プログラミング－言語・基礎・実践－研究報告　9－5
（1992．10．30）

プロセス記述を指向した並列論理型言語
— *LPd* —

宮崎敏彦
沖電気工業(株)　関西総合研究所

Guarded Horn Clauses (GHC) をはじめとする並列論理型言語は，コンカレントなモデルをプログラミングする道具として，その簡潔な同期規則や不完全データの扱いなど幾つかの優れた性質を持っている．また，再帰呼び出しによるプロセスの実現とリスト構造を用いたストリーム通信は，並列論理型言語におけるオブジェクト指向プログラミング(プロセス指向プログラミング)のテクニックとして広く知られている．しかしその反面非常に平面的なシンタックスのため，同じ述語名を何度も繰り返し記述する必要があったり，変更していないプロセスの状態(ゴールの引数)を明示的に書く必要があるなど，問題点が指摘されている．本稿では代表的並列論理型言語の一つ *KL1* を対象として，種々のプログラミングテクニックとそれをサポートするための表記法(*LPd*)を提案する．

Process Oriented Logic Language
— *LPd* —

Toshihiko Miyazaki
Kansai Laboratory　Oki Electric Ind. Co., Ltd.

Concurrent Logic Programming Languages such as *KL1* have several good features in making parallel programs. The languages can handle the incomplete data structure and have clear semantics about synchronization, etc. However they are weak in notational supports for typical programming techniques in concurrent logic language such as object-oriented programming, stream manipulation. In this paper we propose a language that extend a notation to support the typical programming styles. We called the new language "*LPd*".

1 はじめに

Guarded Horn Clauses (GHC) をはじめとする並列論理型言語は，コンカレントなモデルをプログラミングする道具として，その簡潔な同期規則や不完全データの扱いなど幾つかの優れた性質を持っている．また，再帰呼び出しによるプロセスの実現とリスト構造を用いたストリーム通信は，並列論理型言語におけるオブジェクト指向プログラミング (プロセス指向プログラミング) のテクニックとして広く知られている．

しかしその反面非常に平面的なシンタックスのため，同じ述語名を何度も繰り返し記述する必要があったり，変更していないプロセスの状態 (ゴールの引数として実現される) を明示的に書く必要があるなど，シンタックス上の問題点が指摘されている．これらの問題を解決するために，[2],[5],[10],[11] など多くの表記上の工夫が提案されている．また，[6] のようにストリーム通信とオブジェクト指向の融合を進めた言語の研究もある．

本稿では代表的並列論理型言語の一つ *KL1* を対象として，プロセスの記述やストリーム通信，差分リストなど種々のプログラミングテクニックをサポートするための表記法と，*KL1* で十分サポートできていない条件判定のための表記上の拡張を有した言語 *LPd* を提案する．*LPd* は *KL1* へのソースレベルの変換によって実現する．なお，本稿で述べる「プロセス」という概念は [3] にある「オブジェクト」と同様のものと見なせるが，通常のオブジェクト指向言語の機能である継承がまだサポートされていないこと，および後で述べるステータスという概念があるなどの理由からプロセスと呼んでいる．

2 *LPd* におけるプロセスのモデルと定義

LPd におけるプロセスのモデルを図1に示す．プロセスは1本のメッセージストリームと呼ぶストリームを介して外部からのメッセージ (要求) を順次受け取る．一つのプロセスが他の複数のプロセスからメッセージを受ける場合は，マージャを介してそれらのプロセスとつながる．

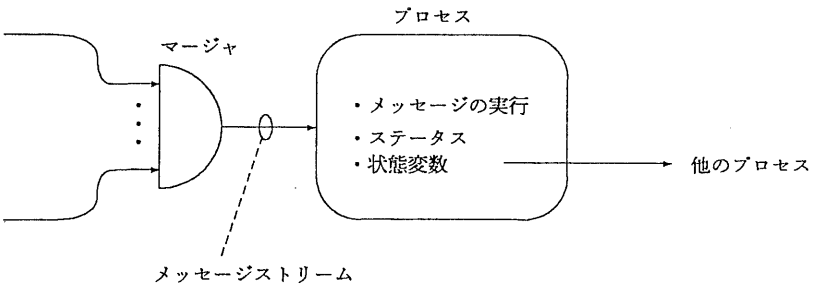

図 1: *LPd* のプロセスモデル

個々のプロセスは内部状態を持つが，この内部状態は，ステータスと呼ぶ複数のメッセージ解釈のコンテキストと，種々のパラメタを維持する状態変数からなる．状態変数は，そのプロセスの全てのステータスに共通の変数と，個々のステータス固有の変数に分類される．他のプロセスへのメッセージストリームは適当な状態変数に格納されることになる．

