガントで強力な方法論である. この方法では, 再帰呼びだしにより繰り返されるゴールの実行をプロセスとみなし, そのゴールの引数をプロセスの状態とみなす. しかし, これはプログラムを解釈する方法にすぎずプログラミング言語としてサポートされていない. このため, 実際のプログラミングは煩雑になる. たとえばプロセスの状態を表す引数はすべて, 毎回述語のヘッド部および再帰呼びだしのゴールに記述しなくてはならない. また, プロセスの状態を表す引数を 1 つ増やすことは, プロセスの概念上ではささいな変更であるが, 実際のプログラムにおいてはほとんどすべての述語の引数を 1 つずつ増やすという煩雑な変更になり, この修正は新しいバグがはいる原因となる.

プログラミング言語 $\mathcal{AYA}$ はこのような問題を解決するために設計したものである. $\mathcal{AYA}$ のプログラムは KL1 のプログラムに簡単にコンパイルできるが, 状態をもったプロセスの概念を言語の基本要素として提供しているので, 前述したようなプログラミング上のささいな問題にわずらわされずに負荷分散の方法などもっと大局的な問題に集中することができる.

1　はじめに

プログラミング言語 KL1[1] は GHC(Guarded Horn Clause) に, 実用的な大規模システムを記述するための拡張を行った並列論理型言語である. 並列推論マシンのオペレーティング・システム PIMOS[2] は全面的に KL1 で記述している. この上で動作するアプリケーション・プログラムもまた KL1 で記述している.

プロセス指向プログラミングの手法は, KL1 プログラミングで広く使われているテクニックである. この方法では, 相互にメッセージを交換しながら並列に動作する協調的なプロセスとして問題をモデル化しプログラムを記述する. これによりプログラムのモジュラリティを高く保つことができる. また, 並列性の制御が容易に行える. PIMOS においては, このモデルをはじめから設計に取り入れている. たとえば, PIMOS の管理プロセス (e.g. デバイス・プロセス) は集中管理を行うことによりおこるボトルネックを避けるために分散している.

しかし, このテクニックは単なるプログラミング・テクニックにすぎず, このモデルに基づいてプログラムを直接記述するための言語上のサポートはない. このためプログラミングをする上での煩雑さがある. たとえば, プロセスの状態を表す KL1 述語の引数を毎回記述しなくてはならない. また, 新しい引数を加えるためにはすべての述語を修正しなくてはならないなどである. また, プロセスの概念をデバッガに反映しにくいという問題もある.

$\mathcal{AYA}$ は, これらの煩雑さや問題を軽減することを目的としている. このため $\mathcal{AYA}$ では, プロセスおよびプロセス間の通信を言語の構成要素として, そのまま記述することができるようにした. 次章以降で, 言語・文法および実装の方針について述べる.

2 言語仕様

2.1 特徴

KL1のプロセス指向プログラミングの手法では，プロセスは再帰呼びだしを行うゴールで実現していた．しかし，*AYA*ではプロセスを言語の構成要素として記述できるようにした．プロセスはまたプログラムの実行の単位でもある．*AYA*のプログラムは，クラスを単位として記述する．

*AYA*で記述した簡単なプログラムを次に示す．

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

このプログラムはカウンタのプロセスのプログラムである．このプロセスは，counterという名前のクラスで定義されている．このプロセスの起動は，プロセスがメッセージを受けとるためのストリームInを渡して行う．カウンタのプロセスは，内部状態としてそのときのカウンタの値を保持するために状態変数'state'を持つ．この変数ははじめ0に初期化されている．:up, :down, :show(Current), :/が，このプロセスの受信することのできるメッセージである．各メッセージに対して，そのメッセージを受けとったときに行う動作が定義してある．このプロセスは:upを受けとると状態変数stateの値を1増やし，:downを受けとると1減らす．:show(Current)を受けとるとそのときの値をメッセージの引数'Current'に返す．:/と記述してあるのは，closeメッセージである．これを受信するとプロセスは実行を終了する．受信メッセージとそれに対応した処理の定義をメソッドと呼ぶ．

