ICOT Technical Report: TR-635

TR-635

ＫＬ１上の並列オブジェクト指向言語 ＡＹＡ（綾）の設計

寿崎　かすみ、近山　隆

March, 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 上の並列オブジェクト指向言語 $\mathcal{AYA}$(綾) の設計

寿崎かすみ　近山隆
新世代コンピュータ技術開発機構
東京都港区三田 1ー 4ー 28 三田国際ビルディング 21 階
03-3456-3193, susaki@icot.or.jp

概要

$\mathcal{AYA}$(綾) は, 並列論理型言語 KL1 の上のオブジェクト指向言語である. $\mathcal{AYA}$ は KL1 を始めとする並列論理型言語で広く使用されているプログラミング・テクニックである ' プロセス指向プログラミング ' を言語レベルでサポートし, より使い易い言語を提供することを目的として設計を進め, 現在処理系を開発中である.

本論文では, この $\mathcal{AYA}$ の言語仕様について述べる. また, 実現方式についても簡単に触れる.

1　はじめに

KL1 は, GHC に基づき, 実用的な大規模システムの記述に適した機能を付加した並列論理型言語である. 現在 KL1 は並列推論マシンの実験機 Multi-PSI の上の分散処理系上で稼働している.

並列推論マシンのオペレーティング・システム PIMOS[1] は KL1 で記述した. また, KL1 による大規模なアプリケーション・プログラムの開発も進められている.

この KL1 のプログラミング・テクニックに ' プロセス指向プログラミング ' がある. これは, メッセージを交換しながら並列に動作するプロセス群としてプログラムを構成するものである.

この方法では, プログラムを独立性の高いプロセスの集まりとして記述するため, プログラムのモジュラリティを高く保てる. また, プロセスを単位として並列性の制御を考えやすい利点もある. PIMOS では, 基本的な設計から全面的にプロセスを中心としたモデルを採用し, デバイスなどの管理プロセスを分散させることにより集中管理によるボトルネックを回避している.

しかし, この技法は 1 つのプログラミングスタイルに過ぎず, 言語のレベルでのプロセス指向による抽象化のためのサポートはない. このため, プロセスの状態を表すための引数をそれに対するアクセスの有無にかかわらず毎回記述しなければならない. また, 状態変数を追加するような場合つまり引数の数を増やす場合に, プログラムに対して大量の修正を行うことになりがちである. また, デバッガなどにこのスタイルのための機能を取り入れることも困難である.

そこで, このプロセスとプロセス間の通信をそのまま記述できる上位言語 $\mathcal{AYA}$ を KL1 上に設計した. 次章で KL1 と ' プロセス指向プログラミング ' について簡単に説明したあと, $\mathcal{AYA}$ の言語仕様・文法・実現方式について簡単に述べる.

2 KL1 とプロセス指向プログラミング

2.1 KL1

KL1[2] は, GHC をもとに, オペレーティング・システムを始めとする大規模かつ実用的なソフトウェアを記述するさいに必要となる機能を加え, 拡張した言語である.

KL1 への主な拡張点は荘園機能と呼ばれるメタレベル実行制御機構である. 荘園機能によりメタレベル (監視するレベル) とオブジェクトレベル (監視されるレベル) のプログラムを明確に分けることができる. そして, メタレベルプログラムはオブジェクトレベルプログラムを安全に監視 / 制御できるようになる. このメタレベル・プログラムとオブジェクトレベル・プログラムの関係は, そのままオペレーティング・システムとユーザ・プログラムの関係となる. KL1 で記述したオペレーティング・システム PIMOS は, この荘園機能を利用してユーザ・プログラムの管理をしている.

このほか, 荘園間, ゴール間, クローズ間の優先度管理機能, 1 次元のランダムアクセス可能な構造体の導入などがある.

2.2 プロセス指向プログラミング

KL1 で使用しているプロセス指向プログラミングのスタイル [3] は, はじめ, Concurrent Prolog に対して提案されたものであるが, すべての並列論理型言語に共通して使用できるものである.

