ICOT Technical Memorandum: TM-0120

TM-0120

AND-OR-Queuing in Extended Concurrent Prolog

横森　貴，岸下　誠（富士通）
田中二郎

June, 1985

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

AND-OR-Queuing in Extended Concurrent Prolog*

田中二郎** 横森 貞 岸下 誠
（富士通 国際研） （富士通SSL）

[Abstract]

We have implemented the Extended Concurrent Prolog (ECP) Interpreter[Fujitsu 85], which has OR-parallel, set- abstraction and meta-inference facilities, by modifying Concurrent Prolog (CP) Interpreter [Shapiro 83].

In Shapiro's CP Interpreter, only the AND-related goals are enqueued to the scheduling queue. However, in our ECP interpreter, all the AND-related and OR-related goals are enqueued to one scheduling queue.

We have named this scheduling method "AND-OR-Queuing". By this "AND-OR-Queuing" method, it becomes possible to handle all kinds of AND-relations and OR-relations in a uniform manner.

1. はじめに

Concurrent Prolog (CP) [Shapiro 83] とは、Pure Prolog に、コミット・オペレータ及びread-only annotationを付加した、並列論理型言語である。CPの持つ並列性を逐次マシンの上でimplement しようとするとき、良く使われる技法にscheduling queueをつかう方法がある。即ち、並列処理可能なプロセスをqueue に入れ、並列関係を疑似的に模倣実行するわけである。

筆者らは、Shapiro のCP Interpreter [Shapiro 83] を拡張し、AND並列及びOR並列機能、集合抽象化機能、メタ推論機能などを持つExtended Concurrent Prolog (ECP) Interpreterの試作を試みた[Fujitsu 85]。

本論文では、我々の試作したECP の諸機能のうち、特にscheduling queueに関連した部分に絞って、その基本インプリメント技法の解説を行なう。

2. Extended Concurrent Prolog

前述したように、ECP とはCPに AND並列及びOR並列機能、集合抽象化機能、メタ推論機能などを付加したもので、そのそれぞれの機能は、1984年の核言語第１版概念仕様[Furukawa 84] に基づいている。ECP のこれら拡張機能につき、以下簡単に説明する。

2.1 AND並列及びOR並列機能

まずECP の基本的並列推論機構としてあるのが AND並列及びOR並列機能である。

AND 並列機能とは、論理的にAND で結ばれたゴールを並列に評価する機能である。この機能は、評価を行なうべきゴール列をscheduling queueの中に格納し、queue の先頭のゴールをとり出し、リダクションを行ない、その結果をqueue の末尾に登録することにより実現される。このAND並列機能については、すでにShapiro のInterpreter で実現されていた。

一方、OR並列機能とは、ゴールとユニファイ可能なヘッドを持つ節が複数個あるとき、これらの節についてガード部の実行を並列に行なう機能である。このOR並列はShapiro のInterpreter では実現されていない。

OR並列機能を利用するプログラムとして以下のような例を考える。

```
solve(P,Hes) :-  call(P) |... .
solve(P,Hes) :-  find-stop(Hes)  |... .
```

solve が呼ばれたとき、上の二つの節はOR並列機能により同時に実行される。上の節はP を実行しているが、その実行中に下の節でHes の中にstopが見つかれば、下の定義節がコミットされ、P が解きかけであればP の実行がabortされる。これによりabort 付きsolve が実現される。

2.2 集合抽象化機能

集合抽象化機能とは、並列環境下におけるPure Prolog的全解探索機能の実現のために導入された機能である。こうした集合抽象化機能を実現する述語として、以下のような形式のeager-enumerate とlazy-enumerateの二つが提案されている[Fujitsu 84]。

```
eager-enumerate({X | Goals},L)
lazy-enumerate({X | Goals},L)
```

ここでGoals は、Pure Prolog の世界で定義されている述

* 本研究は、第５世代コンピュータ・プロジェクトの一環として行われたものである。

**1985年 6月より、ICOT、第１研究室。

語からなるゴール列である。Pure Prolog の世界の定義の仕方であるが、ここでは、

```
pp((<ヘッド>←<ボディ部>)).
```

という形で、即ち、ppというfunctor 名を持つfactの有限集合としてassertされているものと仮定する。

この二つのenumerate は、Pure Prolog の世界でGoals を解き、それを満たすようなX の集合をリストの形でL に入れる。

集合抽象化の例として以下の例を考える。

```
eager-enumerate({X | grand-child(jiro,X)},L)
```

このとき、pure prolog の世界は以下のように定義されているとする。

```
pp((grand-child(X,Z)<---child(X,Y),child(Y,Z))).
pp((child(jiro,keiko)<---true)).
pp((child(yoko,takashi)<---true)).
pp((child(jiro,yoko)<---true)).
pp((child(keiko,makoto)<---true)).
```

この場合解としてL には[takashi,makoto]が入る。

一般に、eager とlazyの違いはL の具体化の仕方の相違である。eager の場合は能動的にL を具体化するのに比べ、lazyの場合は受動的であり、L は解が外から要求されのを待ち、それに応じて解のリストを生成していく。例えば、

```
:- lazy-enumerate({X | prime(X)},L?),
   display(L,Hes?),keybord(Hes).
```

の場合には、display より解が要求されるに従い解のリストが生成されていく。

2.3 メタ推論機能

メタ推論機能とは、与えられたゴールを、ある世界(World) の中で定義されている知識を用いて解く機能である[Furukawa 84] 。このメタ推論機能を実現するプリミティブとして、次のような形式の、述語simulateを用意する。

```
simulate(World, Goals, Result, Control)
```

ここでWorld は与えられる世界名、Goals は解かれるべきゴール列、Resultはゴールを与えられた世界の知識を使って解いた時の結果、Control はメタ推論機能を一時止めたり再開したりするコントロール情報を入れるストリームである。ある世界の知識は、ここでもpure prolog 世界と同様に、

```
world名((<ヘッド>←<ガード部> | <ボディ部>)).
```

という形で、即ち、 world名というfunctor 名を持つfactの有限集合としてassertされているものと仮定する。

メタ推論機能の例として、コントロール情報を見ながら動いたり止まったりするforegroundジョブと、常に一定に動くbackgroundジョブが走る、shell の例[Clark 84]を挙げる。

```
shell([], _).
shell([fg(G) | N],C) :-
        simulate(f-world,G,R,C)&
        remove(C,NewC)&
        shell(N?,NewC).
shell([bg(G) | N],C) :-
        simulate(b-world,G,R,_),
        shell(N?,C).
:-shell([bg(primes),fg(primes)],C),
        control(C).
```

この例ではprimesを計算するジョブがbackgroundとforegroundの両方で走り、foregroundの方はcontrol の影響を受けることになる。

3. ECP の拡張機能とその実現

以上ECP の拡張機能について簡単に説明した。これら拡張機能の実現にあたっては、当然ながら、Shapiro のInteroreter では不十分である。それぞれの機能の中に存在するAND 関係及びOR関係を、scheduling queueの中でうまくimplement することが重要となってくる。

我々は、考察の後、従来、AND 関係のみを格納し、OR関係一つ一つにつき作られていたscheduling queueをグローバルなもの一本だけにすることに決め、プログラム実行中に現れるすべての並列AND 関係及び並列OR関係をこの一本のscheduling queueの中に入れることに決めた。イメージとしてはプログラムの計算過程で生じるAND-OR Tree をそのまま一本のscheduling queueの中に入れた感じである。我々はこの方式を"AND-OR-Queuing"と命名した。この方式により、AND 関係とOR関係を統一的に考察することが可能となった。本稿では、以下、各機能のscheduling queueへの格納方式について解説する。

3.1 AND並列及びOR並列機能の実現

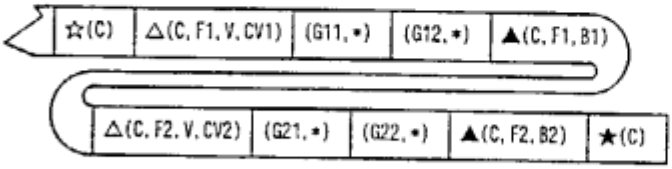
前に述べたように、AND 並列については、scheduling q

ueueにAND 関係のゴールを格納することで既に満たされている。

OR並列については、考察の末、OR並列を二種類のマーカで囲み、scheduling queueの中に格納することにした。例えば、queue の先頭から取り出されたゴールがPであり、ゴールPに対する候補節が

$$P_1 \ :- \ G_{11}, G_{12} \ | \ B_1$$
$$P_2 \ :- \ G_{21}, G_{22} \ | \ B_2$$

であるとき、scheduling queueの末尾には以下のように格納される。



ここでOR-Claueses 全体は☆と★で囲まれている。定義節のガード部分はそれぞれ△と▲に囲まれて表されている。☆と★の間はOR関係、△と▲の間はAND 関係になっている事に留意されたい。