このプログラムと同じものをKL1で記述した例を次に示す．

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

2.2 クラス・シーン

クラスは，その内部にシーンを持つ．シーンは任意段数ネストできる．

すべてのクラスが必ず持つシーンとしてinitial sceneがある．その他のシーンはこのinitial sceneにネストしたシーンとして定義する．この，クラス・initial scene・シーンの関係を図1に示す．

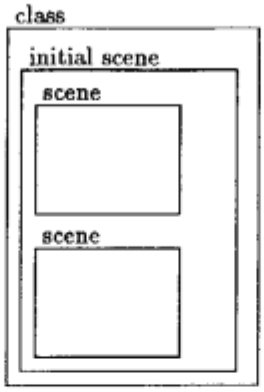

図 1: クラスとシーン

クラスおよびシーンを持つプログラムの例として, :start メッセージを受信してから :stop(Amount) メッセージを受信するまでに到着した整数を足し算し, その結果を引数 Amount に返すプログラムを示す.

```
class sigma(In)
    with +in := In, +amount := 0
    \\ wait_start.

scene wait_start.
input in.
 :start -> continue \\ adding.
 :/ -> \\ .
end scene.

scene adding.
input in.
 :stop(Amount) -> Amount <- @amount, @amount := 0
        \\ wait_start.
 :N -> integer(N) | @amount := @amount + N.
end scene.
end class.
```

クラス sigma は initial scene の中に, wait_start と adding の 2 つのシーンを持つ. このプロセスは:start メッセージを受けとるとシーン adding に移動する. :stop(Amount) が到着するまでのあいだに受けとった整数の和を求め, :stop(Amount) を受けとるとそれまでの合計を引数にかえし, 再び:start が来るのを待つ.

このように, クラスはシーンで構成する. 各シーンはそのクラスで定義したプロセスの状態 (この例では, start メッセージの待ち状態と到着する整数を足していく状態の 2 つ) に対応する. 受けとり可能なメッセージはこの例のように, シーンごとに異なって定義できる.

このようにシーンという概念を導入したことにより, 良く似たクラスを何度も生成したり終了したりするオーバーヘッドを軽減することができる.

2.3 メッセージ・ライン・ソケット

プロセス間の通信に使用するデータをメッセージと呼ぶ. メッセージとして, 具体化された任意の KL1 データを使用できる.

メッセージは送り手から受け手へラインを介して運ばれる. *AYA* のラインは KL1 の変数に対応する. プロセスは, ラインを保持するためのホルダーとしてソケットを持つ. ソケットに保持しているラインにはソケット名でアクセスする.

ソケットには入力用と出力用がある. 入力用ソケットはメッセージの受信のために使用し, 出力用ソケットはメッセージの送信に使用する. メッセージは出力用ソケットから送信し, そのソケットとラインを共有している入力用ソ

ケットで受信する．プロセスは複数の入力用ソケットを持ち，そのそれぞれに対してメソッドを定義することができる．このため，あるソケットに到着するメッセージ列の処理の途中でも他のソケットに到着したメッセージつまり別のルートからのメッセージを受け付けることができる．このことを利用して割り込み処理の記述が行える．

ラインはメッセージを１つだけ運ぶことができる．１つのソケットでメッセージ列の送受信を行うためにはメッセージを１つ送受信するたびに，送信・受信の両方が新しいラインをセットする必要がある．そこでメッセージ列のためには，リストセルに送りたいメッセージと次に使う新しいラインをつめて，１つのメッセージとして送るという方法を用いる．

この方法を用いると，前述の counter のプログラムは次のようになる．

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

はじめのメッセージ，[up|In] に対する処理を見てみよう．

```
@state := ~(@state + 1), @in := In
```

ここでは，ソケット state の値を１増やし，ソケット in にリストセルの Cdr として運ばれてきた次のライン In をセットしている．一般にソケットの値の更新は，':=' で行う．

このように，リストセルにメッセージと次に使用するラインをつめて１つのメッセージとしてやりとりする方法をストリームと呼ぶ．このストリームの用法を記述するために，[up |In] を :up のように記述するシンタックスを提供している．ストリームによる通信を終了するために，close message を使用する．close message を送ることを ' ストリームを閉じる ' といい，受けとることを ' ストリームを閉じられた ' という．close message として [ ] を使用する．