このプログラミングスタイルではプロセスを次のようなものと見なしている.

プロセス プロセスは, 自分自身を再帰的に呼び出すことにより, 継続して繰り返し実行される KL1 のゴールである. プロセスの状態は, 他のプロセスとの間で共有しない引数により保持する.

プロセス間の通信 プロセスは, 複数のプロセス間で共有されている変数に値を渡すことによりプロセス間通信を行う.

メッセージの受信 プロセスは共有変数に値が渡されると処理をはじめる. 処理が終わると次の値を受けとるまで待ち状態にはいる.

プロセスの生成 プロセスはゴールが呼び出されることにより生成される.

2.3 プロセス指向プログラミングの不便さ

プロセス指向プログラミングは, メッセージ交換を行いながら独立に動作するプロセスの集まりとしてプログラムを記述するので, 並列性の制御を考えやすい, プログラムのモジュラリティを高く保てるなど利点も多い.

しかし, プログラミング・テクニックの 1 つに過ぎないため実際に大規模なプログラムを開発しまた更新していくうえではつぎのような不都合がある.

- デバッガにこのスタイルのための機能を取り入れることが困難である. プログラムは, プロセスとその間のメッセージ通信を意識して記述する. しかし, プログラムをデバッグするときには, ゴールの呼びだしとユニフィケーションしかあらわれず, 意識の上でギャップがある. しかし, 再帰呼びだしも単なる 1 つのゴール呼びだしに過ぎないため, デバッガがそれを特別に扱うことは難しい.

- プロセスの引数を再帰呼びだしのゴールの引数として必ず記述しなければならず煩わしい. またそのさいに, 変数名・引数の順番を間違えるなどのバグも入り易い.
- 状態変数を追加するような場合, つまり引数の数を増やす場合に, プログラムに対して大量の修正を行うことになりがちである. これは, 煩雑でありバグがはいりやすい.

このような問題点を解決するために, KL1 上にプロセス指向でそのまま記述できる言語を設計した.

3 言語仕様

3.1 概要

KL1 のプロセス指向プログラミングでは, 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, ゴール間の共有変数を用いて通信を行うことは前に述べた. AYA ではこのプロセスを抽象化したものを ' プロセスのクラス ' とし, これをプログラムの記述・実行上の単位とする.

プロセスのクラスはさらに, プロセスのおおまかな状態変化に対応する ' シーン ' の集まりとして定義する. 受けとれるメッセージの種類, メッセージを受けとったときに行う処理はシーンごとに異なって定義できる. プロセスは, メッセージを受けとると現在いるシーンの記述に従って処理を行い, 他のシーンに移ることもある. この繰り返しで処理を行う.

3.2 メッセージ

プロセス間を共有変数によって運ばれ, 通信を行うものをメッセージという. メッセージの特定の記法はない. また, メッセージとして使用できるデータ型に制限はない.

3.3 ラインとソケット

プロセスはメッセージの送受信を行うくちとしてソケットを持つ. ソケットには送信用 ('−') ソケットと受信用 ('+') ソケットがある. メッセージは '−' のソケットから '+' のソケットへ送られる. ソケット間を運ぶのがラインである. メッセージはラインを共有する '−' のソケットから '+' のソケットに運ばれる. メッセージを送信するひとは, メッセージの受けとり手がどのプロセスかを意識する必要はない.

ラインにはソケット名あるいは変数名でアクセスする. ライン 1 つで, メッセージを 1 つ運ぶことができる. メッセージの送受信にソケットを再度使用するためには再度ラインをセットしなくてはならない.

ソケットにアクセスするオペレーションとして次のものを用意する.

```
set_socket(ソケット名, 現在のライン, 新しいライン)
```

このオペレーションを行うと指定したソケットの現在の値が ' 現在のライン ' に返り, ' 新しいライン ' で指定した値が新たにセットされる. 現在のラインを取り出すことと, 新しいラインをセットすることは不可分の操作とする. 新しいラインを設定しない, つまりソケットが空であるという状態は, 自分あるいはメッセージの受け取りてとなるプロセスのデッドロックを引き起こすおそれがあるからである.