ここで一つ一つのプロセスの中での'*' はそのプロセスの属する世界が、グローバルデータベースの世界であることを示している。また各マーカに共通の引数C は、OR-Clausesのある節がコミットされたかどうかの情報を格納している。マーカ△の引数Fiは、各OR-Clause に失敗が起ったか否かの情報を示す（この引数は、ガードが成功したという情報を示す必要がない。なぜなら、この引数は、各OR-Clause に固有の変数であり、ガードが成功し、コミットと同時に捨てられてしまい、他から参照されることはないからである）。引数V は、OR-Clausesの呼び出し元のゴールP に含まれている変数のリストであり、引数CVi は、V が各ガード部においてコピーされて作られる変数リストである。マーカ▲の引数Biは、それぞれの定義節のボディ部分を格納している。

このscheduling queueは、queue からゴールが取り出されたときは、通常のscheduling queueと全く同様に処理が行なわれるが、マーカが取り出されたときは以下のようになる。即ち、

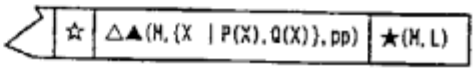
①マーカ☆(C) もしくは★(C) が取り出されたときには、引数C に'committed' がセットされていればマーカは捨てられ、そうでなければ、queue の末尾に再格納される。
②マーカ☆と★とが続けて取り出されたならば、これはガード部がすべて失敗していることを示しているので、呼び出し元のゴールP は失敗となる。
③マーカ△(C,Fi,V,CVi)が取り出され、引数C に'committed' 、もしくは引数Fiに'failed'がセットされているならば、△から▲までをすべて取除く。
④マーカ△(C,Fi,V,CVi)と▲(C,Fi,Bi) が続けて取り出されたならば、これはガード部がすべて成功したことを示しているので、引数C を'committed' に具体化し、V とCVi をユニファイし、Biをschedulingする。
⑤マーカ▲(C,Fi,Bi) は取り出されるとそのまま末尾へ再格納する。

3.2 集合抽象化機能の実現

集合抽象化機能の例として、scheduling queueから次のようなゴールが呼出されたケースを考える。

```
eager-enumerate({X | P(X),Q(X)}, L)
```

このとき、scheduling queueの末尾には、つぎのように格納されると考えられる。



ここで、ふたたび二種類のマーカが現れていることに留意されたい。マーカの意味は前章とは多少異なるが、☆と★の間がOR関係、△▲で表されるマーカの内部がAND 関係になっている事は同じである。☆と★は全体を囲んでおり、解を集めそれを外部に公開する役目をする。△▲は、解の一つを計算する役目をする。

引数H は、集合を満たす解を集めるための変数であり、L は、外部へ解を出力するストリームである。ppはPure Prolog の世界名である。

queue からマーカが取り出されたときは、以下の処理を行う。

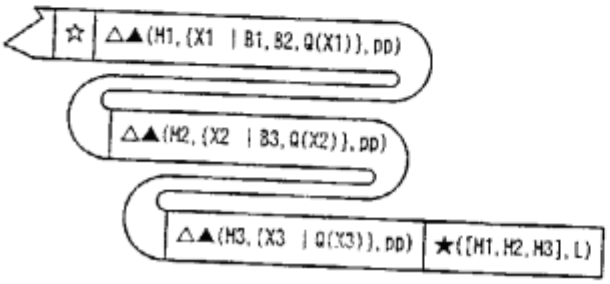
①マーカ☆が取り出されたとき、次がマーカ★(H,L) である、即ち☆と★との間が空であれば、それは集合を満たす解はすべて見い出されたことを意味するので引数L をnil('[]') に具体化し、ストリームを閉じる。そうでなければ、queue の末尾に再格納する。
②マーカ△▲(H,{X | …}, PP)が取り出されたときには、この集合の条件節の最左のゴール（この列ではP(X)）の定義節を捜し出し、複数のユニファイ可能節があれば核分裂を起こす。引数H は、核分裂の際に一緒に分裂していく。

③マーカ★(H,L) が取り出されたときには、引数H を調べ解が送られていれば、その解をストリームL に送り、マーカは末尾に格納する。

核分裂については、解りにくいので例で説明する。いま、マーカ△▲(H,{X | P(X)…}, pp)が取り出され、P がPure Prolog の世界で以下のように定義されていると仮定する。

```
pp((P(X)←B1,B2)).
pp((P(X)←B3)).
pp((P(X)←true)).
```

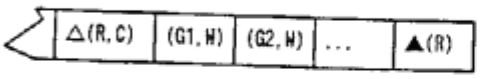
P(X)の定義節は三つあるので、マーカ△▲は核分裂を起こし、最終的には以下のようなものがscheduling queueの末尾に格納される。



こうして核分裂を次々に起こしながら、最終的に全解が求まることになる。ここでは、OR並列に基づき、depth-first で解が集められていることに留意されたい。lazy-enumerateについても、☆における解の集め方が異なるだけで、全く同じ原理で処理できることになる。

3.3 メタ推論機能の実現

scheduling queueからsimulate述語が呼出された場合、今Goals がG1, G2, ... であるとすれば、scheduling queueの末尾には、つぎのような形で格納されると考えられる。



上の図で、マーカ△と▲がまた使われていることに留意されたい。引数のR はResult、C はControl 、W は world名をそれぞれ表している。queue からマーカが取り出された時の処理を簡単にまとめると次のようになる。

①マーカ△(R,C) がqueue から取り出された時、R に既に'failure' がセットされているときは、△から▲までをすべて取除く。
②マーカ△(R,C) がqueue から取り出された時、queue の先頭が▲である、即ち△と▲の間が空であれば、Resultに'success' がセットされる。
③マーカ△(R,C) がqueue から取り出された時、C が[……, abort | 変数] であるときは、△から▲間のゴールを取り除き、R に'abortion'をセットする。即ちゴールを強制終了する。
④マーカ△(R,C) がqueue から取り出された時、C が[... stop | 変数] であるときは、△以降のゴールにリダクションを行なうことなく、対応する▲までのゴール列をqueue の末尾につける。
⑤マーカ△(R,C) がqueue から取り出された時、C が[... cont | 変数] もしくは変数であるときは、単にマーカ△をqueue の末尾につけ、次にすすむ。
⑥マーカ▲(R) は取り出されるとそのまま末尾へ再格納する。

△と▲の間がAND 関係になっている事はここでも同じであり、△と▲の間のゴール列の処理についても、与えられた世界を意識しつつ普通のリダクションと全く同じ処理が行なわれる。むろんこの中でOR並列、集合抽象化、メタ推論がネストしても構わない。△と▲の間でゴールが失敗したときには、△と▲の間はAND 関係であるのでResultに'failure' がセットされる。

4. ECP 処理系作成

前章までECP の諸機能及びその実現方式について述べた。筆者らは、Shapiro のCP Interpreter [Shapiro 83] を拡張しECP の処理系を試作したが、本章ではECP のインタプリタについて簡単に解説する。

4.1 Shapiro のCPインタプリタ

本節ではECP 処理系について述べる前に、まず、その土台となったShapiro のCPインタプリタについて簡単に説明する。Shapiro のCPインタプリタはprologで記述されており、CPの動きを示すのに適したコンサイスな処理系である。以下のプログラムはShapiro のインタプリタを更に単純化し処理の骨組みだけを示したものである。実際の処理系ではこれにdeadlockの検出、system述語、debug 、trace 機能等が付加され、処理系はもっと複雑になる。

```
①cp(A):-
    schedule(A, X-X, Head-Tail),
    solve(Head-Tail).

②solve([]-[]):-!
  solve([A | Head]-Tail):-
    system(A), !, A,
```

```
    solve(Head-Tail).
  solve([A | Head]-Tail):-
    reduce(A, B),
    schedule(B, Head-Tail, NewHead-NewTail),
    !, solve(NewHead-NewTail).

③reduce(A, B):-
    guarded-clause(A, (G | B)),
    cp(G), !.
  reduce(A, A).

④guarded-clause(A,B) :-
    copy-func(A, A1),
    clause(A1, B),
    unify(A, A1).

⑤schedule(true, Head-Tail, Head-Tail) :- !.
  schedule((A,B), Head-Tail, Head2-Tail2) :-!,
    schedule(A, Head-Tail, Head1-Tail1),
    schedule(B, Head1-Tail1, Head2-Tail2).
  schedule(A,Head-[A | Tail],Head-Tail).
```

このプログラムの意味は以下の通りである。（以下の説明において数字①～⑤は、プログラムにつけた数字①～⑤に対応している。）

①cpのゴール列A を解くためには、A をまずキューの中にスケジューリングし、それを解けばいい。
②solve はキューに格納されたゴールを解く述語である。キューが空なら処理は終りである。それ以外はキューの先頭からゴールを取り出して解くが、取り出されたゴールが組込みの述語ならそれを実行し、ユーザ定義の述語ならそれをreduceし、その結果の新しいゴールをキューに格納する。
③④reduceはユーザ定義のゴールを解く述語である。guarded-clauseは、与えられたゴールについてunify 可能なclauseを捜しだす。reduceは、まずguarded-clauseで与えられたゴールについてunify 可能なclauseを一つ捜しだし、そのガードを解く。ガードが解けない場合には自動的にバック・トラックして、次のunify 可能なclauseを捜し同じ動作を成功するまでくりかえす。またそれらが全て成功しなかったときにはsuspend とみなしゴールをそのまま結果として返す。
⑤scheduleは与えられたゴール列をキューに格納する。scheduling queueはD-listの形で書かれていることに留意されたい。述語scheduleはゴールがtrueであれば何も格納しない。ゴール列が(A,B) の形であれば、A を格納してからB を格納する。その他のケースにはゴールをキューの末尾に格納する。