ソケットは，クラスおよびシーン定義のはじめのところでソケット名とソケットのモード (入力か出力か) を指定して定義する．併せて初期値を指定することもできる．初期値の既定値は [ ] である．シーンはすべてのクラスが必ず持っている 'initial scene' の中に定義されているので，クラスで定義したソケットはそのクラスで定義されているすべてのシーンからアクセスできる．

2.4 ターム・変数・ユニフィケーション

具体化された KL1 のタームは，その値を持ったラインのことである．変数は，ラインを意味する．同じ変数は同じラインを表す．変数のスコープはたとえばメソッドの中である．

２つのラインを結合する処理をユニフィケーションと呼ぶ．ユニフィケーションは次のように記述する．

```
Out <- In
```

２つのライン Out と In が結合される． counter の例では，ユニフィケーションを用いてそのときの状態変数の値をメッセージ :show(Current) の引数 Current に返している．

```
Current <- @state
```

ソケットからのメッセージの送信も，ユニフィケーションを用いて記述する．ソケット out からメッセージ end を送信することを次のように記述する．

```
@out <- end
```

2.5 メソッド定義

シーンは複数の入力用ソケットを持つことができるので，受信したメッセージに対する処理はメッセージの到着したソケットの名前とメッセージの組に対して定義する．これを，メソッド定義と呼ぶ．

カウンタのプログラムの例を少し変更して，:down を受けとったときの処理が状態変数 state の値によって２通りに定義する．

```
input in.
 :down -> @state > 0 | @state := ~(@state - 1).
 :down -> @state = 0 | continue.
```

上の例の,

```
input in.
```

を基本ソケット宣言と呼ぶ. これは, メッセージの到着するソケットを宣言している.

```
:down
```

は, ソケット in にメッセージ:down が到着するというイベントのことである. つぎの

```
@state > 0
```

は条件チェックである. これをコンディションという. メッセージが基本ソケットに到着してパタンマッチに成功し, コンディションを確認して満たされていれば対応して定義されている処理を実行する. この処理をアクションという. ここでは

```
@state := ~(@state - 1)
```

および

```
continue
```

がそれぞれアクションである. 'continue' は何もしないこと (KL1 のボディの true) を意味する.

このあと必要に応じて, 他のシーンに動くことを指定できる. たとえば, sigma の例では,

```
 :start -> continue \\ adding.
```

のように記述して:start メッセージを受けとった場合は, シーン adding に移動するよう指定している. これを, ' つぎのシーンの指定 ' と呼ぶ.

このようにメソッドは, イベント, コンディション, アクション, 次のシーンの指定により定義される. イベントおよびコンディションは省略することができる. イベントが省略された場合は, 直接コンディションを調べる. コンディションも省略されているときはアクションがいきなり実行される. イベントは省略されてコンディションが記述されているときは, コンディションを調べ, 条件を満たしていればアクションを実行する. アクションとしてなにも行わないときは continue と記述する. 次のシーンの指定は, 他のシーンに移動するときのみ記述する. プロセスの実行を終了する場合は, 言語定義のシーン terminate へ移動する.

1 つのソケットへのアクセスが 1 つのメソッド中に複数回出現する場合は, ソケットは出現順序に従って更新される.

3 ストリームのサポート

3.1 ストリーム型メッセージ

リストセルに次に使うラインも詰めて 1 つのメッセージとして扱うものを, ストリーム型メッセージと呼ぶことにする.

ストリーム型のメッセージのリストセルを毎回記述するかわりに, リストの Car としてセットするメッセージの前に ':' をつけることにする.

```
:up
```

は, up がストリーム型のメッセージであることを示す. このシンタックスを用いるとストリーム型メッセージの列はつぎのように記述できる.

```
:up:down:show(Current)
```

ストリームを閉じる close message は, つぎのように書く.

```
:/
```

3.2 メッセージの送受信

メッセージの送受信もストリーム型メッセージのシンタックスを用いてそれぞれつぎのように記述できる.

- メッセージ送信.
  メッセージの送信は, ユニフィケーションを用いて次のように記述する.

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

  これは, ソケット out からメッセージ up, down, show(Current) を送信してストリームを閉じることを意味する. 変数で表されるラインに対しても同様に

  ```
  Out <- :up:down:show(Current)
  ```

  のように記述する. これは, 変数 Out で示されるラインにストリーム型のメッセージを 3 つ送ってストリームを閉じることである.

  close message のみの送信も同様に

  ```
  @out <- :/

  Out <- :/
  ```

  などと記述する.