メッセージの送受信もこの set_socket( ) により表せる. メッセージの送信は, この ' 現在のライン ' にメッセージを渡す操作である. メッセージの受信は, ' 現在のライン ' にメッセージが返されることである.

ソケットは，プロセスあるいはシーンの定義のときに，名前とモードを指定して宣言する．このとき初期値を指定することもできる．なにも指定しないと，既定値の‘[ ]’が設定される．

3.4 プロセスの定義

プロセスはプロセスのクラス定義ごとに定義する．プロセスのクラスは必ず‘初期化シーン’を持つ．初期化シーンは名前をもたない．ここで，ソケットを宣言し，初期化処理を記述する．

プロセスは，シーンの集まりとして定義する．これらのシーンは，初期化シーンにネストしたシーンとして定義する．シーンは必要なだけネストすることができる．各シーンは，独立にソケット定義・初期化処理・メソッド定義を持つ．メソッド定義とは，受信したメッセージに対する処理の記述である．

プロセスは必要なだけシーンをもち，このシーンを移動しながら処理を行うことができる．シーンを移ることにより，処理対象とする入力ソケット，扱うメッセージを変えることができる．つまりシーンという記述レベルを導入したことにより，よく似たオブジェクトの生成と消滅を繰り返すことによるオーバーヘッドを避けることができる．

3.5 メソッド定義

シーンは複数個の入力用ソケットを持つことができる．このため，メッセージとそれの到着したソケット名の組に対して処理を定義する．これをメソッドと呼ぶ．メソッドの定義は次の形式で行う．

```
input ソケット名.
イベント -> コンディション | アクション \\ 次のシーンの指定.
```

‘input ソケット名.’を基本ソケットの宣言という．この宣言に続くメソッドの定義の‘イベント’は，ここで宣言されたソケットにメッセージが到着することをいう．

ソケットにメッセージが到着するとコンディションをチェックし，満たされていればアクションで指定した処理を行う．さらに指定があれば，シーンを移る．イベントの指定が省略されることもある．この場合は，まずコンディションをチェックし条件を満たしていればアクションを実行する．

コンディションとしては，基本ソケット以外のソケットの値のチェックなどのパタンマッチ，言語定義のコンディション用プロセスが記述できる．

アクションとしては，メッセージの送信および他のプロセスの起動が記述できる．プロセスの起動は，プロセスのクラス名と必要な初期化パラメタを渡して行う．

他のシーンに移動する場合は，そのシーンの名前と必要な初期化用パラメタを渡して行う．プロセス実行の終了は，‘terminate’という言語定義のシーンへ移動することである．

3.6 次のシーンを直接記述

次のシーンとして，メソッドを直接記述することができる．このシーンは名前を持たない．ソケットの定義はこのシーンに移ろうとしているメソッドのものを共有する．また，このメソッドと変数のスコープを共有する．このシーンを繰り返すことはできない．このシーンを用いて，メッセージの戻り値による条件分岐などを記述できる．

このシーンは次のように記述する．

```
イベント -> コンディション | アクション
```

```
\\ { メソッド ;
   ...
  メソッド}.
```

3.7 複数の入力ソケット

1つのシーンが複数の入力ソケットからのメッセージを扱うことができる. これにより, 異なるプロセスからのメッセージを, あるメッセージ列の処理の間に受け付けることができる. とくに, 次に述べるプライオリティを併せて使用して, 緊急割り込み処理に応じることもできる.

たとえば, ウインドウなどのデバイスに次のような定義をすることができる. この例では, abort というソケットにはデバイスをアボートするメッセージのみ到着する. 通常のメッセージは request で受けとるようになっている.

```
input abort.
abort -> アボートされたときの処理.
input request.
:gett(Term) -> タームの読み込み処理.
:putt(Term) -> タームの書きだし処理.
```

このように記述することにより, 通常のメッセージの合間にアボートの指令を受け付けることができる.