4.2 ECP 処理系の作成

ECP の処理系(Interpreter) 試作にあたって、筆者らは、Shapiro のCP Interpreter [Shapiro 83] を拡張して試作を行った。我々の処理系の主な変更点は以下の通りである。

☆Shapiro の処理系では前節③に示したように、OR関係をバックトラックにより処理している。③のreduceは一つ一つのガードごとにトップ・レベルの述語cpを呼ぶので、3章にも述べたように、一つのOR関係ごとに一つのscheduling queueが作られることになる。我々は、OR関係をマーカ付きでグローバルなscheduling queueに格納し、OR並列を実現した。
☆Shapiro の処理系では、ゴールの処理（ユニフィケィション）において、失敗とサスペンドを区別していないが、我々の処理系では、失敗とサスペンドを区別するようにした。これによりOR並列におけるガードの失敗を扱うことが可能となった。
☆Shapiro の処理系では、直接インプリメントしていなかったが、我々は、集合抽象化機能、メタ推論機能について、マーカ付きで直接scheduling queueに入れることにし、インタプリタに各種マーカの処理をさせるようにした。

本処理系はVAX11/780 のC-Prolog及びDEC2060 のDEC-10 Prolog上で作成し、処理系は主に、スケジューリング部分、マーカ処理部分、及びPure Prolog 処理部分からなる。各部分のsizeは、順に約 150行、 170行、50行である。速度については、OR並列を実現しているせいもあり、Shapiro の処理系に比べ約 2～ 3倍の遅さである。これは、多環境のscheduling及びマーカの処理（例えば、ガード部がネストした場合、scheduling queue内には、非常に多くのプロセスとマーカとが格納される）に時間がかかるためである。本処理系の問題点としては、多環境を実現するためにコピーを多用すること、呼び出しのネストが深くなるためProlog処理系のstack を大量に消費してしまうこと、などがあげられよう。

5. ECP のプログラム例

ECP のプログラム例としてshell の例を考える。これは2.3で述べたshell の例[Clark 84]をよりrealistic にしたものである。

このプログラムでは、shell は複数個のジョブを同時に走らせ、またその実行をコントロールすることができる。(2.3の例では、forground ジョブは一度に一個しか走らせることができず、backgroundジョブは一度に多数個走らせ

ることができるものの、その実行をコントロールすることはできなかった。）このプログラムでは、shell 内で起動されるジョブは、すべてプロセスIDを持つようになっており、shell はプロセスIDを利用して、コマンドを受取り、ジョブの中断、回復、中止を行うことが出来る。

```
①  shell :- shell(I?,[]),in(I).

②  shell([proc(ID,Goals)|Input],IDL) :-
       prolog(id_print(IDL,IDL1))
       :  simulate(*,Goals,R,C),
          shell(Input?,[(ID,R,C)|IDL1]).
   shell([wproc(ID,W,Goals)|Input],IDL) :-
       prolog(id_print(IDL,IDL1))
       :  simulate(W,Goals,R,C),
          shell(Input?,[(ID,R,C)|IDL1]).
   shell([Com|Input],IDL) :-
       prolog((id_print(IDL,IDL1),
               send(IDL1,Com,NewIDL)))
       :  shell(Input?,NewIDL).

③  send([],_,[]).
   send([(ID,R,C)|IDL],Com,
                  [(ID,R,NewC)|IDL]) :-
       Com =.. [M,ID],!,C = [M|NewC].
   send([(ID,R,C)|IDL],Com,
                  [(ID,R,C)|NewIDL]) :-
       send(IDL,Com,NewIDL).
```

このshell のコマンドには以下のものがある。

コマンド	意味
proc(ID,Goals)	グローバルデータベースの世界でゴール列を解く
wproc(ID, 世界名, Goals)	与えられた世界内でゴール列を解く
stop(ID)	与えられたIDを持つプロセスの実行を中断する
cont(ID)	与えられたIDを持つ中断されているプロセスを回復する
abort(ID)	与えられたIDを持つプロセスを強制終了する

ここに示した、shell プログラムの意味は以下のとおりである。（以下の説明において数字①～③は、プログラムの数字①～③に対応している。）

①shell プログラムのトップレベルであり、2引数shellと、述語inを呼び出す、2引数shell は、inからのメッセージを受けて動く。

②shell プログラムの本体であり、第1引数は，外部からコマンドの入力を受けるストリーム、第2引数は、カレントに動いているプロセスのリストである。（これをIDリストと呼ぶ。）ここで1つ1つのプロセスは（ID,R,C）であらわされている。IDはプロセスを識別するためのidentifier、R は計算結果を外へ送るための変数、C はプロセスの動きをコントロールするためのチャネルである。

このshell は第1引数からメッセージを受けたときactiveになる。すなわち

・コマンドproc(ID,Goal) を受けたとき、述語simulateを呼び出し、グローバルデータベースの世界で与えられたGoalを解き、同時にIDリストに与えられたID、Goalの計算結果の帰る変数R 、及びコントロールチャネルC を組にして格納し、次の入力を待つ。

・コマンドwproc(ID,W,Goal)を受けたとき、述語simulateを呼び出し、世界W でGoalを解き、同時にIDリストに与えられたID、Goalの計算結果の帰る変数R 、及びコントロールチャネルC を組にして格納し、次の入力を待つ。

・その他のコマンド（stop, cont, abort)を受けたとき、述語sendを呼び出し、与えられたIDを持つプロセスにメッセージを送り、次の入力を待つ。

なお、各ガード部で呼び出される。id-printは、IDリストの中に、計算の終了したプロセスのIDがあるならば、その結果を出し、それをIDリストから削除する。

③sendは、stop, cont, abort 等のコマンドを受けたとき呼出され、IDリストの中からコマンドで指定されたプロセスを捜しだし、そのコントロールチャネルに与えられたコマンドを送る述語である。

sendの第1引数は入力されるIDリスト、第2引数は、入力されるコマンド、第3引数は出力されるIDリストであり、次のshell に引継がれる。

述語sendの動きを詳しく説明すると以下の通りである。

・IDリストが空であれば、NewID リストに空を返す。

・IDリストの先頭の組が、与えられたコマンドのIDと同じならば、そのプロセスのコントロールチャネルにメッセージを送り、そのプロセスのチャネルを新チャネルに置換える。

・上記以外の場合には、IDリストのプロセスは、そのまま残りのプロセスリストsendを呼び出す。

このshell プログラムを起動させた例を以下に示す。

```
?- solve(shell,R).
> proc(p01,primes).
> wproc(p02,s,prime(10)).
2
3
```

```
   2 (s)
   3 (s)
5
> stop(p01).
   5 (s)
   7 (s)
> cont(p01).
Result([p02,success])
7
11
> abort(p01).
Result([p01,abortion])
```

ここでは、'p01' として素数の無限列を生成するprimes、'p02' として、 1から10までの素数を生成するprime(10)を、世界s で起動している。

この実行例では、p01 が 2,3,5と出力した後、stop(p01) を送り、p01 を中断した。p02 が 7まで出力したときcont(p01) を送り、p01 を回復させた。その後、p02 は 1から10までの素数をすべて生成したので無事終了し、p01 が 11を出力した後、abort(p01)を送り、p01 を中止させた。

6. 関連研究の動向

この辺で、本研究に関連した、Concurrent Prolog の拡張機能のインプリメンテーションの研究動向について簡単にまとめてみたい。

①OR並列については、Levyによってグローバルなqueue を使う方式が既に提案されている[Levy 84] 。またICOTにおいても、OR並列が幾通りかの方式で試みられている [Miyazaki 84, Sato 84, Tanaka84] 。しかしながら、これらのアプローチにおいてはインプリメントがPascalやLispなどの汎用言語でなされており、我々のpurely logicalなアプローチとは異なっている。

②集合抽象化については、平川らのPOPS[Hirakawa 84] により、研究が進められてきた。POPSとは、Concurrent Prolog で書かれたPure Prolog のインタプリタであり、与えられたゴールに対する全解を次々にストリームとして数え上げる。我々のアプローチは、本質的にはPOPSのアプローチをscheduling queueの中にマップしたものと考えられる。

③メタ推論機能についての要点は、対象言語のインタプリタを対象言語の枠内でいかに実現するかということである。これについては、Shapiro により、Concurrent Prolog でConcurrent Prolog のインタプリタを記述する、いわゆるメタ・インタプリタ方式で研究が進められてきた[Shapiro 84]。それらの機能、使用法などについては、その後Clark らによって検討が加えられてきた[Clark 84]。我々の方式は、自分自身のscheduling queueの中に、処理されるべきゴールを放り込む方式であり、Shapiro の方式と比較した場合、より直接的方式となっている。