図2に *LPd* におけるプロセス定義の概観を示す．図において，(1) はこのプロセス全体を通して共通に使われる状態変数の宣言である．(2) はプロセスが取りうるステータスの宣言であり，ステータス固有の状態変数もこのとき同時に宣言する．(3) では全てのステータスで受け付け可能なメッセージパターンとその処理の定義を記述し，(4) では (2) で宣言した個々のステータスに固有なメッセージとその処理を定義する．

簡単な例を使って *LPd* のプロセス定義を説明する．Program-1 はオブジェクト指向 (プロセス指向) プログラミングの例として良く使われるカウンタプログラムの *LPd* によるプログラミング例である．このカウンタの例では状態が一つの counter という名前のプロセスを定義している．

```
process プロセス名 has
    状態変数の宣言                          ..... (1)
  status
    ステータスの宣言                        ..... (2)
  interface
    メッセージパターンとその処理の定義      ..... (3)
  状態名 interface
    メッセージパターンとその処理の定義      ..... (4)
  local
    プロセス固有の述語の定義
end.
```

図 2: プロセスの定義

```
1:  process counter has
2:       C : integer ;
3:    interface
4:       clear:    true | C <- 0 ;
5:       add:      true | C <- C + 1 ;
6:       read(X): true | X = C ;
7:  end.
```

Program-1: *LPd* によるカウンタプロセスの例

2行目の C:integer はプロセスの状態変数 C の宣言であり，同時に型が integer であることを宣言している．3行目以降は，受付可能なメッセージパターンとそれを受け取った場合の動作の定義であり、これをメッセージ定義と呼ぶ．メッセージ定義は ":" より前のメッセージパターンと ":" から "|" までのガード部および "|" 以降のアクティブ部よりなる．ガード部とアクティブ部を合わせてボディ部とも呼ぶ。アクティブ部の中にボディ部がネストしてもよい．

メッセージパターン : ガード部 | アクティブ部 ;
(ガード部 | アクティブ部 をボディ部)

中置演算子 "<-" は右辺の値で左辺に指定された状態変数の値を更新するという意味があるが，状態変数の型によって *KL1* への展開形式は異なる[1]．中置演算子 "=" は *KL1* と同様通常のユニフィケーションを意味する．

参考として Program-1 と同じ仕様のカウンタプロセスを *KL1* を使って記述すると Program-2 のように書ける．

```
1: counter([clear  |In], _) :- true | counter(In, 0).
2: counter([add    |In], C) :- true | C1 := C + 1, counter(In, C1).
3: counter([read(X)|In], C) :- true | X = C, counter(In, C).
4: counter([],           _) :- true | true.
```

Program-2: *KL1* によるカウンタプロセスの定義

Program-1 と Program-2 を比べて分かるように，*LPd* では状態の遷移や他のプロセスの生成が無い場合，再帰呼び出しを明示的に記述する必要はない．また，次のメッセージを受けるための変数 (Program-2 における変数 In) を明示的に記述する必要もない．プロセスの終了は，デフォルトではメッセージストリームが閉じられた場合であり，この場合の個々の状態変数の後処理も変数の型によってデフォルト動作が決められている．適当なメッセージの受信時の動作としてプロセスの終了を明示したい場合は，自身の入力ストリームを表す予約変数である Self に対しアクティブ部で Self<-[] とする．また，メッセージストリームが閉じられた場合の処理を明示したい場合は，メッセージパターンを []: とし，ボディ部にその動作を定義する．

3 状態変数の宣言と型

状態変数には，プロセスの全ての状態に共通なものと status で宣言した個々の状態に固有なものとがある．両者の違いは変数のスコープである．状態変数の宣言形式を図3に示す。

