ICOT Technical Report: TR-557

TR-557

KL1における永久中断ゴールの検出と報告

大西　諭，稲村　雄

May, 1990

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

KL1 における永久中断ゴールの検出と報告

大西 諭　　稲村 雄

新世代コンピュータ技術開発機構

概要

KL1 は、第5世代プロジェクトで開発中の並列推論計算機の核言語として開発された committed-choice 型の言語である。Committed-choice 型の言語では、共有変数の具体化によりプロセス間の同期を行なうが、プログラムを間違うと簡単に永久中断ゴールが発生する。処理系がその検出・報告を行なう機構がないと、プログラムの間違いの発見や修正は困難である。本稿では、一括 GC を用いた永久中断ゴールの検出及び報告機構について述べる。本アルゴリズムでは、永久中断ゴールの因果グラフの中から極大ゴールのみを抽出して報告している。これらは、Multi-PSI 上に実装されている。

Detection and Report of Perpetual Suspension in KL1

Satoshi ONISHI　　Yū INAMURA

Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

Abstract

KL1 is a committed-choice language, designed as the kernel language of the parallel inference machines which are under development in the Japanese fifth generation project. In a committed-choice language, execution of concurrent processes is controlled by shared variables, but it is easy to cause perpetual suspension because of some mistakes in the programs. It is difficult to find out mistakes and correct them in the programs without detection and report mechanism in the implementation. This paper presents an algorithm to detect perpetually suspended goals at garbage collection time. The algorithm detects maximal goals in perpetually suspended goals by searching their causality graph. The mechanism has already been implemented on the Multi-PSI.

1 はじめに

KL1 は、Flat GHC[12] に基づく committed-choice 型言語であり、新世代コンピュータ技術開発機構 (ICOT) で開発中の並列推論マシン Multi-PSI [6] および PIM [6] の核言語として設計された。

他の committed-choice 型言語 CP [10]、PARLOG [4]、Strand [5] 等と同様、KL1 では全ての AND ゴールが同時並列に実行される。そして、プロセス間の通信と同期は、それらのゴール間で共有される変数に対する具体化と待ち合わせの機構によって実現される。たとえば、committed-choice 型言語における一つの典型的なプログラミングスタイルである producer/consumer モデルでは、producer と consumer が共有変数 (ストリーム) をもち、consumer の実行は共有変数の具体化を待つことにより、その変数を具体化する producer の実行に同期される。

committed-chioce 型言語の場合、このような同期機構が言語レベルでサポートされているため、プログラマが同期に注意する必要のある他の並列言語に比べて生産性が高く誤りも少ない。たとえば、現在 ICOT では、Multi-PSI 上の OS PIMOS [3] を KL1 で記述している。その開発過程で、最初 1 台のプロセッサで動かしてバグを取り除いて、その後複数台で実行を行なったが、その場合でも同期のバグには出会っていない。これは、プログラム開発上の大きな利点となっている。

しかし、KL1 のようなデータフロー言語で問題となりやすいのは、プログラム中の誤りにより実行が中断することがある、ということである。その例を次に示す。

```
?- producer(X), consumer(X).

producer(X) :- true | X = [msg|X2], producer(X2).
consumer([msg|X]) :- true | consumer(X).
```

図 1: Producer/Consumer プログラムの例

図 1は、KL1 で書かれた簡単な producer/consumer プログラムである。もし、このプログラムで、producer/1 が次のように書かれていたとする。

```
producer(X) :- true |
        Y = [msg|X2], producer(X2).
```

この場合、共有変数 X は具体化されず、その結果、consumer/1 は、誰からも具体化されない変数の値を待っていることになり、永久に中断してしまう。このようなゴールを永久中断ゴールといい、またこのような状態を永久中断状態と呼ぶ。

もちろん、ここで示した例は、単純なタイプミスによるものであり、この場合は 1 クローズ中にそれぞれの変数が何回出現しているかを調べる“変数チェッカー”のような静的な解析によって発見することができる。しかし、このような静的な解析では発見できない複雑な場合も多く、プログラム実行時に動的に発見する機構が必要になる。