7. まとめ

以上、簡単に我々のExtended Concurrent Prologにおける拡張論機能の実現のための骨組み及び関連研究の動向について説明した。なお、本稿では詳しい記述は省略したが、実際のインプリメンティションにおいては、OR並列に伴って起こる変数のコピーの問題[Tanaka 84] などがあり、処理はもっと複雑である。

我々はAND-OR-Queuingを提案したが、従来、全く違った機能と考えられていたOR並列機能、集合抽象化機能、メタ推論機能などが一つの骨組みの中で説明できることは驚くべきことである。またハードウェアという視点から見た場合、scheduling queueが動的に生成されるとは考えにくく、この計算モデルは、シミュレーターとして見た場合、より現実的と言えそうである。また、一つのグローバルなqueue を仮定し、それによりすべてのAND 関係およびOR関係を処理することは、よりfairなschedulingにつながっている。すなわち、

①並列性については、OR並列を実現している。

②集合抽象化については、解を少しずつ求めながら、他のゴールも同時に少しずつリダクションしていくことができる。

③メタ推論においても、幾つかのsimurate述語をAND 並列により、並列に少しずつ走らすことが出来る。

さて、AND-OR-Queuing方式の適用範囲であるが、むろん適用範囲はConcurrent Prolog に限らない。新しいKL1 の中核と想定されているGHC [Ueda 85] においてもその方式は有効である。むしろOR並列に伴う多環境を生成しない分だけ適合性があるともいえる。また、竹内によって提案されている、AND-OR関係に着目したストリームAND 並列型言語の拡張[Takeuchi 84] においても、より強力なインプリメント手段を提供していると言えよう。

8. 研究分担、謝辞

本研究は、第５世代コンピュータ・プロジェクトの一環として行なわれたものである。本研究の研究分担であるが、ECP インタプリタの設計構想については、横森、田中が、コーディングについては、主として、岸下が担当した。

本研究にあたっては、ICOT第１研究室の竹内、上田の両氏から、様々の関連した話題につき協力を戴いた。また、国際研の北川会長及び榎本所長、ICOTの古川第１研究室長から助言を戴いた。また、富士通SSL の吉井氏より、いろいろ協力を戴いた。関係各位に感謝する次第である。

［参考文献］

[Clark 84]K.Clark and S.Gregory: Notes on Systems Programming in Parlog 、in Proceedings of the International Conference on Fifth Generation Computer Systems 1984、pp.299-306、ICOT、1984.

[Fujitsu 84]富士通: 58年度電子計算機基礎技術開発再委託成果報告書、5G核言語第１版の検証用ソフトウェア・詳細仕様書 Part II、1984.

[Fujitsu 85]富士通: 59年度電子計算機基礎技術開発再委託成果報告書、5G核言語第１版の検証用ソフトウェア・詳細仕様書（第２版）及び試験評価データ Part I 、1985.

[Furukawa 84] 古川、國藤、竹内、上田: 核言語第１版概念仕様書、ICOT、1984年 3月.

[Hirakawa 84] H.Hirakawa and T.Chikayama: Eager and Lazy Enumeration in Concurrent Prolog 、in Proceedings of the Second International Logic Proramming Conference、 Uppsala、pp.89-100 、1984.

[Levy 84] J.Levy: A Unification Algorithm for Concurrent Prolog 、in Proceedings of the Second International Logic Proramming Conference、 Uppsala、pp.333-341、1984.

[Miyazaki 84] 宮崎、竹内、古川: Concurrent Prolog のシーケンシャル・インプリメンテーション(Shallow binding方式による多環境の実現）、日本ソフトウェア科学会第１回大会論文集、3D-2、pp.295-298、1984年12月.

[Sato 84] 佐藤、市吉、太細、宮崎、竹内: Concurrent Prolog のシーケンシャル・インプリメンテーション(deep binding 方式による多環境の実現）、日本ソフトウェア科学会第１回大会論文集、3D-3、pp.299-302、1984年12月.

[Shapiro 83]E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter、ICOT Technical Report 、TR-003、1983.

[Shapiro 84]E.Shapiro: Systems Programming in Concurrent Prolog 、in Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages、 Salt Lake City 、pp.93-105 .

[Takeuchi 84] 竹内彰一: ストリームAND 並列論理型言語の一拡張について、日本ソフトウェア科学会第１回大会論文集、3D-1、pp.291-294、1984年12月.

[Tanaka 84] 田中、宮崎、竹内: Concurrent Prolog のシーケンシャル・インプリメンテーション(Copy 方式による多環境の実現）、日本ソフトウェア科学会第１回大会論文集、3D-4、pp.303-306、1984年12月.

[Ueda 85] K.Ueda: Guarded Horn Clauses、ICOT Technical Report 、TR-103、1985.

APPENDIX A

ECP インタプリタのインプリメント方式

☆ゴールについて

本文中では、各ゴールはその解が解かれるべき世界と対にしてqueue 内に格納すると表わしたが、インプリメントでは、マーカを付加し、

$P（ゴール、世界）

の形式を用いた。

☆マーカについて

本文中では、概要を示すために、scheduling queue内のマーカは、単純な記号で示したが、実際のインプリメントに際しては、以下のようなマーカを用いた。

	本文中	インプリメント
並列推論	☆(C) △(C,F V,CV) ▲(C,F,B) ★(C)	$GS(C) $G(C,F,V,CV) $G(C,F,B) $GE(C)
集合抽象化	☆ △▲(H,{X | …},pp) ★(H,L)	$SSET $SET(H,{X | …}) $ESET(H,L)
メタ推論	△(R,C) ▲(R)	$SIMU(R,C) $SIMU(R)