[1]状態変数の型とその扱いについては 3章で説明する．

```
<状態変数宣言>  ::= <状態変数名> ":" <型> [ "," <状態変数名> ":" <型> ] ";"
<型>            ::= <基本型> | <応用型> | <ユーザ定義型>
<基本型>        ::= "integer" | "list" | "atom" | "vector" | …
<応用型>        ::= "instream" | "outstream" | "oldnew" | "shared"
```

図 3: 状態変数の宣言形式

状態変数の宣言は, 変数名とその変数が取る型とを対にして宣言する. ここで宣言された型情報は, *KL1* から *LPd* への変換の際に, 状態変数の値の更新やゴール間での状態変数の受渡し方法を決めるために使われる. 特に応用型は *LPd* のプログラムを *KL1* に展開する際の展開形式を決めるために使われる型であり, 基本型は応用型のいずれかに属する. (なお, 上記のような応用型は [9] で最初に導入された.)

以下, 応用型を中心に状態変数の型を説明する.

3.1 ストリーム

LPd では *KL1* と同様, プロセス間の通信をストリームと呼ぶリスト構造を使って実現している. Program-3 の例は *KL1* でのプログラミング例である.

```
p([read(X)|In], File) :-
    true | File = [read(X)|NewFile], p(In, NewFile).
```

Program-3: *KL1* でのストリーム処理の例

述語 p の第1引数は p が受け取る入力ストリームとメッセージパターンである. 第2引数 File は p の外部のプロセスに対する出力ストリームである. 例に示すように, ストリームの実現にリスト構造を使っているため, メッセージを受けた後 (あるいは送った後) のストリームの口に相当する新しい論理変数 (In や NewFile) を明示的に記述する必要がある.

また, Program-4 に示す例のように, 1本の出力ストリーム (File) を複数のプロセスに渡す場合には, 2本のストリーム (File1 と File2) を用意し, それらをもとの出力ストリームにマージするように書く必要がある.[2]

```
p([read(X)|In], File) :-
    true | File = [read(X)|NewFile],
           merge(File1, File2, NewFile),
           q( …, File1, … ),
           p(In, File2).
```

Program-4: *KL1* におけるストリームのマージ

LPd では出力ストリームに関するこのような定型的な記述は, 状態変数の型を outstream 型と宣言するだけで良い. Program-5.1 に Program-4 と同等の出力ストリームを使った例を示す. 例において #q は, q という名前のプロセスを生成するという意味であり, 引数はそのプロセスの対応する状態変数の初期値である.

```
process p has
     File : outstream ;
  interface
    read(X): true | File<-read(X), #q( …, File, … ) ;
        …
```

Program-5.1: *LPd* でのストリームの利用

状態変数 File は, outstream というストリーム型と宣言されているので, Program-5.1 の read(X) のメッセージ定義のボディ部は, Program-5.2 のように展開される.

```
… :- true | File=[read(X)|NewFile],
            merge(NewFile, File1, File2),
            q( …, File1, … ), p(NewIn , …, File2, … ).
```

Program-5.2: Program-5.1 の展開例

[2]勿論, もとの出力ストリーム (File) をマージせずにそのまま q と p に渡すプログラムも有りうるが, 意味が異なるうえほとんど使われることは無い.

ストリーム型状態変数のプロセス終了時のデフォルト処理は, instream については何もせず, outstream には空リスト([])をユニファイする. また, プロセス終了時に instream を他のプロセスへの outstream に直接繋げることによって以降のメッセージを委譲したい場合があるが, これは "<-" の右辺に instream 型の状態変数名を書くことによって記述することができる.

3.2 差分リストとベクタ

リストを扱う場合のプログラミングテクニックとして差分リストがある. 例えば文法記述形式 DCG(Definite Clause Grammer) のように, 入力されたリスト(この場合トークン列)の処理前と処理後(文法適用前と適用後)を差分リストとしてあらわしたり, 逆に個々のサブゴールにより生成される解を, 順序を保ったまま一つのリストにまとめたい場合などに使う. *KL1* における差分リストを用いたプログラムの概観はおおよそ Program-6 のような変数の受け渡しになるのが普通である.

```
p( …, Head, Tail, … ) :-
    true | q( …, Head, Mid1, … ),
           r( …, Mid1, Mid2, … ),
           p( …, Mid2, Tail, … ).
```

Program-6: *KL1* における差分リストの利用

また, あるゴール群全体の処理が終了したかどうかを検出するショートサーキットと呼ばれるテクニック [1] も, プログラムの形は Program-6 のようになる.