もう一つ考慮しなければならないのは、永久中断ゴールがチェインを作るという問題である。

普通、consumer のゴールは、他の consumer に対する producer である (すなわち、フィルタである) 場合が多いため、一つのゴールが永久中断状態になると、他のゴールもそのために永久中断になることが多いのである。通常のアプリケーションでは、ゴール間の依存性は複雑であるため、多くのゴールが永久中断状態になってしまう。

そのため、プログラムが中断した状態では、多くの永久中断ゴールが存在し、かつ、その中で本当に永久中断の原因になっているゴールはわずかしかない、という厄介な状態になるのである。

本稿では、実際に Multi-PSI 上に実現されている永久中断ゴール発生の検出のアルゴリズムについて述べる。このアルゴリズムは、一括 GC を利用したもので、永久中断の発生の検出のフェーズと、永久中断の因果関係における極大ゴールを発見するフェーズに分けられる。次章以降で述べるように、これらの極大ゴールは永久中断の真の原因と考えられるため、デバッグに非常に役立つ機能であるといえる。

2 永久中断ゴールの報告

2.1 KL1 での例外処理機構

永久中断の発生を検出した場合、ユーザのデバッグを助けるためにその事実を知らせる必要がある。この永久中断は一種の例外事象とみなし、他の例外事象 (リダクション失敗、組み込み述語不正入力等) と同様に取り扱う方針とした。KL1 には、これらの例外事象を扱うために荘園 [3] というメタプログラミング機能が組み込まれている。図 2にこの荘園機構の論理的な構造を示す。全てのゴールは荘園の下で実行され、荘園自身の実行はコントロールストリームを通して制御される。荘園内部の実行の状態は、レポートストリームを通じて報告されることになる。荘園を制御するプロセス (コントロールプロセスと呼ぶ) は、レポートストリームを監視し、状況に応じてコントロールストリームから適当なメッセージを送って荘園の実行を制御する。例えば、荘園内の実行

を中断したければコントロールストリームに stop を流せばいいし、荘園内の実行が終了すれば、terminated がレポートストリームに流れる。

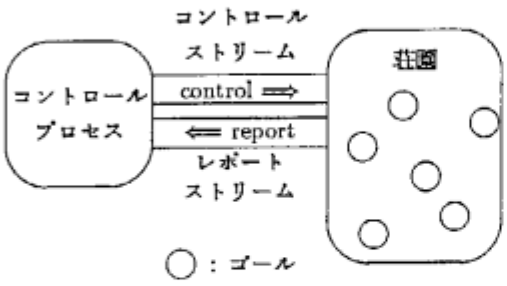

図 2: 荘園

2.2 永久中断ゴールの報告形式

永久中断ゴールが発見されると例外事象としてレポートストリームに以下のような形式で報告される。

```
exception(Info, Goal, ^NewGoal)
```

それぞれの引数は、次の通り

Info: 例外発生の原因

Goal: 例外を発生させたゴールの情報 — Code へのポインタと引数ベクタで表される

^NewGoal: Goal の代わりに実行するゴールを指定する変数 — Code へのポインタと引数ベクタを与える

例外報告を受けとったコントロールプロセスは、NewGoal を具体化することにより、例外に対する実行の制御を行なう。

ここで、永久中断ゴールの報告について、考慮すべき点は、

- すべての永久中断ゴールを報告すべきではない。永久中断の真の原因となったゴールのみを報告すべきである。

ということである。このために、処理系は永久中断の原因となった極大ゴール[1]のみを報告する。極大ゴールは、永久中断ゴール群の因果関係の中で最も“因”の位置にあるゴールなので、永久中断ゴール群の中から極大ゴールのみを抽出・報告することは、効率的なデバッグ環境の実現という点で非常に意義が高いといえる。真に永久中断の要因であるゴールが、他の膨大なゴール群に紛れてしまい、特定し難くなるということがなくなるためである。