APPENDIX B

ECP インタプリタ・コード

```
/* operator priority declaration */

:- op(1200,xfx,((<--))), %%% FOR SET EXP.
   op(550,yfx,'\').      %%%

:- op(450,yf,'#'),       %%% WRITE EARLY
   op(450,yf,'?'),       %%% READ ONLY
   op(1025,xfy,'&'),     %%% SEQENTIAL AND
   op(1050,xfy,((:))).   %%% COMMIT OP.


/* top level */

solve(A,R) :-
   kl_system(A),!,
   solve_system(A,R),!.
solve(A,R) :- !,
   schedule(*,A,X\X,
            H\['$SIMU'(R),
               '$SIMU'(R,**)|T]),!,
   simulate(H\T),!.


/* scheduler */

schedule(_,true,H\T,H\T) :-!.
schedule(World,(A,B),H\T,H2\T2) :-
   schedule(World,A,H\T,H1\T1),
   schedule(World,B,H1\T1,H2\T2).
schedule(World,A,H\['$P'(A,World)|T],
                        H\T) :-!.


/* simulate ... queue handling */

simulate([]\[]) :- !.

simulate(['$SIMU'(R)|H]\
          ['$SIMU'(R)|T]) :- !,
   simulate(H\T),!.
simulate(['$SIMU'(R,_)|H]\T) :-
   R == failure ,!,
   del_simu(H,NH),!,
   simulate(NH\T),!.
simulate(['$SIMU'(success,_),
        '$SIMU'(success)|H]\T) :- !,
   simulate(H\T),!.

simulate(['$SIMU'(R,C)|H]\
         ['$SIMU'(R,C)|T]) :-
   (C == **;var(C)),!,
   simulate(H\T),!.
simulate(['$SIMU'(abortion,C)|
               H]\T) :-
   mem_abort(C),!,
   del_simu(H,NH),!,
   simulate(NH\T),!.
simulate(['$SIMU'(R,[stop,Cont|C])|
          H]\['$SIMU'(R,C)|T]) :-
   Cont == cont,!,
   simulate(H\T),!.
simulate(['$SIMU'(R,[stop|C])|H]\
         ['$SIMU'(R,[stop|C])|T]) :- !,
   skip_simu(H,T,NH,NT),!,
   simulate(NH\NT),!.
simulate(['$SIMU'(R,[cont|C])|H]\
         ['$SIMU'(R,C)|T]) :- !,
   simulate(H\T),!.

simulate(['$SSET'(_),
          '$SSET'(_,[])|H]\T) :-
   simulate(H\T),!.
simulate(['$SSET'(C)|H]\
           ['$SSET'(C)|T]) :- !,
   simulate(H\T),!.
simulate(['$SET'(Mess,Cls)|H]\T) :- !,
   reduce_set(Cls,Mess,T,T1),!,
   simulate(H\T1),!.
simulate(['$SSET'(Mess,S)|H]\
           ['$SSET'(Mess1,S1)|T]) :-
   collect_s(Mess,Mess1,S,S1),!,
   simulate(H\T),!.

simulate(['$P'(susp(ROV*A),World)|
                          H]\T) :- !,
   (check_nonvar(ROV),!,
      simulate(['$P'(A,World)|H]\T)
   ;  T = ['$P'(susp(ROV*A),World)|T1],!,
      simulate(H\T1)),!.

simulate(['$GS'(Ct)|H]\T) :-
   Ct == committed,!,
   simulate(H\T),!.
simulate(['$GE'(Ct)|H]\T) :-
   Ct == committed,!,
   simulate(H\T),!.
simulate(['$G'(Ct,Fail,_,_)|H]\T) :-
   (Ct == committed
   ;Fail == failed),!,
   del_guard(H,NH),!,
   simulate(NH\T),!.
simulate(['$UNIF'(Ct,_,_,_,_,_,_)|
                          H]\T) :-
   Ct  == committed,!,
   simulate(H\T),!.

simulate(['$GS'(Ct),'$GE'(Ct)|H]\T) :-
   del_goals_at_failure(H,T,NH,NT),!,
   simulate(NH\NT),!.

simulate(['$UNIF'(Ct,OV,CV,ROV,
            World,CGoal,Head,Bs)|H]\T) :- !,
   check_unify(Ct,OV,CV,ROV,World,
               CGoal,Head,Bs,H,T,Qs),
   simulate(Qs),!.

simulate(['$G'(committed,_,OV,CV),
          '$G'(committed,_,
               (Body,World))|H]\T) :-
   unify(OV,CV),!,
   schedule(World,Body,H\T,Qs),!,
   simulate(Qs),!.

simulate(['$GS'(Ct)|H]\
          ['$GS'(Ct)|T]) :- !,
   simulate(H\T),!.
simulate(['$GE'(Ct)|H]\
          ['$GE'(Ct)|T]) :- !,
   simulate(H\T),!.
simulate(['$G'(Ct,Fail,OV,CV)|H]\
         ['$G'(Ct,Fail,NO,NC)|T]) :- !,
   copy(OV,CV,NO,NC),!,
   simulate(H\T),!.
simulate(['$G'(Ct,Fail,Bs)|H]\
         ['$G'(Ct,Fail,Bs)|T]) :- !,
   simulate(H\T),!.

simulate(['$P'(simulate(World,Goal,R,
          C),_)|H]\['$SIMU'(R,C)|T]) :- !,
   schedule(World,Goal,H\T,
            NH\['$SIMU'(R)|NT]),!,
   simulate(NH\NT),!.
```

```
simulate(['$P'(set({X:Goal},Str),_)|H]\
         ['$SSET'(C),
          '$SET'(Mess,{XX : Goal1}),
          '$ESET'(Mess,Str)|T]) :- !,
    copyp((X,Goal),(XX,Goal1)),!,
    simulate(H\T),!.

simulate(['$P'(A,_)|H]\T) :-
    kl_system(A),!,
    solve_system(A,R1),!,
    (R1 == success,!,
       simulate(H\T)
    ;R1=susp(ROV),!,
       T = ['$P'(susp(ROV@A),_)|T1],!,
       simulate(H\T1)
    ;  del_goals_at_failure(H,T,NH,NT),!,
       simulate(NH\NT)),!.

simulate(['$P'(A,World)|H]\T) :-
    (simu_reduce(World,A,H\T,NH\NT)
    ;  del_goals_at_failure(H,T,NH,NT)),!,
    simulate(NH\NT),!.


/* reduce */

simu_reduce(World,Goal,
            H\['$GS'(Ct)|T],RH\RT) :-!,
    w_clauses(World,Goal,Clauses),!,
    reduce(World,Goal,Clauses,H\T,
            RH\['$GE'(Ct)|RT],Ct),!.

reduce(_,_,_,Qs,Qs,Ct) :-
    Ct == committed.
reduce(_,_,[],Qs,Qs,_).
reduce(World,Goal,[(Head <-- Bs)|Clauses],
                  Qs,Qs2,Ct) :-
    copy(Goal,CGoal,OV,CV),
    unify(CGoal,Head,UR),
    enqueue(World,UR,CGoal,Head,Bs,
                  OV,CV,Qs,Qs1,Ct),
    reduce(World,Goal,Clauses,Qs1,Qs2,Ct).


/* enqueue */

enqueue(_,failure,_,_,_,_,_,Qs,Qs,_).
enqueue(World,success,_,_,(Guard : Body),
        OV,CV,H\['$G'(Ct,Fail,OV,CV)|T],
                NH\NT,Ct) :-
    Guard \== true,
    schedule(World,Guard,H\T,NH\
            ['$G'(Ct,Fail,
                (Body,World))|NT]).
enqueue(World,success,_,_,
        (true : Body),OV,CV,
        Qs,Qs1,committed) :-
    unify(OV,CV),
    schedule(World,Body,Qs,Qs1).
enqueue(World,success,_,_,Body,
        OV,CV,Qs,Qs1,committed) :-
    unify(OV,CV),
    schedule(World,Body,Qs,Qs1).
enqueue(World,susp([ROV]),CGoal,Head,
        Bs,OV,CV,H\['SUNIF'(Ct,OV,CV,
        ROV,World,CGoal,Head,Bs)|T],
                        H\T,Ct).


/* pure prolog reduction */

reduce_set(Cls,Mess,T,T1) :-
    reducep(Cls,NextCls),!,
    fork_set(Cls,NextCls,Mess,T,T1),!.

reduce_set(Cls,'$SOL'(Mess),T,T) :-
    terminatep(Cls,Mess),!.
reduce_set(Cls,Mess,
           ['$SET'(Mess,NewCls)|T],T) :-
    systemp(Cls,NewCls),!.
reduce_set(_,'$FAIL$',T,T) :- !.

fork_set(Cls,NextCls,Mess,T,T1) :- !,
    select(NextCls,ClsList),
    forks(ClsList,Cls,Mess,T,T1),!.

forks([H],Cls,[Mess],
        ['$SET'(Mess,NewCls)|T],T) :- !,
    newset(H,Cls,NewCls),!.
forks([H|R],Cls,[Mess1|Mess],
       ['$SET'(Mess1,NewCls)|T],T1) :- !,
    newset(H,Cls,NewCls),!,
    forks(R,Cls,Mess,T,T1),!.

collect_s(X,FlatX,Str,Str1) :- !,
    connect_s(X,[]\FlatX,Str\Str1),!.


/* unifier */

unify(X0,Y1,R) :-
    unifier(X0,Y1,R),
    check_result(R).
unify(_,_,failure) :-!.

unifier(_,_,R) :-
    nonvar(R),!.
unifier(X0,Y1,R) :-
    nonvar(X0),
    X0 =.. [#,X00],
    (cvar(X00,X000),!,
       inherit(Y1,Y11,R),
          (var(R),
             Y11 = Y1,
             Y11 = X000
          ;  true)
    ;  unifier(X00,Y1,R)),!.
unifier(X0,Y1,R) :-
    nonvar(Y1),
    Y1 =.. [#,Y11],!,
    unifier(Y1,X0,R),!.

unifier(X0,Y1,_) :-
    (var(X0);var(Y1)),!,
    X0 = Y1.

unifier(X0?,Y1,R) :-!,
    (cvar(X0,X00),
       R = susp([X00])
    ;  unifier(X0,Y1,R)),!.
unifier(X0,Y1?,R) :-!,
    (cvar(Y1,Y11),
       R = susp([Y11])
    ;  unifier(X0,Y1,R)),!.

unifier([],[],_) :-!.
unifier([X0|X00],[Y1|Y11],R) :-
    unifier(X0,Y1,R),!,
    unifier(X00,Y11,R),!.

unifier(X0,Y1,R) :-
    sub_unifier(X0,Y1,R),!.

sub_unifier(X0,Y1,R) :-
    X0 =.. [Functor|X00],
    Y1 =.. [Functor|Y11],!,
    unifier(X00,Y11,R),!.
sub_unifier(_,_,failure).
```

ICOT Technical Memorandum: TM-0120(E)

TM-0120(E)

AND-OR-Queuing in Extended Concurrent Prolog

by
T. Yokomori, M. Kishishita
(Fujitsu Ltd.)
and J. Tanaka

September, 1985

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

AND-OR QUEUING IN EXTENDED CONCURRENT PROLOG[1]

Jiro Tanaka*[2], Takashi Yokomori**, Makoto Kishishita***[3]

* International Institute for Advanced Study of Social Information Science (IIAS-SIS)
Fujitsu Limited, 1-17-25, Shinkamata, Ohta-ku, Tokyo 144, Japan

** IIAS-SIS, Fujitsu Limited 140 Miyamoto, Numazu-shi, Shizuoka 410-03, Japan

*** Fujitsu Social Science Laboratory, 7-5-9, Nishigotanda
Shinagawa-ku, Tokyo 141, Japan

ABSTRACT

We have modified Concurrent Prolog (CP) Interpreter (Shapiro 1983) and implemented Extended Concurrent Prolog (ECP) Interpreter (Fujitsu 1985), which has OR-parallel, set-abstraction and meta-inference facilities. In Shapiro's CP interpreter only the AND-related goals are enqueued to the scheduling queue. None of OR-related clauses is dealt with. However, our ECP interpreter has only one scheduling queue to which all the AND-related goals and all the OR-related clauses are enqueued. This scheduling method is designated "AND-OR queuing." AND-OR queuing makes it possible to handle all kinds of AND-relations and OR-relations in a uniform manner.