3.8 イベント間プライオリティ

メソッド定義の選択つまり, 各メソッドの選択の要件であるイベントの間にプライオリティを指定することができる. このプライオリティは逐字的に順序をつけるものではなく, 1つのイベントあるいはいくつかのイベントといくつかのイベントの間の優先順位を定めるものである. このプライオリティを指定するために

```
alternatively.
```

を導入する. alternatively で区切られたメソッドの間に, 基本ソケット宣言があってもよい.

```
input ソケット1.
イベント1 -> コンディション1 | アクション1 \\ 次のシーン1.
イベント2 -> コンディション2 | アクション2 \\ 次のシーン2.
input ソケット2.
イベント3 -> コンディション3 | アクション3 \\ 次のシーン3.
alternatively.
イベント4 -> コンディション4 | アクション4 \\ 次のシーン4.
```

この場合, イベント1,2,3 はイベント4よりプライオリティが高い. イベント4はソケット2へのメッセージの到着である.

alternatively を用いて前節の例をつぎのように記述することができる.

```
input abort.
abort -> アボートされたときの処理.
alternatively.
input request.
:gett(Term) -> タームの読み込み処理.
:putt(Term) -> タームの書きだし処理.
```

このようにすることで, abort の到着を request にとどく通常の I/O メッセージより高い優先順位で処理することができる.

3.9 否定の略記

それ以前に記述してあるイベント及びコンディションがすべて失敗したことを示す略記法として

```
otherwise
```

を導入する. otherwise は異なる基本ソケットに対するイベント・コンディションに対しても有効である.

```
input ソケット1.
イベント1 -> コンディション1 | アクション1 \\ 次のシーン1.
イベント2 -> コンディション2 | アクション2 \\ 次のシーン2.
input ソケット2.
イベント3 -> コンディション3 | アクション3 \\ 次のシーン3.
otherwise.
イベント4 -> コンディション4 | アクション4 \\ 次のシーン4.
```

イベント4は, イベント・コンディションの1から3までがすべて失敗したときにのみ試される.

3.10 ユニファイ

2本のラインをつなぐプロセスとして'ユニファイ'を用意する.

```
unify(Out,In)
```

これは, 'In' に渡されたラインに値が渡されたら, 'Out' に渡されたラインからその値を出力するというプロセスである.

3.11 システム提供のプロセス

コンディション・チェック, 算術演算, 構造体アクセスなどのためのプロセスは言語定義プロセスとして提供する. これら言語定義プロセスのうち算術演算などについてはマクロ記法を用意する. これはそれぞれの演算に対してそれをあらわす算術式を用意し, それを

```
~(    )
```

で囲うことにより, 式の計算結果をあらわすことにする.

```
 ~(1 + 2)
```

これは1+2の計算結果を表す.

```
unify(Ans,~(1+2))
```

このように前節で紹介したユニファイと共に使用すると, 計算の結果が Ans に返ることになる.

3.12 ストリームの用法

ラインの用法の1つとしてリストによるストリーム通信を用意する. これは, リストセルのCar にメッセージ, Cdr に新しいラインをつめて1つのメッセージとして送受信するというものである.

`set_socket(ソケット名,[メッセージ | 新しいライン], 新しいライン)`

この用法を用いて任意個数の順序のあるメッセージ送受信を実現することができる. ストリームによるメッセージ送信を終わりにすることを ' ストリームを閉じる ' ということにする. これを, '[ ]' を送信することで示す. 同様に '[ ]' を受信することを ' ストリームを閉じられた ' ということにする.