- メッセージ受信.
  - イベントの場合.
    基本ソケットに到着したメッセージとのパタンマッチを行う. :up に 続く要素がソケットにセットされる.

    ```
    :up
    ```

    close message の場合は次のようになる.

    ```
    :/
    ```

  - コンディションの場合.
    到着したメッセージとのパタンマッチを行う. :interrupt につづく要素が ソケットにセットされる.

    ```
    @in2 = :interrupt
    ```

  close message の場合は次のようになる.

  ```
  @in2 = :/
  ```

3.3 ストリーム操作

複数本のストリームをマージして 1 本にする, 2 本のストリームをつないで 1 本にするといった操作を行うために, merge および concatenate プロセスがある.

- **merge(In,Out)**

  merge プロセスは, ストリームのマージャとして機能する. In が入力ストリーム, Out が出力ストリームである. In にベクタが渡されると, その要素が入力ストリームとして加えられる. 入力ストリームの 1 本とベクタのユニフィケーションを ' マージイン ' と呼ぶ. マージインをつぎのように書く.

  ```
  @out <= Out2
  ```

  これは, 現在ソケット out に入っているラインと {Out1,Out2} をユニフィケーションして, Out1 をソケット out の新しい値としてセットすることである.

  マージインは, 入力・出力どちらのソケットにも行える.

- `concatenate(Stream1,Stream2,Stream3)`

  concatenate プロセスは，ソケットに保持されているストリームともう1本のストリームを結合し，その結果をソケットの新しい値とする．この2本のストリームのどちらを前にするかにより2通りのつなぎかたがある．

  - ソケットに入っているストリームを前にする．

    ```
    concatenate(@socket,Stream,New)
    ```

    この場合をアペンドと呼び，つぎのように記述する．

    ```
    @socket <=< Stream
    ```

  - ソケットに入っているストリームを後ろにする．

    ```
    concatenate(Stream,@socket,New)
    ```

    この場合をプリペンドと呼び，つぎのように記述する．

    ```
    @socket <<= Stream
    ```

アペンドおよびプリペンド操作は，入力・出力どちらのソケットに対しても行える．

入力ソケットに対するアペンドは，次に読み込むメッセージ列を指定することである．出力ソケットにアペンドすると，そのソケットから送信したメッセージは一旦 concatenate プロセスに送られ，その後，先につながるプロセスに届けられることになる．

入力ソケットに対してプリペンドするとプリペンドされたストリーム中のメッセージが，そのソケットから次に読み込まれるメッセージとなる．出力ソケットに対するプリペンドは，ストリームを共有する入力ソケットにそのストリームを渡すことになるので，ストリーム中のメッセージを送り届けることになる．このようなことから，プリペンドによりメッセージの送信を行うことができる．プリペンドによるメッセージの送信はユニフィケーションの場合と異なり，メッセージを送信したあとストリームを閉じない．

4 実装

AYA のプログラムは，KL1 プログラムにコンパイルして実行する．コンパイル結果の KL1 プログラムはプロセス指向のプログラミング・テクニックで記述したプログラムに近いものになる．

ソケットおよびラインは KL1 述語の引数に展開する．メッセージの待ち合わせは，KL1 の同期のメカニズムで実現する．他のプロセスの初期化と次のシーンへの移動は同じボディ・ゴールとなる．

AYA では，ソケットに対して入力あるいは出力のモードを定義した．また言語定義プロセスの引数には，入力あるいは出力のモードが定まっているものもある．これらの情報を使用して，モードの定義されていない変数のモードを定め，その整合性を調べることができる．これによりバグのもととなる変数を見つけることができる．

5 今後の課題

最大の課題は，継承機構の導入である．継承機構としては，複数のクラス間でメソッド定義を共有できるようなものが望ましい．しかし *AYA* では，クラスの下にシーンという概念を導入したこと，入力として複数のソケットを扱えるようにしたことから，共有したい定義がどれであるかを特定することが難しい．また KL1 にコンパイルして実行するため，実現上の制約もある．