1 INTRODUCTION.

Concurrent Prolog (CP) (Shapiro 1983) is a parallel logic language which includes a commit operator and read-only annotation as language constructs. We have extended Shapiro's Concurrent Prolog (CP) Interpreter and implemented Extended Concurrent Prolog (ECP) Interpreter (Fujitsu 1985), which has OR-parallel, set-abstraction and meta-inference facilities. A "scheduling queue" is often used in implementing a parallel logic language on a sequential machine. Processes reduced in parallel are enqueued to the scheduling queue. They are dequeued from the queue and reduced one by one. In this paper, focusing on the role of the "scheduling queue," we outline the implementation method for realizing extended features, and show how one can nicely handle those features in a uniform manner.

2 EXTENDED CONCURRENT PROLOG.

As mentioned above, ECP is an extension of CP with OR-parallel, set-abstraction and meta-inference features. Each feature is based on the conceptual specification of Kernel Language Version 1 (KL1) (Furukawa 1984). We briefly explain these features in the following sections.

2.1 AND-parallelism and OR-parallelism

AND-parallelism and OR-parallelism are the basic parallel inference mechanisms of ECP. AND-parallelism is the mechanism which evaluates AND-related goals in parallel. This

[1] This research has been carried out as a part of Fifth Generation Computer Project.

[2] Current address: ICOT Research Center, Institute for New Generation Computer Technology Mita-kokusai-building 21F, 1-4-28, Mita, Minato-ku, Tokyo 108, Japan

[3] Current address: IIAS-SIS, Fujitsu Limited, 1-17-25, Shinkamata, Ohta-ku, Tokyo 144, Japan

function can be realized by enqueuing goals to the tail of the scheduling queue, dequeuing a goal from the head of the queue, and enqueuing the newly created goals to the tail of the queue. This AND-parallelism has already been implemented in Shapiro's Interpreter. On the other hand, OR-parallelism is the mechanism which realizes the parallel evaluation of guards, when there exists more than one potentially unifiable clause with the given goal. This OR-parallelism was not implemented in Shapiro's Interpreter. The following program is an example of exploiting OR-parallelism.

```
solve(P,Mes):- call(P) | ...  .
solve(P,Mes):- find_stop(Mes) | ...  .
```

When "solve" is called, the above two clauses are executed in parallel by OR-parallelism. The first clause executes "P." However, as soon as "stop" is found in "Mes" in the second clause, the second clause is committed and the first clause is aborted. This realizes the "solve" with abort.

2.2 Set-abstraction

Set-abstraction is a mechanism for realizing the all-solution-search feature in a parallel environment. The following two predicates have been proposed (Fujitsu 1984).

```
eager_enumerate({X|Goals}, L)
lazy_enumerate({X|Goals}, L)
```

In the above description, "Goals" is the sequence of the goals defined in a Pure Prolog world. We assume that the Pure Prolog world is defined as follows:

```
pp(( <head> <- <body> )).
```

That is, the Pure Prolog world is asserted as the set of "facts" which have a functor name "pp."

These two "enumerate" predicates solve the Goals in the Pure Prolog world and put the set of all solutions in L in stream form. The following is an example of "eager_enumerate."

```
eager_enumerate({X | grand_child(jiro,X)}, L)
```

We assume that the Pure Prolog world is defined as follows:

```
pp((grand_child(X,Z) <- child(X,Y),child(Y,Z))).
pp((child(jiro,keiko) <- true)).
pp((child(yoko,takashi) <- true)).
pp((child(jiro,yoko) <- true)).
pp((child(keiko,makoto) <- true)).
```

In this case, L is instantiated as [takashi,makoto].

The difference between "eager_enumerate" and "lazy_enumerate" is the way it instantiates the second argument. "eager_enumerate" instantiates it actively. "lazy_enumerate" instantiates it passively in accordance with the request from the stream consumer. In the following example, a solution list "L" is created in accordance with the request from "display."

```
:- lazy_enumerate({X | prime(X)}, L?),
     display(L, Mes?), keyboard(Mes).
```

2.3 Meta-inference

Meta-inference means to solve a given goal using knowledge defined in a user-defined world (Furukawa 1984). We set up the predicate "simulate" with the following form.

```
simulate(World, Goals, Result, Control)
```

Here, "World" is the name of a world, "Goals" is the goal sequence to be solved, "Result" is the computation result, and "Control" is the stream through which we can stop and resume the computation. We assume that knowledge of the world is given as a set of facts whose principal functors are the name of the world. That is, knowledge of the world has the following format.

```
world_name((<Head> <- <Guard> | <Body>)).
```

As an example of meta-inference, we give the "shell" example (Clark 1984) which can run the foreground and background jobs. In this example, the foreground job always checks its control information while running. The background job runs steadily without looking up its control information.

```
shell([], _).
shell([fg(G)|N],C) :-
     simulate(f_world,G,R,C)&
     remove(C, NewC)&
     shell(N?,NewC).
shell([bg(G)|N],C) :-
     simulate(b_world,G,R,_),
     shell(N?,C).

:- shell([bg(primes),fg(primes)],C), control(C).
```

In this example, the "primes" programs to compute the infinite sequence of prime numbers runs both foreground and background jobs. Execution of the foreground job can be controlled by "C."

3 THE IMPLEMENTATION OF ECP

We have explained the extended features of ECP in the previous section. However, Shapiro's interpreter is not enough to realize these features. We need to implement AND-relations and OR-relations well using a scheduling queue.

In Shapiro's interpreter, a scheduling queue only contains AND-related goals. It is created for each OR-relation. Therefore, many local scheduling queues are created at program execution time. After extensive consideration, we have decided to make scheduling queues global. In our approach, only one global scheduling queue is created. All AND-related and OR-related goals are contained in one scheduling queue. We can imagine that the AND-OR tree created at program execution time is encapsulated in this scheduling queue.

We have named this scheduling method "AND-OR Queuing." Using this method, it

becomes possible to handle all kinds of AND-relations and OR-relations in a consistent way. In this section, we describe the queuing method for each feature.

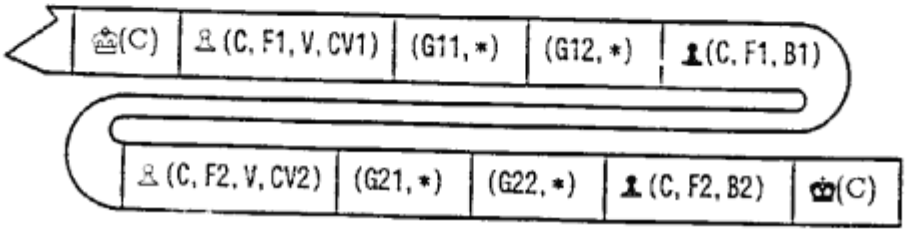
3.1 AND-parallelism and OR-parallelism

As we mentioned before, AND-parallelism has already been satisfied by enqueuing AND-related goals in the scheduling queue. To deal with OR-parallelism, we have decided to enqueue OR-relations sandwiched in between two kinds of markers.

For example, assume that the head of the scheduling queue is a goal "P" and the potentially unifiable clauses for "P" are as follows:

```
P1 :- G11, G12 | B1.
P2 :- G21, G22 | B2.
```

In this case, we put goals at the tail of the scheduling queue as follows:



Here, OR-clauses are sandwiched in between the markers and . The guard part of each clauses is placed between the markers and . Notice that markers and express the OR-relation and that markers and express the AND-relation. The symbol "*," the second argument of each goal, shows that the goal should be solved by using the global database world. The argument C, common to all markers, contains the information whether one of the OR-clauses is committed or not. The argument Fi of the marker shows whether the i-th OR-clause has failed or not. Note that this argument only needs to show that the i-th OR-clause has failed. Since the i-th OR-clause is committed as soon as the i-th OR-clause succeeds, it does not need to show that the i-th argument succeeded. The argument V is a list of variables which contains all variables in the original goals. The argument CVi is the copied list of V. The argument Bi of the marker is the body part of each clause.

Goals between markers are processed in exactly the same manner as the ordinary goals when goals are picked up from the scheduling queue. However, when markers are picked up, they are processed as follows:

(1) When marker (C) or (C) is picked up, the marker is aborted if "committed" is set in argument "C." Otherwise, the marker is put on the tail of the scheduling queue.

(2) When marker is picked up and the top of the queue is marker , i.e., the markers and are neighbors, this shows that all guards failed for a given goal. Since the "failure" of all guards means the "failure" of the given goal, "failure" is transmitted to the AND-relations to which they belong.[1]

[1] If the goal is at the top level, it means the total failure of the computation. For more detailed description, see the last paragraph of 3.3.

(3) When marker ♙(C,Fi,V,CVi) is picked up, it checks whether "committed" is set in argument "C" or "failed" is set in argument "Fi." In these cases, all goals from ♙ to ♟ are removed from the scheduling queue.

(4) When marker ♙(C,Fi,V,CVi) is picked up and the top of the queue is marker ♟(C,Fi,Bi), i.e., the markers ♙(C,Fi,V,CVi) and ♟(C,Fi,Bi) are neighbors, it means that all goals of a guard succeed. In this case, we set "committed" to the argument C, unify V and CVi, and schedule Bi.

