並列論理型言語 KL1 の最適化手法

大野 和彦[†]，伊川 雅彦[†]，
森 眞一郎[†]，中島 浩[†]，富田 眞治[†]

[†] :京都大学 工学部 情報工学科
〒 606-01 京都市左京区吉田本町

現在、ICOT において並列論理型言語 KL1 処理系 KLIC が開発されており、様々な並列計算機での利用を目指している。しかしながら、現時点での KLIC の実装は、速度やメモリなど実行効率の面で手続き型言語に及ばない。そこで本研究では、静的解析による実行最適化手法を提案する。本手法はプロセス単位の処理により、効率的な処理が可能である。解析段階ではモード・タイプ解析によりデータ型や依存関係を抽出し、これを利用してプロセス内逐次実行の効率化や、プロセス間入出力の最適化コード生成による物理通信削減を実現する。

並列論理型言語、静的解析、最適化、KL1、KLIC

Optimization Method of Concurrent Logic Language KL1

Kazuhiko OHNO[†], Masahiko IKAWA[†],
Shin-ichiro MORI[†], Hiroshi NAKASHIMA[†], Shinji TOMITA[†]

[†] : Department of Information Science
Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606–01 Japan

E-mail: {ohno, ikawa, moris, nakasima, tomita }@kuis.kyoto-u.ac.jp

The KLIC system, a portable implementation of a concurrent logic programming language KL1, is being developed in ICOT. However, the current implementation is less efficient compared to procedural languages. We propose an optimization method using static analysis. Each process is dealt with independently for efficient mode-type analysis, which derives data types and dependencies. With the analysis result, efficient execution of process by optimal scheduling and reduction of physical messages by optimizing message generation are achieved.

concurrent logic programming language, static analysis, optimization, KL1, KLIC

1　はじめに

並列論理型言語 KL1 は、第五世代計算機計画の核言語として開発され、定理証明や遺伝子情報処理など数々の知識処理プログラムが作成されている。現在、ICOT において移植性の高い KL1 処理系 KLIC が開発されており、様々な並列計算機やワークステーション上での利用を目指している。

しかしながら、現時点での KLIC の実装は、速度やメモリなど実行効率の面で手続き型言語に及ばず、実用的な処理系実現のためには実行方式の改良が欠かせない。そこで本研究では、静的解析により実行を最適化する手法を提案する。

本手法ではプロセス単位の解析・最適化を行うため、処理コストは比較的小さくできる。解析段階ではモード・タイプ解析 [1],[2] により、生じるデータの型や依存関係を抽出する。続いてこの情報を利用し、サスペンションの削減によるプロセス内逐次実行の効率化や、プロセッサ間通信になりうるプロセス間入出力の最適化コードを生成する。

以下、2 章で KL1 と KLIC について説明し、3 章で今回提案する方式の概略を述べる。続いて、4 章および 5 章で、それぞれ静的解析および最適化の具体的手法を説明し、最後に 6 章で、まとめを行う。

2　背景

本節では、並列論理型言語 KL1 [3], [4] およびその処理系 KLIC [5] について簡単に説明する。

2.1　並列論理型言語 KL1 の概要

2.1.1　プログラム構造

KL1 は Flat GHC に基づく言語であり、プログラムは以下の形をした節 (クローズ) の集合で表される。

$$H \quad :- \quad G_1, \ldots, G_m | B_1, \ldots, B_n$$

H, G, Bは述語であり、それぞれクローズヘッド、ガードゴール、ボディゴールと呼ばれる。述語は、ヘッドの述語名、および引数の数が等しい節の集合で定義され、“述語名/引数の個数” の形で表記される。

実行の単位はゴールと呼ばれる。与えられたゴールHを述語と同一化し、その定義部を次の新たな実行ゴールとする。この過程を繰り返すことによって、プログラムの実行が行われる。

このとき、述語 Hを定義する各節のヘッド H_iとの同一化、ならびにガード部$G_{i1}, \ldots, G_{im}$の実行を、並列に行うことができる。ただし、H の引数を具体化することはできない。続いて、ガード部が成功した節のうち一つが選択され、そのボディ部 $B_{j1}, \ldots, B_{jn}$が実行される。