ここで注意しなければならないのは、報告された極大ゴール以外の永久中断ゴールは、例外報告を行なった後では、もう永久中断状態にはないということである。何故ならば、極大ゴールの引数が報告されているため、コントロールプロセスにはその引数の中のどれかを具体化することにより、中断中の非極大ゴールの実行を再開することが可能だからである。このことは、永久中断ゴールの中で極大ゴールのみを報告すべきであるという方針の理由の一つになっている。

3 KL1 の中断機構と因果グラフ

3.1 KL1 の中断機構

KL1 処理系の実装においては、実行を効率化するために、ゴールの中断を busy-wait によらない方法で行なっている。すなわち、あるゴールの実行が具体化されていない変数のために中断する場合、そのゴールの引数やコードへのポインタなどの環境を格納したゴールレコードを中断の原因となった未定義変数からポインタで指させる (変数へのゴールのフックという)。その後、変数が具体化される時に、変数にフックされたゴールは実行可能な状態となり、スケジュールされるのを待つことになる。

```
?- a(X,Y,Z), b(Y), c(Z).

a([msg1|X],Y,Z) :- true |
     Y = [msg2|Y1], Z=[msg3|Z1], a(X,Y1,Z1).
b([msg2|Y]) :- true | b(Y).
c([msg3|Z]) :- true | c(Z).
```

図 3: 永久中断ゴール例

例えば、図3に示したプログラム例では、ゴール a/3、b/1、c/1 は永久中断し、図4のような構造を形作る。ここで、a/3 が因果関係における極大ゴールである。

図4から、中断ゴールレコードとその引数自体が中断の因果関係のグラフを表現し、因果関係において下流にあるゴールへは上流にあるゴールの引数からポインタをたどっていくことが可能であることが明らかである。このような関係を、到達可能であるという。

[1]次章で説明

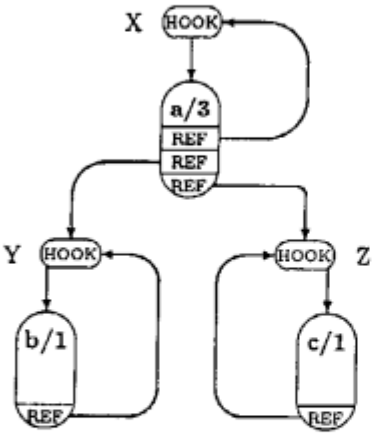

図 4: ゴールの状態

3.2 極大ゴール

ここで、極大ゴールを定義するためにゴール間の関係を表す記法

- $A \succ B$ もしくは $B \prec A$

を導入し、これを『ゴール B はゴール A の引数から到達可能である』と読む。さて、ゴール A とゴール B の間に

- $(A \succ B) \wedge (A \not\prec B)$

という関係がある場合、因果関係においてゴール B はゴール A の因果関係の下流にある。また、

- $(A \succ B) \wedge (A \prec B)$

という関係が成立している場合、因果関係においてゴール A とゴール B は同値であるといえる。この記法を使用することにより永久中断ゴール G_p は

- $(\forall$ 実行可能ゴール $G_e), G_p \not\prec G_e$

のようなゴールであると定義できる。

以上の関係を考慮すると、永久中断ゴール群中の極大ゴール(群)とは、永久中断ゴール群を因果関係において同値であるゴールの集合 G_i に分割した後、それらの中で

- $(\forall G_i), G_j \not\prec G_i$

という条件を満たす同値類 G_j のことであるといえる。

4 極大ゴール発見アルゴリズム

ここでは、前節で定義した永久中断ゴール中の極大ゴール発見アルゴリズムについて説明する。アルゴリズムは

1. 永久中断状態の検出
2. 永久中断ゴール(群)の発見
3. 極大ゴール抽出

の3つのフェーズに分かれる。なお、本アルゴリズムは、コピーイング GC を実装している Multi-PSI 上での実現を想定している。

4.1 永久中断状態の検出

KL1 処理系では、通常実行中に永久中断ゴールを全て検出することは非常に困難である[2]。しかし、Non Busy-waiting 方式でゴールの実行中断を行なっているため、永久中断ゴールは通常の実行可能ゴールの引数からは到達不可能であるという性質がある。この事実を利用して、コピーイング方式による局所 GC 時に永久中断状態の検出を行なうことにした。コピーイング方式の GC とは、メモリ空間を二つに分割し、それを交互に使用し、一方のメモリ空間を使い切ると、その中のアクティブなデータ構造のみをもう一方のメモリ空間にコピーする方式である。局所 GC では実行可能なゴールのみをマーキングルートとするために、永久中断ゴールは発見不可能であり、故に GC 前のゴール数と GC 中にコピーしたゴール数を比較することで永久中断の発生が検出できるのである。