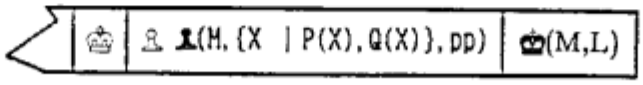
(5) When marker ♟(C,Fi,Bi) is picked up, the marker is simply put on the tail of the scheduling queue.

3.2 Set Abstraction

We consider the case where the following goal is taken from the scheduling queue.

```
eager_enumerate({X| P(X),Q(X)}, L)
```

In our implementation, the goal is appended to the tail of the scheduling queue in the following way.



Two pairs of markers appear again. The meanings of these markers are slightly different from the previous ones. However it is still true that the markers ♔ and ♔ express OR-relation, and the markers ♙ and ♟ express AND-relation. The markers ♔ and ♔ surround the OR-relation and work as a solution collector. The solutions are collected in "M" in stream form. The markers ♙ and ♟ compute one solution. The computed value is substituted into the argument "M."

When markers are taken from the scheduling queue, they are processed as follows:

(1) When marker ♔ is picked up and the top of the queue is marker ♔, i.e., the markers ♔ and ♔ are neighbors, this means that all solutions for the given goal have already been computed. We put [] onto the tail of the argument "L" in this case.

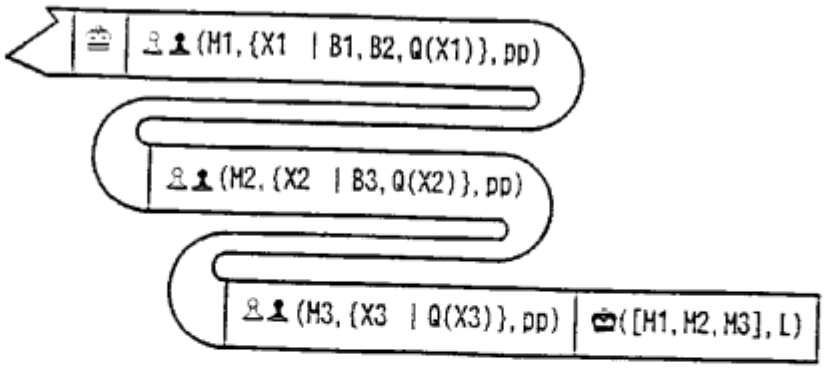
(2) When marker ♙ ♟(M,{X | ...}, pp) is picked up, we find definition clauses for the leftmost goal of this set. If more than two clauses are found, it is broken up into several goals. The argument "M" is also reproduced by fission.

(3) When marker ♔(M,L) is picked up, the argument "M" is checked. If it is instantiated, its value is sent to the stream "L" and the marker is appended to the tail of the scheduling queue.

The following is an example of fission. Assume that the marker is picked up, and P is defined in the Pure Prolog world as follows:

```
pp((P(X) <- B1,B2)).
pp((P(X) <- B3)).
pp((P(X) <- true)).
```

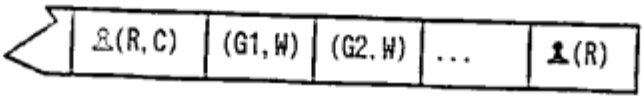
There are three clauses. The marker ♙ ♟ breaks up into three goals and they are appended to the scheduling queue in the following form:



We can get all solutions for the given goal by invoking fission. Notice that the solutions are computed by the depth-first search based on OR-parallelism. We have explained all-solution computing in the case of "`eager_enumeration`." The basic mechanism for "`lazy_enumeration`" is exactly the same as that for "`eager_enumeration`."

3.3 Meta-inference

We assume that the goal "simulate" is taken from the scheduling queue. In our implementation, the goal is put on the tail of the scheduling queue in the following form.



Markers and appears again. The arguments "R," "C" and "W" express the Result, Control, and World name, respectively.

The following summarize the actions when markers are taken from the scheduling queue.

(1) When marker (R,C) is picked up and "failure" is already set in argument "R," all goals from to are removed from the scheduling queue.

(2) When marker (R,C) is picked up and the top of the queue is marker , i.e., it is empty between marker and marker , we set "success" to the argument "R."

(3) When marker (R,C) is picked up and "C" is instantiated as `[ ..., abort | variable]`, all goals from to are removed from the scheduling queue and "abortion" is set to the variable "R."

(4) When marker (R,C) is picked up and "C" is instantiated as `[ ..., stop | variable]`, all goals from to are enqueued onto the tail of the scheduling queue without reducing these goals.

(5) When marker (R,C) is picked up, it checks whether "C" is a variable or instantiated as `[ ..., cont | variable]`. In this case, the marker is just appended to the tail of the scheduling queue.

(6) When marker is picked up, the marker is appended to the tail of the scheduling queue.

Just as before, the markers and express AND-relation. If a goal between and fails, "failure" is set to "R." Goals between and are processed as exactly same as

the ordinary goals, except that goals are reduced in a specified world. No special problems are created if OR-parallelism, set abstraction and meta-inference are nested within each other.

4 ECP INTERPRETER

In previous sections we explained the various features of ECP and the realization of these features in the scheduling queue. We have extended Shapiro's CP Interpreter (Shapiro 1983) and created the ECP Interpreter. In this section, we will explain the details of this ECP interpreter.

4.1 Shapiro's CP Interpreter

As mentioned above, our ECP interpreter is based on Shapiro's CP interpreter. Shapiro's interpreter is written in Prolog. The following program is a simplified version of his interpreter. The actual implementation is more complicated since it includes deadlock detection, system predicates and debug/trace features.

```
(1) cp(A):-
        schedule(A, X-X, Head-Tail),
        solve(Head-Tail).
(2) solve([]-[]):-!.
    solve([A|Head]-Tail):-
        system(A), !, A,
        solve(Head-Tail).
    solve([A|Head]-Tail):-
        reduce(A, B),
        schedule(B, Head-Tail, NewHead-NewTail), !,
                solve(NewHead-NewTail).
(3) reduce(A,B):-
        guarded_clause(A, (G|B)),
        cp(G), !.
    reduce(A,A).
(4) guarded_clause(A,B):-
        copy_functor(A, A1),
        clause(A1, B),
        unify(A, A1).
(5) schedule(true, Head-Tail, Head-Tail) :- !.
    schedule((A,B), Head-Tail, Head2-Tail2) :- !,
        schedule(A, Head-Tail, Head1-Tail1),
        schedule(B, Head1-Tail1, Head2-Tail2).
    schedule(A, Head-[A|Tail], Head-Tail):- !.
```

The meaning of this program is as follows:

(1) To solve a CP goal, the goal must be scheduled into the scheduling queue first. The "solve" predicate actually solves the goal.

(2) The "solve" predicate solves goals in the scheduling queue expressed as a D-list. If the scheduling queue is empty, the process terminates. Otherwise, a goal is taken

from the queue. If the goal is a system predicate, it is executed and the rest of goals are solved. If the goal is not a system predicate, the goal "A" is reduced to the new goals "B," and "B" is scheduled to the scheduling queue.

(3) The "reduce" predicate solves user-defined goals. The "guarded_clause" predicate looks for a potential unifiable guarded clause for a given goal "A." We solve the guard part of the unifiable clause. If it succeeds, "B" is the body part of that clause. If it fails, it backtracks and "guarded_clause" looks for another candidate clause. When all candidate clauses have failed, the "reduce" predicate does nothing and the second argument is equated to the first argument.

(4) The "guarded_clause" looks for a potential unifiable clause "B" for a given goal "A." The "copy_functor" makes the copied goal "A1" by copying the top level functor from the given goal "A." The "clause" finds a potentially unifiable clause "B" from "A1." If this succeeds, the "unify" predicate unifies A and A1.

(5) The "schedule" predicate contains the given goals to a scheduling queue. As mentioned before, the scheduling queue is expressed as a D-list. The "schedule" predicate enqueues nothing if the given goal is "true." If the given goal is "(A,B)," "A" is scheduled first and "B" is scheduled next. If the given goal is "A," it is simply appended to the tail of the scheduling queue.

4.2 Our ECP Interpreter

We have extended Shapiro's CP Interpreter (Shapiro 1983) and created the ECP Interpreter. The differences between our ECP interpreter and Shapiro's interpreter are as follows:

- Shapiro's interpreter processes OR-relations by backtracking as shown in (3) in the previous subsection. Since the "reduce" predicate in (3) calls the top-level predicate "cp" for each OR-relation, one scheduling queue is created for each OR-relation as mentioned in section 3. We have implemented OR-relations by appending them to the global scheduling queue with markers.
- Shapiro's interpreter does not distinguish "fail" and "suspend" on processing goals. However, we distinguish them so that the failure of a guard can be handled in OR-parallelism.
- Shapiro's interpreter does not directly realize OR-parallel, set-abstraction and meta-inference features. We directly implemented these using one global scheduling queue.