4 文法・シンタックス

4.1 クラス・シーン定義

クラスおよびシーンはつぎのように定義する.

```
<class definition> ::=
    ''class'' <class name>[''('' <formal>{'',''<formal>}'')'']
        [''with'' <socket declaration>['':='' <initial value>]]
        {'',''<socket declaration>['':=''<initial value>]}['';''
        <initiation>]
        [''\\'' [<initial scene>]]''.''
    ({<scene definition>} | <method definition> ''.'')
    ''end class.''
```

クラス定義は, ソケット宣言・クラス呼びだし時の初期化処理およびシーンの定義あるいはメソッド定義よりなる. メソッド定義は, < *initial scene* > を指定しない場合のみ記述できる. ここで定義したメソッドは, クラスの初期化のメソッド定義として実行する.

```
<scene definition> ::=
    ''scene'' <scene name>[''('' <formal>{'',''<formal>}'')'']
        [''with'' <socket declaration>['':='' <initial value>]]
        {'',''<socket declaration>['':=''<initial value>]}['';''
        <initiation>]
        [''\\'' [<next scene>]]''.''
        {<method definition> ''.''}
    [''end scene'']
```

シーン定義は, ソケット宣言・シーン呼びだし時の初期化処理およびメソッド定義よりなる. メソッド定義は, < *next scene* > を指定しない場合のみ記述できる.

4.2 メソッド定義

```
<method definition> ::=
    <input socket declaration> | <method> |''alternatively'' | ''otherwise''
```

```
<method> ::=
    [<event>] ''->'' [<condition>{'',''<condition>} ''|'']
        <action>{'',''<action>} ['']''<next scene><pragma>]
```

メソッド定義として基本ソケットの宣言, alternatively, otherwise およびメソッドを記述することができる. メソッドは < *event* >, < *condition* >, < *action* >, < *next scene* > により記述する.

1 つのメソッド中で同一のソケットに対するアクセスが複数回行われる場合は, ソケットはメソッド内での出現順序に従って更新される.

4.3 ソケット・オペレーション

ソケットへのアクセスのため set_socket( ) というオペレーションを用意した. この操作を記述するための記法を用意する.

set_socket(ソケット名, 現在のライン, 新しいライン)

を次のように記述する.

@ ソケット名 ! 新しいライン

この記法は, 述語の引数としてのみ使用できる. この記述の値が ' 現在のライン ' である. '+' のソケットに対して ' 新しいライン ' の指定を省略した場合は, ' 現在のライン ' の参照とみなし, ' 現在のライン ' を再度設定する. '−' のソケットの場合はそのソケットに対する最後のアクセスのときのみ省略できることとする.

set_socket( ) の限定的な用法ではあるが, ソケットの値を更新する用法に対して, 次のような記法を別に用意する.

@ ソケット名 := 新しいライン

ただし新しいラインとして, 算術演算のマクロなども記述できる. たとえば, 次のような記述ができる.

@ ソケット名 := ~(@ ソケット名 + 1)

現在の値に 1 を加えた結果が新しく設定される.

ソケットの更新とは, つぎの操作を意味する.

set_socket(ソケット名, _, 新しいライン)

現在のラインはすてて, 新しいラインを設定することになる.

4.4 ユニファイ

ユニファイのプロセス

unify(OUT,IN)

を次のように書く.

```
OUT <- IN
```

このユニファイ・プロセスを使用して, メッセージの送信が記述できる. つまり, 次のように記述する.

```
@ソケット名 <- メッセージ
OUT <- メッセージ
```

4.5 構造体へのアクセス

ベクタおよびストリングのアクセスのためには, 要素の読みだし・更新のために言語定義プロセスを用意する. これらのプロセスを記述する別の方法として次の記法を用意する.

- ソケットに格納されているとき.

  ```
  @<socket name>{(<vector pos> | <string pos>)}[! <term>]
  ```

  ソケットに格納されているベクタあるいはストリングの指定された要素を 読み出し, それを値とする. $< term >$ が指定されているときは, 指定された 要素の更新を 同時に行う. 更新されたベクタあるいはストリングがソケットに格納される.

- ソケットに格納されていないとき.

  ```
  <variable name>{(<vector pos> | <string pos>)}
  ```

  指定された要素を読み出し, それを値とする. 更新は記述できない.