なお、KL1 では、ガードゴールに書けるのは組込述語のみという制限が付けられている。ボディゴールには、組込述語とユーザ定義述語を記述できる。また、実行開始時の初期ゴールとして、`main/0` が与えられる。

2.1.2　データ構造

KL1 のデータ領域は、セルと呼ばれる最小単位で管理される。アトミックデータ型にはアトムや整数があり、1 セルに値が直接格納される。構造データ型にはファンクタやリストがあり、複数のセルからなる。これらのセルには別のデータに対する参照ポインタを格納することができ、複雑なデータ構造を表現することが可能になっている。

変数は参照ポインタで表現され、値に関して特定の型は持たない。初期状態では未具体化で値を持たず、同一化により対象へのポインタが格納される。具体値との同一化 (具体化) が行われると具体値へのポインタが格納され、以後この変数を参照すると値を読み出すことができる。一度具体化された変数は、他の値に書き換えることはできない。

現在の KLIC にはベクタ型やストリング型なども実装されているが、本論では省略する。

2.2　KL1 処理系 KLIC の概要

KLIC は、現在 ICOT で開発が進められている KL1 の処理系である。これは KL1 プログラムをいったん C プログラムに変換し、ターゲットマシンのオブジェクトコード生成は、その計算機用の C コンパイラに行わせる方式を取っている。このため、物理通信などの機種依存部を除き移植性の高い処理系になっている。従来 KL1 プログラムは並列推論マシン PIM 上でしか動作しなかったが、KLIC により、様々な並列計算機やワークステーション上での利用が期待できる。

現在、ワークステーション上で PVM を用いた分散処理系が実装されている。以下、本論での KLIC 処理系とは、この実装を指すものとする。なお、幾つかの並列計算機への移植作業が、現在進行中である。

2.2.1　逐次実行方式

ある節が選択された時の変数操作やガード / ボディゴール呼び出しなどは、C 言語に変換され、最終的にプログラムコードとして格納される。実行時のゴール

インスタンスは、ゴールレコードとして表現され、対応するコードへのポインタと引数の内容などの環境を持つ。プログラムの実行は、実行キューにつながれたゴールレコードを取り出して対応するコードを実行し、その結果得られたボディゴールのレコードをキューにつなぐことで行われる。

実行しようとしたゴールで値を参照する引数が、未具体化変数 X だった場合、そのゴールレコードは X にキューとしてつながれ、次のゴールが実行される。のちに X が具体化されると、つながれていたゴールは再び実行キューに戻される。

2.2.2 並列実行方式

KLIC 分散処理系の計算モデルは、共有メモリを持たない複数のノードが、互いにメッセージ通信をしながら処理を行うというものである。

並列実行は`@node`プラグマを用いて明示的に記述するようになっており、$goal(X_1, \ldots, X_n)$`@node(`N_c`)`という形のボディゴールがノード N_p 上に現れると、このゴールはノード N_c に送られ、そこで実行される。このとき、未具体化の変数 X_i については、N_p 上の変数を指す外部参照ポインタが N_c に送られる。

N_c 上で X_i の値が参照されると、N_c から N_p に、この外部参照ポインタが指す値を要求する read メッセージが送られる。また、N_c 上で X_i を具体化した場合は、N_p にこの具体値との同一化を要求する unify メッセージを送る。

read の返事を待っている間や、unify が来る前に N_p で値を参照しようとした場合、それらのゴールは外部参照ポインタや X_i につながれ、他のゴールの実行に切り替わる。メッセージ到着によりこれらが具体化されると、このゴールの実行が再開される。この処理により、ノード間でも変数間の依存関係が保たれている。

3 本手法の概要

3.1 KLIC の問題点

KLIC では、実行コード生成を C コンパイラが行うので、低レベル部分の最適化に関しては、比較的効率の良いコードが得られる。しかし、以下に述べるような KL1 の実行モデルに依る速度低下は、解決できていない。