一方, *KL1* のように変数への代入が単一代入であるような言語では, 要素の更新を伴うベクタの操作は, 意味的にはもとのベクタの更新された要素のみが異なる新たなベクタの生成である. 従って, 複数のゴール(プロセス)で同一のベクタを更新しながら参照するようなプログラムでは, Program-6 のように, それぞれのゴールに変更前と変更後のベクタをバインドする変数を記述し, さらにゴール間で次々に受渡ししなければならない.

このように, ベクタも差分リストやショートサーキットもゴール間の変数の関係という意味では同じ形をしている. *LPd* ではこのような使われ方をするデータ型を総称して oldnew 型と呼ぶ. 状態変数の型として vector と宣言した場合は, ベクタ固有の操作を除いて基本的には oldnew 型として扱う. なお, ベクタの要素の参照と更新のために次の構文が用意されている.

```
参照:   状態変数名 ! インデックス              % Vect ! I
更新:   状態変数名 ! インデックス <- 要素      % Vect ! I <- NewElement
```

Program-6 の例からも分かるように oldnew 型の状態変数は, *KL1* 上では 2 つの変数に展開される. プロセス終了時の oldnew 型のデフォルト処理は, この 2 つの変数のユニファイである.

3.3 共有データ

参照のみのベクタや変更されない閾値など, 多くのプロセスに共有されるようなデータを保持する状態変数は shared 型として宣言する. integer や atom などアトミックデータも shared 型として扱う. 例えば Program-1 のカウンタの例で示された状態変数 C は intger として宣言されており, 展開時には shared 型として扱われる. shared 型変数の値の変更は, ボディ部の定義の順序に依存しており, ソーステキスト上で左から右へと順次優先される. 従って更新前の値を参照したい場合は, 一旦一時的な別の変数に保存しておく必要がある.

```
ソース:   C <- C+1,  p(C),  C <- C+2,  q(C)
展開後:   C1 := C + 1,  p(C1),  C2 := C1 + 2,  q(C2)
```

なお, プロセス終了時の shared 型に対するデフォルト処理は特にない.

3.4 ユーザ定義型

LPd ではユーザによる構造体の定義も許している. ユーザによって定義された型をユーザ定義型と呼ぶ.

```
定義:   struct ユーザ定義型名 has
            フィールド名 : 型名,
            …
        end.

参照:   状態変数名 ! フィールド名
```

ユーザ定義型の展開時の扱いは oldnew 型と同じである．従って，その型の構成要素として outstream 型や shared 型を持つような場合は，利用方法を誤ると不必要な逐次性が入ってしまう．

4 ボディ部の記述

4.1 ガード部からのユーザ定義述語の呼び出し

KL1 のガード部では組込述語の実行しか許していない．この制限は主に記述上の美しさよりも実行効率を重視したためである．しかし，ガード部の役割が同期と条件判定であると考えるならば，このような制限はプログラマに無用の労力を強いていることになる．例えば Program-7 に示すプログラムのように，ある値がリスト要素であるか否かで動作を変えたいような場合，条件である member をボディ部に記述し，member の実行結果を明示的に返してもらって判定するという煩雑なプログラムを書かなければならない．参考として prolog で普通に定義する member 述語を Program-8 に示す．Prolog の定義では，メンバーか否かの判定は述語の成功 / 失敗という実行結果として検出している．

```
p([do(Command)|In], ComList) :-
    true | member(ComList, Command, Result),
           p_check(In, ComList, Result).

p_check(In, ComList, yes) :- true | do(Command), p(In, ComList).
p_check(In, ComList, no ) :- true | p(In, ComList).

member([],     Key, Result) :- true      | Result=no.
member([K|_], Key, Result) :- K = Key   | Result=yes.
member([K|L], Key, Result) :- K ~= Key | member(L, Key, Result).
```

Program-7: *KL1* における member 述語の利用

```
member([K|_], Key) :- K ==  Key, !.
member([_|L], Key) :- member(L, Key).
```