5 ストリーム

5.1 オペレーション

ストリームに対するオペレーションとして次の 2 つを用意する.

- merge(In,Out)
- concatenate(Stream1,Stream2,Stream3)

5.1.1 merge( )

言語定義プロセスとして

```
merge(IN,OUT)
```

を提供する. このプロセスはマージャとして動作する. このマージャは IN に渡されたストリームを入力ストリームとする. マージャは要素がストリームのベクタを受けとると, ベクタ内の全ての要素を新たに入力ストリームとして加える. この操作をマージインと呼ぶ. マージインは次のような操作である.

```
set_socket(@ソケット名, 現在のストリーム, 新しいストリーム),
現在のストリーム = {新しいストリーム, 増えるストリーム}
```

この操作のために次のような記法を用意する.

```
@ソケット名 <= 増えるストリーム
```

5.1.2 concatenate( )

concatenate(A,B,C)

ソケットに入っているストリームと，もう1本のストリームをつないで1本にする操作である．concatenate した結果の新しいストリームをソケットの新しい値とする．このときのソケットの'+'・'−'とストリームをつなぐ順番によって4通りの操作になる．

ソケットに入っているストリームを前につなぐ この操作をつぎのように書く．

@ソケット名 <=< ストリーム

@in <=< S

はソケットの'+'・'−'により，つぎの操作を意味する．

- ソケットが'+'のとき．

  X <- @in ! Y, concatenate(X,S,Y)

- ソケットが'−'のとき．

  @out ! Y <- X, concatenate(Y,S,X)

ソケットに入っているストリームを後につなぐ この操作を次のように書く．

@ソケット名 <<= ストリーム

@in <<= S

はソケットの'+'・'−'により，つぎの操作を意味する．

- ソケットが'+'のとき．

  X <- @in ! Y, concatenate(S,X,Y)

- ソケットが'−'のとき．

  @out ! Y <- X, concatenate(S,Y,X)

メッセージの送信をこの操作で表すことにする．'−'のソケットに対するときは相手へのメッセージの送信になり，'+'のときは自分がそのソケットから次によみこむメッセージを設定していることになる．

5.2 シンタックス

ストリームの用法のためのマクロ記法を用意する．

5.2.1 ストリーム型メッセージ

ストリーム型のメッセージであることを，メッセージの前に ':' をつけて示す．

':' メッセージ

これは，

[メッセージ]

のことである．ストリーム型メッセージを複数個続けるときはそのまま後につづけて

':' メッセージ':' メッセージ':' メッセージ

と記述する．これは，

[メッセージ，メッセージ，メッセージ]

である．

ストリームを閉鎖するメッセージは '::' で表す．これは，

[ ]

のことである．

5.2.2 ストリーム型メッセージの受信

ストリーム型メッセージの受信は，パタンマッチを用いてつぎのように記述する．

- イベントのとき．

  : メッセージ

- コンディションのとき．

  @ソケット名 = : メッセージ

5.2.3 ストリーム型メッセージの送信

ストリーム型メッセージの送信はソケットに対しプリペンドすることである．

@ソケット名 <<= : メッセージ1: メッセージ2

ここで，ソケットが '+' の場合は自分自身に対する送信となり，'−' のソケットの場合は，他のプロセスへの送信となる．この方法で自分自身に対して送ったメッセージは，このソケットに到着する他のメッセージより前に処理される．

5.2.4 ストリームの閉鎖

- 基本ソケットが閉鎖された.

  :: [! 新しいライン]

- コンディションでの確認.

  @ソケット名 = :: [! 新しいライン]

- ストリームを閉鎖する.

  @ソケット名:: [! 新しいライン]
  @ソケット名：メッセージ：メッセージ::

  この場合 '::' は, ストリームを閉鎖するオペレーションである.

5.2.5 タームの記法

ストリームのための操作をタームとして記述するための記法も用意する.