Our ECP interpreter is written in DEC-10 Prolog on the DEC2060 and in C-Prolog on the VAX11/780. The interpreter consists of the scheduling part, the marker processing part, and the Pure Prolog processing part. In terms of program size, these are approximately 150 lines, 170 lines, 50 lines respectively. The processing speed is two or three times slower than Shapiro's system since our system realizes OR-parallelism. The slow speed is caused by the fact that so many markers and goals are contained in the scheduling queue when we have a nested guard.

Our system was created to meet ICOT's research objectives. Therefore, our system has still several problems, such as the consumption of memory, exhaustion of stack space and others.

5 ECP PROGRAM EXAMPLE

We examine the "shell" program in this section. This is a more realistic version of the shell program discussed in 2.3 (Clark 1984). The shell program in 2.3 can run only one foreground job and multiple background jobs. We can control the execution only for the foreground job. However, in our "realistic" version, there is no distinction between foreground and background jobs, so we can run and control multiple jobs at the same time. In this "shell" program, every job has a process-ID and the execution of jobs can be controlled by commands which include process-IDs. A job may be aborted, suspended and resumed. The realistic "shell" program is shown below:

```
(1) shell :- shell(I?, []), in(I).

(2) shell([proc(ID,Goals)|Input], IDlist) :-
       true | print_result(IDlist,IDlist1),
            print_process(ID,Goals),
            simulate(*, Goals, R, C),
            shell(Input?, [(ID,R,C)|IDlist1]).

    shell([wproc(ID,W,Goals)|Input], IDlist) :-
       true | print_result(IDlist,IDlist1),
            print_wproc(ID,W,Goals),
            simulate(W, Goals, R, C),
            shell(Input?, [(ID,R,C)|IDlist1]).

    shell([Com | Input], IDlist) :-
       otherwise | print_result(IDlist,IDlist1),
            print_com(Com),
            send(IDlist1, Com, NewIDlist),
            shell(Input?,NewIDlist).

(3) send(□,_,□).
    send([(ID,R,C)|IDList],Com,
            [(ID,R,NewC)|IDList]) :-
                 Com =..  [M,ID] | C = [M|NewC].
    send([(ID,R,C)|IDList],Com,
            [(ID,R,C)|NewIDlist]) :- otherwise |
                 send(IDlist,Com,NewIDlist).
```

The meaning of this "shell" program is as follows:

(1) "shell" is the top level predicate. It calls the two-argument-"shell" and "in."

(2) Two-argument-"shell" is the main part of this program. The first argument of "shell" is a stream which receives commands from the goal "in." The second argument is the list of processes controlled by "shell." This list of processes is called "IDlist." A process is expressed as (ID,R,C), where ID is an identifier of a process, R is a variable which sends a message to the outside, and C is a variable which controls its execution.

This "shell" behaves as follows:

- When it receives the message "proc(ID, Goals)" at its first argument, it calls "simulate," executes "Goals" in the global database world, and adds this process "(ID,R,C)" to the "IDlist."
- When it receives the message "wproc(ID, W, Goals)" as its first argument, it calls

"simulate," executes "Goals" in world "W," and adds this process "(ID,R,C) to the "IDlist."

- When it receives other commands, such as "stop(ID)," "cont(ID)" or "abort(ID)," as its first argument, it sends that command to the control variable of the specified process.

The predicates "print_process," "print_wproc" and "print_com" are used just for printing out the message which "shell" received. The predicate "print_result" is used to print out the result when a process is aborted or ends successfully. In such cases, it prints out the process termination information and removes that process from the given "IDlist."

(3) "send" transmits a message such as "stop(ID)," "cont(ID)" or "abort(ID)" to the process with a process identifier "ID." It looks for the "IDlist." If it finds the process, it sends the message to the control variable of that process.

The following is an execution example of this "shell" program.

```
?- solve(shell, R).
> proc(p01,primes)
> wproc(p02,s,prime(10))
2
3
        2 (s)
        3 (s)
5
> stop(p01)
        5 (s)
        7 (s)
> cont(p01)
> result([p02,success])
7
11
> abort(p01)
> result([p01,abortion])
```

Here, we invoked two processes. One is the process "p01" which generates the infinite sequence of prime numbers. The second is the process "p02" which computes prime numbers up to 10 following the definition of prime in the world "s." In this example, we stopped "p01" after it printed out 2, 3 and 5, and resumed after "p02" printed out 2, 3, 5 and 7. The process "p02" is terminated after it prints out all primes up to 10. We also terminated process "p01" by sending the abort message to "p01."

6 RELATED WORKS

Here, we would like to survey various research on extended features of Concurrent Prolog.

(1) For OR-parallelism, Levy (1984) proposed the implementation method using a global queue. His method is based on the lazy copying scheme, but it has been pointed out that this method still has bugs. ICOT also tried OR-parallelism using several implementation schemes (Miyazaki 1984; Sato 1984; Tanaka 1984). For these methods, implementations were written in Pascal or Lisp. On the other hand, our implementation was done in a logical way using Prolog.

(2) The research in set abstraction is preceded by POPS (Hirakawa 1984). POPS is a Pure Prolog interpreter written in Concurrent Prolog. It enumerates all solutions for the given goals in stream form. In our approach, the enumeration of all solutions is directly realized by the scheduling queue.

(3) The key issue in meta-inference is how to implement the interpreter of the target language. In this field, research has been done by writing meta-interpreters (Shapiro 1984, Clark 1984). We have implemented meta-inference predicates directly onto the scheduling queue. Compared with the traditional approach, our approach is more direct.

7 CONCLUSION

In this paper, we described the rough outline for realizing extended features of ECP. Related work in this field was also surveyed. Although we have omitted here, there are various problems which occur in the actual implementation. One is the problem of copying variables involved in the realization of OR-parallelism.

We proposed the "AND-OR queuing" method. It is surprising that the various features of ECP, such as OR-parallel, set-abstraction and meta-inference, can be implemented in a consistent manner. From the architectural point of view, it is more realistic to assume one global queue than assuming many local scheduling queues created dynamically. And it leads to the more consistent scheduling. That is,

(1) It realizes OR-parallelism.

(2) In set abstraction, we can reduce other goals while generating solutions.

(3) In meta-inference, we can compute several "simulate" predicates at the same time.

By the way, the scope of this "AND-OR queuing" method is not limited to Concurrent Prolog. This method is also applicable to GHC (Ueda 1985). In this case, implementation is simpler because it does not generate multiple environments in implementing OR-parallelism.

ACKNOWLEDGMENTS

This research was carried out as a part of the Fifth Generation Computer Project. T.Yokomori and J.Tanaka chiefly designed the ECP interpreter. M.Kishishita actually coded the ECP interpreter.

We would like to thank Akikazu Takeuchi, Kazunori Ueda, and other members of the KL1 implementation group at ICOT for their useful comments and suggestions. We would also like to thank Dr. Furukawa, the chief of the First Research Laboratory, ICOT, Dr. Kitagawa, the president of IIAS-SIS, Fujitsu, Dr. Enomoto, the director of IIAS-SIS, Fujitsu, and Mr. Yoshii, Fujitsu Social Science Laboratory, for giving us the opportunity to pursue this research and helping us with it.

REFERENCES

Clark K, Gregory S (1984) Notes on Systems Programming in Parlog. Proceedings of the International Conference on Fifth Generation Computer Systems 299-306

Fujitsu (1984) The Verifying Software of Kernel Language Version 1 – Detailed Specification–, PART II. In: The 1983 Report on Committed Development on Computer Basic Technology, in Japanese

Fujitsu (1985) The Verifying Software of Kernel Language Version 1 – the Revised Detailed Specification and the Evaluation Result–, PART I. In: The 1984 Report on Committed Development on Computer Basic Technology, in Japanese

Furukawa K et al. (1984) The Conceptual Specification of the Kernel Language Version 1. Technical Report TR-054. ICOT

Hirakawa H et al. (1984) Eager and Lazy Enumeration in Concurrent Prolog. Proceedings of the Second International Logic Programming Conference 89-100

Levy J (1984) A Unification Algorithm for Concurrent Prolog. Proceedings of the Second International Logic Programming Conference 333-341

Miyazaki T et al. (1985) A Sequential Implementation of Concurrent Prolog Based on Shallow Binding Scheme. Proceedings of 1985 Symposium on Logic Programming 110-118

Sato H et al. (1984) A Sequential Implementation of Concurrent Prolog - based on the Deep Binding Scheme. Proceedings of the First National Conference of Japan Society for Software Science and Technology 299-302, in Japanese

Shapiro E (1983) A Subset of Concurrent Prolog and its Interpreter. Technical Report TR-003. ICOT

Shapiro E (1984) Systems Programming in Concurrent Prolog. Conference Record of the 11th Annual ACM Symposium on Principles of Programming Language 93-105

Tanaka J et al. (1984) A Sequential Implementation of Concurrent Prolog – based on the Lazy Copying Scheme. Proceedings of the First National Conference of Japan Society for Software Science and Technology 303-306, in Japanese

Ueda K (1985) Guarded Horn Clauses. Technical Report TR-103. ICOT