Program-8: Prolog による member の定義

そもそも *KL1* のガード部における制限の目的は，同期メカニズムの実現を簡単にすることであった．しかしガード部におけるユーザ定義の述語呼び出しを全て禁止しているため，上記 Program-8 のような，呼び出し側の変数に対する値の束縛がおきない述語すらガード部に記述することができなくなっている．そこで *LPd* では，SafeGHC と呼ぶクラスに属す述語の呼び出しをガード部で許すことによって，プログラマの負担を軽減している．

ここで使っている Safety という概念は Concurrent Prolog を対象とした [8] で述べられているものに近く，アクティブ部にあるどのユニフィケーション (以下アクティブユニフィケーションと呼ぶ) もサスペンドしないような GHC のサブクラスである[3]．当該プログラムが SafeGHC の要件を満たすか否かは *LPd* のプログラムを *KL1* に変換する際に静的に解析する．SafeGHC のクラスに含まれるプログラムは，アクティブユニフィケーションがサスペンドしないという性質上，容易に *KL1* に変換することができる．SafeGHC から *KL1* への変換の結果のイメージは，上記 Program-7 の member の定義のように実行の成功 / 失敗を返す引数を追加し，p_check のような結果を判定するための述語を付加するというものである．

4.2 ボディ部のネスト

Program-7 から読み取れる *KL1* のもう一つの問題点は，member 述語の結果 (Result) の判定のために p_check という余分な述語を定義しなければならない点である．この問題を回避する為に，*LPd* では DEC-10 Prolog などで導入されているクローズ内 OR の記述形式を採用する．

Program-7 は *LPd* におけるボディ部のネストの例である．例では，ファイル (File) に対して読み込み要求メッセージ (read(Data,Rst)) を送り，その結果 (Rst) によって次の動作を決めるというプログラムを，クローズ内 OR の形式で書いたものである．なお，OR のネストは何段になっても良い．

```
read(D): true | File<-read(Data, Rst),
                        ( Rst = yes          | D=Data ;
                          Rst = error(Code) |  ···      );
```

[3] 勿論アクティブ部にユニフィケーションがない場合も含む．

5 ステータスの記述

プロセスの内部状態によって，メッセージに対する動作を変えたり，受け付け可能なメッセージを変えたりしたい場合がある．*LPd* ではこのような場合，status 宣言を使って複数の状態（ステータスと呼ぶ）を宣言することができる．また，そのステータス固有の状態変数も同時に宣言可能である．Program-9 に 2 分探索木のプログラム例を示す．[4]

```
process btree has
  status leaf : default,
         branch( Key    : integer ,
                 Value  : shared ,
                 Left   : outstream ,
                 Right  : outstream ) ;

  leaf interface
     search(_, V):
        true | V = undefined ;
     update(K, V):
        true | @branch( Self!Self, Key!K, Value!V, Left!L, Right!R ),
               @leaf( Self!L ),
               @leaf( Self!R ) ;
  branch interface
     search(K, V):
        ( K = Key | V = Value ;
          K < Key | Left  <- search(K, V) ;
          K > Key | Right <- search(K, V)   );
     update(K, V):
        ( K = Key | Key <- K, Value <- V ;
          K < Key | Left  <- update(K, V) ;
          K > Key | Right <- update(K, V)   ) ;
end.
```

Program-9: 2 分探索木

プロセス btree のステータスには leaf と branch があり，leaf はデータの無いノード，branch はデータ（変数 V に保持される）を有するノードを意味する．btree の初期ステータスは，データの無い leaf ステータスで起動され，データ挿入要求メッセージ（update(K, V)）によって，データ保持プロセス（ステータスが branch）と 2 つの leaf プロセスを生成する．

プロセス生成のための記法は，

プロセス名 @ ステータス名 (状態変数名 ! 値 , …)

であるが，自身のプロセスの場合はプロセス名を省略できる．また，プロセスの初期ステータスがデフォルトでよければステータス名は必要ない．

6 自律的プロセス

並列型言語で分散人工知能的な問題を扱う場合，プロセスの自律性は重要である．しかし，[5],[10],[11],[6] などでは外部からのメッセージに応えることによる受動的な動作しか考えられていない．これはオブジェクト指向言語一般にも言える傾向である．