具体的には以下の処理を行なう。まず、通常実行時と局所 GC 時にゴール数を数えるカウンタ *goal_counter* と *copied_goal_counter* を設ける。通常実行時にはゴール生成・実行終了時に *goal_counter* をカウントする。GC 時には、ゴールをコピーするたびに *copied_goal_counter* をカウントする。ただし、通常実行用の *goal_counter* に関しては、計算の終了を検出するために既に導入済みであるため、通常実行時の処理には全くオーバヘッドはかからない。局所 GC 終了時にこの二つのカウンタを比較して、*copied_goal_counter* が *goal_counter* よりも小さければ、永久中断状態に陥っていることが検出されたことになる。

[2]限られた場合のみ MRB を用いて検出が可能である。

4.2 永久中断ゴール(群)の発見

局所GCでマーキングルートから到達可能なデータオブジェクトを旧領域から新領域へコピーした後、永久中断ゴールはコピーされずに旧領域に残ることになる。次のフェーズでは、この旧領域をスイープして永久中断ゴールを探す必要がある。このようなスイープを可能にするために、ゴールレコードにしか現れないような特殊なタグ GOAL を導入し、ゴールレコードと他のデータとの識別に利用する。

永久中断したゴールはその引数情報も含めて荘園に報告されるため、それらのデータは通常のデータと同様、GCにおける新領域にコピーしなければならない。そのため、スイープで永久中断ゴールを発見した場合には、そのゴールをマーキングルートとして旧領域から新領域へのコピーを行なう必要がある。この際、以下に述べるように永久中断ゴール間の依存関係の一部が判明するために、このアルゴリズムは非常に効率的なものになっている。

4.3 極大ゴール抽出

最後に、極大ゴール抽出のためのアルゴリズムについて述べる。このアルゴリズムは sweep-and-copy フェーズと mark-and-sift フェーズに分かれている。以下にそれぞれのフェーズにおける処理について説明する。

4.3.1 Sweep-and-copy フェーズ

このフェーズでは前節で説明したように、旧領域のスイープ & コピーを行なうことにより、全ての永久中断ゴールを新領域にコピーする。処理は、図5の通りである。

極大ゴール候補テーブルは、実際のインプリメントでは旧領域中のゴールレコードのリストとして実現されている。旧領域から新領域へデータをコピーしている最中は、ゴールレコードをコピーする毎に copied_goal_counter をインクリメントしていく。

このフェーズ終了時点での新旧領域の状態は図6のようになっている。

この図中で G_i はメモリ・スイープで発見された極大候補ゴールであり、$CP-G_i$ は G_i をマーキングルートとしたコピーによって新領域に移されたデータオブジェクトの集合である。また、$i<j$ であれば G_i は G_j の前に発見されていることになる。