このような条件のもとで継承機構を導入するための1つの方法として，ある入力ソケットに対するメソッド定義を独立したテーブルに定義し，このテーブルを共有するという方式を検討している．このテーブル間での継承も行える．基本的な方式はほぼ決定しているが，このための文法などの検討は，今後行う必要がある．

このほか，実際の大規模プログラムを書くにあたってはデバッグ環境の整備も必要となる．また，文法の見直しも必要となるかもしれない．

6 おわりに

KL1 のプロセス指向プログラミングの手法をサポートするために，高水準言語 *AYA* を設計した．

KL1 ではプロセスの内部状態を表す変数の記述が煩雑であったが，これをソケットとして必要なときだけ記述することにした．また，再帰呼びだしにより永続するプロセスを基本としてこのためのゴール呼びだしを毎回記述する手間をなくした．このようなことから，*AYA* では KL1 に比べて簡明な記述が可能となった．

シーンの概念を導入したことによりメッセージごとに定義されるプロセスの動作をシーンを移ることにより変えることができるようになった．また，受信可能なメッセージそのもの，受信対象とするソケットも変えることができる．1つのシーンが複数のソケットを入力として用意できるので，あるソケットからメッセージ列を読み込んでいる間に，他のソケットに到着した，つまり他の経路からのメッセージを処理することもできる．

類似した研究として Vulcan[4], FLENG++[5], Polka[6] などがある．いずれも同じような動機にもとづいて同様のアプローチにより設計された言語である．これらとの比較において *AYA* の特徴はつぎのような点である．

- シーンの概念を導入したこと．
- 通信はラインにより行う．

参考文献

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

A　プログラム例

AYA によるプログラムの例としてリーダーズ・ライターズ問題のプログラムを示す．この問題は共有資源アクセスの管理を扱ったもので，次のようなアルゴリズムに基づいている．

- 読みだし要求は，同時にいくつでも扱える．
- 書き込み操作が行われている間は，他の要求は一切受け付けない．
- 読みだし操作の間に書き込み要求がくると，その後あらたな読みだし要求は受け付けない．現在実行中の読みだし操作がすべて終了するのを待って書き込み操作を行う．

次のプログラムは，ファイルへの読みだし・書き込み要求を管理するプロセスである．プロセスは，readerswriters という名前のクラスとして定義されている．このプロセスは起動時に，ファイルへの要求がくるストリーム request，このプロセスからファイルデバイスへ要求を送るストリーム tofile，ファイルから管理プロセスへ処理の終了などを伝えるメッセージの流れるストリーム fromfile の3本を受けとる．

クラス readerswriters は，idling, reading, writing の3つのシーンを持つ．それぞれ，メッセージの到着待ち，読み込み処理，書き込み処理を行うシーンである．プロセスを起動するとまず idling のシーンでメッセージの到着を待つ．読み込み要求ならば reading に，書き込み要求ならば writing に移る．reading のシーンで読み込み要求を受けとるとそれはただちにファイルデバイスに送られる．書き込み要求の場合は reading の中で定義しているシーン waiting に移動し，そのとき実行されている読みだし処理がすべて終了するのを待つ．その後 writing に移って書き込み要求をファイルデバイスに送る．読みだし操作，書き込み操作が終了すると，idling に移動して次の要求を待つ．

```
module readerswriters.
public readerswriters/3.

class readerswriters(+request,-tofile,+fromfile)
```

```
\\ idling.

scene idling.
  input request.
    :read(Data) -> @tofile <<= :read(Data) \\ reading.
    :write(Data) -> @tofile <<= :write(Data) \\ writing.
end scene. % idling

scene reading
      with readers := 1.
    -> @readers = 0 | continue \\ idling.
  input request.
    :read(Data) -> @readers := ~(@readers + 1),
   @tofile <<= :read(Data).
    :write(Data) -> continue \\ waiting(Data).
  input fromfile.
    :readend -> @readers := ~(@readers - 1).

  scene waiting(+writedata).
      -> @readers = 0 | @tofile <<= :write(@writedata) \\ writing.
    input fromfile.
      :readend -> @readers := ~(@readers - 1).
  end scene. % waiting

end scene. % reading

scene writing.
  input fromfile.
    :writeend -> continue \\ idling.
end scene. %writing

end class. % readerswriters
```