LPd では，メッセージパターンに “_” を記述したメッセージ定義によって，外部からのメッセージが来ない間の処理を記述することが可能である．

このようなメッセージ定義は，*KL1* のクローズ間の優先順位付けオペレータ alternatively と組込述語 unbound を使って展開される．組込述語 unbound(X, Result) は，unbound 実行時点で X が未束縛ならば Result に {PE,Address,X} なるベクタをユニファイし，値が束縛されていれば {X} をユニファイする[5]．この機能を使って，上記メッセージ定義を,自身のメッセージストリームにメッセージが到着しているか否かをチェックし，未到着であれば定義のボディ部を実行するというような *KL1* のプログラムに展開する．

[4] このプログラムは [7] に示されているものを *LPd* で書き換えたものである．

[5] PE はプロセサ番号 Address は変数のアドレスであるが，ここではベクタの要素数に意味がある．

7 関連研究

ここでは主に並列論理型言語の記述上の改良に関する関連研究との比較を行なう.

AU'M[6] は, ストリームに基づく計算モデルを実現しようとしている言語であり, データに対する操作をオブジェクトとそれに繋がるストリームという考え方で統一しようとしている. *LPd* は AU'M ほどオブジェクト指向に統一しようというものではなく, *KL1* の言語機能がある程度透けて見えるようになっている.

FLENG++[11] と AYA[10] は, それぞれのベース言語によるプログラミングの負担の軽減を目的としている点で *LPd* に非常に近い. 主な違いは以下の点である.

1. 状態変数に型を宣言させることによって, マージ操作や差分リストの扱いを抽象化している.
2. ガード部にユーザ定義述語の記述を許すことによって, プログラマの負担を軽減している.
3. ステータスを宣言することによって, 内部状態によるメッセージの解釈の違いを記述できる. ただし, AYA にはシーンと呼ばれる同様の概念が有る.

8 おわりに

本稿では, 並列論理型言語 (特に *KL1*) のシンタックス上の問題点を述べ, *LPd* という言語の設計によってこれを解決するとともに, プロセス指向プログラミングを行なう上でのプログラミングテクニックをサポートする記述上の拡張を説明した.

KL1 では, プロセス内でグローバルな状態変数をゴールの引数に書き並べたり, ストリームの操作など定型的な記述を細かく書く必要があるなど, プログラマに余分な負担を掛けていた. *LPd* ではプログラムをプロセスと型を持った状態変数ととらえることにより, 不必要な変数の記述を減らし, プログラマに対する負担を軽減している. また, メッセージの解釈が異なるようなプロセスの内部状態をステータスの宣言によって区別することができる.

LPd の *KL1* へのトランスレータは現在開発中であり, 大規模なプログラムに対する評価は今後の課題である. また, インヘリタンスや多入力のストリームの記述法および要求駆動型のメッセージ通信の導入などについても現在検討中である.

参考文献

[1] Akikazu Takeuchi. *Parallel Logic Programming.* PhD thesis, University of Tokyo, 1990.

[2] Andrew Davison. *Polka: A Parlog Object Oriented Language.* PhD thesis, Imperial College, 1989.

[3] E.Y. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, 1(1):25–48, 1983.

[4] Jacob Levy and Ehud Shapiro. Translation of Safe GHC and Safe Concurrent Prolog to FCP. In *Concurrent Prolog: Collected Papers, Vol 2*, pages 383–414. MIT press, 1987.

[5] Kahn K.M. and Tribble D. and Miller M.S. and Bobrow D.G. Vulcan: Logical Concurrent Objects. In *Concurrent Prolog: Collected Papers, Vol 2*, pages 274–303. MIT press, 1987.

[6] Kaoru Yoshida and Takashi Chikayama. A'UM - A stream based Concurrent Object-Oriented Language. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[7] Kazunori Ueda and Masao Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 799–808. ICOT, 1992.

[8] Michael Codish and Ehud Shapiro. Compiling Or-Parallelism into And-Parallelism. In *Concurrent Prolog: Collected Papers, Vol 2*, pages 351–382. MIT press, 1987.

[9] 宮崎敏彦, その他. PDSS Manual. Technical Report TM-437, ICOT, 1988.

[10] 寿崎かすみ and 近山隆. KL1 上の並列プロセス指向言語 AYA. Technical Report TR-652, ICOT, 1991.

[11] 中村宏明 and 小中裕喜 and 田中英彦. 並列論理型言語 FLENG に基づいたオブジェクト指向言語 FLENG++. In *WOOC'89* 論文集, 1989.