さて、ここで極大候補ゴール G_i の間に

- $G_i \not\prec G_j$, if $i<j$

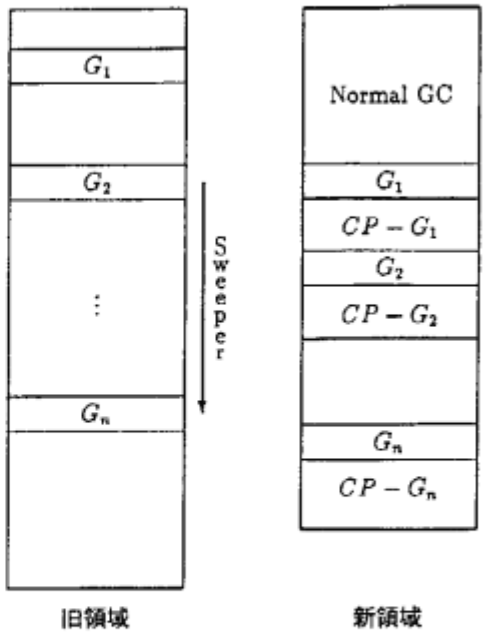

図 6: Sweep-and-Copy フェーズが終了した時の新旧メモリ領域

なる関係があるのは明らかであろう。このことは以下のようにして簡単に証明することができる。

> もし、G_j が G_i に依存し、かつ、$i<j$ ならばゴールレコード G_j は G_i をマーキングルートとしたコピーで発見されていなければならない。しかし、$i<j$ であるから、G_j はそのコピーで発見されていない。故に G_j の実行は G_i に依存しないのである。

特に、図6中のゴール G_n はこのフェーズ終了時点で極大ゴールであることが保証されている。G_n の実行は $G_1,...,G_{n-1}$ に依存しないからである。

4.3.2 Mark-and-sift フェーズ

このフェーズでは前フェーズで確かめられた因果関係と逆向きの

- $G_i \prec G_j$, $i<j$

なる関係があるか否かを確かめる。

真の極大ゴールは『ゴール G_j の引数からデータをトレースし、到達可能なゴール $G_i(i<j)$ を探す』という方法で、極大ゴールテーブルを篩にかけることによって発見できる。同じデータを複数回トレースすることを避けるために、GCに使用されている marking bit を用いて、一度トレースされたデータにはマークをつけることにした。アルゴリズムは図7の通りである。

```
procedure Sweep-and-copy
begin
    while (goal_counter > copied_goal_counter) do
        begin
            旧領域をスイープする;
            if  ゴールレコード G を発見した then
                begin
                    G を新領域にコピーする;
                    G を極大ゴール候補テーブルに登録する;
                    copied_goal_counter をインクリメントする;
                    G を マーキングルートにして、到達可能なデータオブジェクトを全て新領域にコピーする
                        (この間にゴールレコードをコピーすれば、 copied_goal_counter をインクリメントする);
                end
        end
end
```

図 5: Sweep-and-copy アルゴリズム

```
procedure Mark-and-sift
begin
    for j := n downto 1 do
        begin
            if  G_j in 極大ゴール候補テーブル then
                begin
                    while G_j の引数からトレースを行ない、データオブジェクトにマークをつける;
                        begin
                            if  (極大ゴールの候補である G_i がトレース中に見つかった) ∧ (i ≠ j) then
                                begin
                                    G_i を極大ゴール候補テーブルから外す;
                                end
                        end
                end
        end
    メモリセルの mark-bit を off にする;
end
```

図 7: Mark-and-sift アルゴリズム

データのトレースは、一つのゴールの全引数をマーキングルートとして再帰的に行なう。このフェーズを行なう時点では旧領域が自由に使用できるため、旧領域をスタックとして用いた処理を採用した。

この処理が終了した時点で極大ゴール候補テーブルには真の極大ゴールのみが登録されていることが保証される。この処理によって

- $(G_i \prec G_j \wedge i < j)$

の関係にあるゴール G_i は候補テーブルから削除されるからである。

最後に、ここで発見された極大ゴールのみが永久中断ゴールとしてその属する荘園に報告される。

5 評価

5.1 永久中断ゴール検出のためのオーバヘッド

ここで、前章で示したアルゴリズムを実行することによるオーバヘッドに関して考察する。

- 通常実行時には、特別な機構を追加していないので、オーバヘッドはない
- 永久中断状態が発生していない場合、GC 中は copied_goal_counter をカウントするのに僅かのオーバヘッド $O(n)$ (n はゴールの数) だけかかる。
- 永久中断状態が発生している場合、旧領域のメモリスイープを行なうため sweep-and-copy に $O(m)$、mark-and-sift にも $O(m)$(m はメモリサイズ) のオーバヘッドがかかる。