- @ソケット名：メッセージ ! 新しいライン

  ソケットに格納されているラインにストリーム型のメッセージを送った Cdr が値として渡される. ソケットには指定された新しいラインが 格納される.

- @ソケット名 <=

  マージインするストリームが値として渡される. ソケットには, マージイン されたもう一方が新しい値としてはいる.

5.2.6 ラインへのストリーム型メッセージ送信

ソケットに格納されていないラインに対しても, ユニファイのプロセスとストリームの記法を使用してメッセージの送信を記述する.

ソケットの場合にならい,

変数名: メッセージ

は, メッセージを送った Cdr を表すとする.

これとユニファイプロセスを使用してつぎのように記述する.

変数名 1：メッセージ 1：メッセージ 2 <- 変数名 2

変数名 1 の変数にメッセージ 1, メッセージ 2 を送り, その Cdr を変数名 2 とユニファイする.

ストリームを閉じるときはつぎのように記述する. この場合も '::' はストリームを閉じるオペレーションである.

変数名::
変数名: メッセージ 1：メッセージ 2::

6 実現方式

$\mathcal{AYA}$ のプログラムは KL1 にコンパイルして実行する．コンパイルした結果の KL1 プログラムは，プロセス指向スタイルの KL1 プログラムとほぼ同じものになる．

ライン・ソケット ラインおよびソケットは KL1 の論理変数にコンパイルする．このときソケットは KL1 述語の引数としてコンパイルされる．順次更新されていくソケットは，引数が更新されて (置き換わって) いくこととして表す．

メッセージの待ち合わせ メッセージの待ち合わせ (同期処理) には KL1 の同期機構をそのまま使用する．イベントの待ち合わせ，コンディション・チェックのためのパタンマッチは KL1 では同じレベルの同期処理にコンパイルされる．

メッセージの送信・ユニファイ メッセージの送信は，ソケット名であらわされた変数とメッセージとして記述したデータの KL1 のユニフィケーションにコンパイルする．$\mathcal{AYA}$ の提供するユニファイのプロセスも同様である．

プロセスの起動とシーンの移動 プロセスの起動およびシーンの移動は，KL1 のゴール呼びだしにコンパイルする．プロセスを起動するゴールと，シーンを移動するゴールはどちらも KL1 のゴールとなる．

alternatively・otherwise alternatively および otherwise については，そのまま対応する KL1 述語の 'alternatively' および 'otherwise' にコンパイルする．

ストリーム ストリームとして記述されたソケットおよびラインは，すべて KL1 のリストにコンパイルする．ストリームに対するアペンド・プリペンドといった操作はこのリストに対する操作となる．マージについては，KL1 が言語定義として提供しているマージャの機能を使用する．

言語定義プロセス $\mathcal{AYA}$ の言語定義プロセスとして提供する機能は，KL1 の組み込み述語あるいは，KL1 の通常の操作に変換する．

変数の出現のモード解析 $\mathcal{AYA}$ においては，ソケットに '+'・'−' のモードを付けを行った．また，言語定義プロセスの引数の一部は，'+'・'−' のモードが明らかである．これらの情報をつかって，モード不明のラインに対してもモードの予測を行い，全体としての整合性を調べることが可能である．このチェックを行うことによりプログラマのミスによるデッドロックなどを検出することも予定している．

7 今後の課題

今後の課題として，まず継承機構の導入がある．継承機構としては，複数のプロセスのクラスに共通するメッセージ処理を共有できるものであることが望ましい．しかし $\mathcal{AYA}$ では，各プロセスは複数のシーンよりなり，さらに各シーンは複数の入力ソケットを持つ．このため，どのメッセージ処理を共有したいのかを簡単に特定しにくい．また，KL1 にコンパイルして実現するため，実現方法にも制約がある．

そこで，あるひとつの入力ソケットから得られるメッセージ群についてのメソッドをはじめから切り出して記述し，各プロセスのクラスのシーンでは，こうして切り出した処理の呼び出しのみを指定することにする．この切り出した処理の記述の間に，継承機構を導入することを検討している．

そのほか, デバッガの実現も実際に使用するにあたっては必要となる. また, 文法その他の手直しも考える必要がでてくるかもしれない.

8 おわりに

KL1 のプロセス指向の手法によるプログラミングをサポートするために, KL1 上に高水準言語 $\mathcal{AYA}$(綾) を設計した.

KL1 のプロセス指向プログラミングスタイルによりプログラムを記述すると, ソケットに対応するものを述語の引数として, メソッドの繰り返し実行はほとんど同じ引数を持つ再帰呼び出しとして記述することになる. このため, 記述が不必要に繁雑になっていた.

しかし, この言語では, ソケットはアクセスする必要がある場合だけ名前を用いて記述するようにした. また, 繰り返し実行を基本とすることにした. このため簡明で抽象度の高い記述ができるようになっている.

シーンという概念を導入することにより, 等質な処理を繰り返すだけではなく, 大局的な状態によってメッセージに対する処理を質的に変えたり, 異なる経路からのメッセージに対応したりすることができる. また, ひとつのシーン内で複数の経路 (入力ソケット) からのメッセージを扱えるようにした. これにより, 別経路からの通常処理の緊急中断指示などにも応じることができる.

同様の研究としては Vulcan[4], FLENG++[5], Polka[6] などがある. いずれも, $\mathcal{AYA}$ と近い問題意識にもとづいて設計されたものであり, アプローチも似ている. これらとの比較において $\mathcal{AYA}$ の特徴は

- シーンの概念を導入したこと.
- ラインを基本としてストリームを単なる記法として提供したこと.
- 共有変数と非共有変数をシンタックスの上で区別しないこと.

などがあげられる.

参考文献

[1] 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.

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

[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] 中村宏明他. 並列論理型言語 fleng に基づいたオブジェクト指向言語 Fleng++. *WOOC'89*, 1989.

[6] Andrew Davison. Polka : A parlog object oriented language. Ph.d thesis, Imperial College, 1989.

A　サンプル・プログラム

AYA で記述したプログラムの例として，素数生成プログラムと，リーダーズライターズ問題のプログラムを示す．

A.1　素数生成プログラム

素数生成プログラムは 0 から与えられた最大値までの間の素数を生成しウインドウに出力するプログラムである．

```
module prime.
public top/1.

class top(Max) ;
        primes(Max,Ps),
        output(Ps) \\ .
end class.

class primes(Max,Ps) ;
        gen(2,Max,Ns),
        sift(Ns,Ps) \\ .
end class.  % end primes

class gen(+no,+max,-ns).
    -> @no =< @max | @ns <<= :@no,
                     @no := @no + 1.
    -> @no > @max | @ns:: \\.
end class.  % end gen

class sift(+ns,-ps).
   input ns.
     :: -> @ps:: \\ .
     :P -> @ps <<= :P,
           filter(P,@ns!Ys,Ys).
end class. % end sift

class filter(+p,+xs,-ys).
   input xs.
     :: -> @ys:: \\ .
     :X -> (X mod @p) =\= 0 | @ys <<= :X .
     :X -> (X mod @p) =:= 0 | continue.
end class.  % end filter

class output(+ps)
        with -out ;
        shoen:raise(pimos_tag#shell,get_std_out,Out),
```

```
        @out := Out.
    input ps.
      :: -> @out:: \\ .
      :X -> @out <<= :putt(X):nl .
end class.  % output
```

A.2 リーダーズ・ライターズプログラム

ファイルなど共有資源へのアクセスの問題である．ファイルへのアクセスは，次の規則に基づいて行う．

- read 操作は複数人が同時に行うことができる．
- write 操作は 1 人しか行うことができない．write が行われている間は read もできない．
- read 操作ばかりが続くことがないように，read の最中に write の要求が きたら，そのとき実行されている read がすべて終了したところで write 操作を行う．
- write 操作が終了したら，次に来ている read あるいは write 操作を行う．

```
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
```