以上のように、永久中断ゴールが存在しない場合は、局所 GC 時に僅かのオーバヘッドがかかるだけで、事実上ほとんどない。永久中断ゴールが存在する場合は、局所 GC 時にある程度のオーバヘッドがかかるが、このような場合は一般的にプログラムのバグにより発生する状態であり、永久中断ゴールを報告する利点を考慮すると問題になるコストではないと考えられる。

5.2 本方式の限界

本アルゴリズムにおける限界について述べる。

1. 検出が即時にできない。そのため、既に永久中断に陥った状態しか調べることができず、共有変数を具体化せずに終了してしまった producer プロセスを報告することができない。そのため、永久中断の真の原因を見つけ出すことが困難である。

   この問題点については、MRB [2][13] を用いた即時検出を行なうことによりある程度解決できる。

2. 因果関係グラフの中の真の極大ゴールが報告されないことがある。変数への参照が読み出しパスなのか書き込みパスなのかの区別を行なえば、真の極大ゴールが発見できる。

3. 現在、Multi-PSI には大域 GC はまだ実装されていないため、プロセッサ間に渡る永久中断状態は検出できない。しかし、一般的に永久中断ゴールが最もよく発生するのはデバッグの初期の段階であり、その段階では、普通 1 台のプロセッサで実行するため、本アルゴリズムでも十分に実用性があると考えられる。

6 おわりに

KL1 のような committed-choice 型の言語を用いてプログラムを作成する場合、最も多く発生するトラブルの一つが永久中断ゴールの発生である。そのため、永久中断ゴールを報告することは、プログラムのデバッグ作業の効率を飛躍的に高める可能性がある。このような機構がサポートされていなければ、永久中断状態に陥ってしまった原因を発見するには非常に大きな労力が必要であり、ソフトウェア開発の妨げになる。

今回述べた永久中断ゴールの検出方式は、通常実行時に特別な機構を用意することなく実現しているのが大きな特徴となっており、オーバヘッドも GC 時に僅かかかるだけである。

しかし、今回述べた方法では、すでに永久中断状態になってしまったゴールしか発見できない。もし、あるゴールが永久中断状態になった瞬間を検出することができれば、共有変数を具体化せずに捨ててしまった producer プロセスを発見することができ、デバッグが更に簡単になる。そこで現在、MRB [2][13] による検出方式を実装中であり、そうすれば永久中断ゴールの即時検出が実現できる。しかし、MRB の性質のため、全ての永久中断ゴールを発見できるわけではない。そこで、今回述べた局所 GC による検出機構と同時に実現すれば、それぞれの特徴に応じた検出ができ、その結果、効率的なプログラミングを行なえることが期待できる。

7 謝辞

最初に本アルゴリズムの考えを提起された近山隆氏に感謝します。処理方式の改良に御討論をいただいた市吉伸行氏、六沢一昭氏に感謝します。ICOT の淵一博所

長、内田俊一第4研究室室長にこの研究の機会を与えて下さったことを感謝します。さまざまな有益な助言をいただいたその他のICOT第4研究室の研究員の方々に感謝します。

参考文献

[1] H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–249, 1978.

[2] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, Vol. 2, pp.276–293, 1987.

[3] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230–251, ICOT, Tokyo, 1988.

[4] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Trans. on Programming Languages and Systems* 8(1) pp.1–49, 1986.

[5] I. T. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, N. J. 1989.

[6] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208–229, ICOT, Tokyo, 1988.

[7] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, pp 257–275, 1987.

[8] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, pp 468–477, 1987.

[9] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pp.436–451, 1989.

[10] E. Shapiro. *A Subset of Concurrent Prolog*. ICOT Technical Report TR-003, 1983.

[11] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[12] K. Ueda, and T. Chikayama. Efficient stream/array processing in logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.317–326, ICOT, Tokyo, 1984.

[13] 木村、近山: 並列論理型言語KL1の多重参照管理によるガベージコレクション, 情報処理学会論文誌,Vol.31, No. 2, pp.316-327 (1990).

[14] Y. Inamura, and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proceedings of the Seventh International Conference on Logic Programming*, (To appear), 1990.