3.1.1 サスペンション

呼び出されたゴールの入力引数に未具体化のものがある場合、そのゴールはその引数の値を参照しようとした段階でサスペンションを起こし、他のゴールに切り変わる。

この機構によりデータ間の依存関係が自動的に保証されるため、プログラマはデータフローを定義すればよく、厳密なコントロールフローを設計する必要はない。これは、計算機内を複数のコントロールフローが走る並列プログラムの記述では、プログラマの負担を減らすことができ大きな利点となる。

しかし、このゴール切替が頻繁に生じた場合、そのオーバヘッドによる実行効率低下が無視できなくなる。

3.1.2 通信

KL1 では、外部参照ポインタが指す内容は必要になった時点で相手プロセッサに要求する。このため、1 個のデータを受けとるために、データ要求と返信、および外部参照ポインタの解放と、3 個の物理メッセージが必要となる。

また、現在の実装では、ゴール送信時に引数が構造データだった場合、実体を送らず外部参照ポインタを送信する。データ要求の場合、ネストした構造データはネストの最外側の値のみ返信する。

このため、複雑なデータ構造の全体を参照するような処理の場合、ネスト毎に改めて要求メッセージを送ることになり、メッセージ量が増大してしまう。データ構造全体を返信するような実装も可能であるが、この場合は、受信側で構造の一部しか参照しなかった場合、残りの部分に関して無駄な通信となってしまう。

3.2 最適化方針

3.1 節で述べた問題点を解決するため、本論では以下のような最適化手法を提案する。

3.2.1 並列プロセスの導入

KL1 ではプラグマにより並列実行するゴールを指定し、それ以外の部分は逐次実行される。したがって、プラグマで指定されたゴールの各々を開始点とし、そこから逐次実行される部分を、それぞれプロセスとみなすことができる。

プログラムの初期ゴールである `main/0`、およびすべての`@node`プラグマ付きのゴールを並列プロセス初期ゴールとすると、並列プロセス初期ゴール G_I から他の並列プロセス初期ゴールを経由せずに呼ばれるゴールの集合が、G_I の割り当てられたノード上で逐次実行される部分である。この集合を、並列プロセス G_I と呼ぶ。

3.2.2　サスペンションの削減

KL1 での変数間の依存関係は、手続き型言語のように命令の記述順に決まるのではなく、実行時に動的に決定していく。このため、文法上は依存関係を静的に決めることができない。

しかし、モード解析という手法を用いてプログラムを解析することにより、各ゴールにおける変数の入出力モードを決定し、データの依存関係をコンパイル時に決定することができる。この情報を利用し、ある変数を具体化するゴールを、それを参照するゴールの前にスケジューリングしてやれば、サスペンションをなくすことができる。

3.2.3　通信

ノード N_p上のプロセス P_pがノード N_c上にプロセス P_cを生成するとき、N_pから N_cにゴール送信が行われる。このとき、引数のデータのうち P_cが必要とする部分だけを送信するようにすれば、無駄な通信を生じることなく、P_cからのデータ要求メッセージを減らすことができる。また、ゴール送信時に生成されていない部分に関しては改めてデータ要求メッセージが送られるが、このときも生成されたデータの必要な部分をまとめて送るようにすれば、要求メッセージ量を削減できる。

KL1 では変数が型を持たないため、文法上は引数のデータ型は判明しない。しかし、タイプ解析を行うことにより、実行時にある変数がとり得るデータ型をコンパイル時に決定できる。この情報を利用し、前記のようなデータ送信を行うコードを生成する。

4　静的解析手法

前章で述べた最適化を行うため、プログラムのモード・タイプ解析を行う。

なお、本節での説明には図 1に示したスタックのプログラムを用いる。

モード　変数の出現毎に、その変数がそのゴールで入力、出力のどちらであるかを示す。

以下、$m(v_i) = \{in, out\}$ により、v_iのモードを表すとすると、$goal(v_1, \ldots, v_n)$ で $m(v_i) = in$ とは、$goal/n$ 内で v_iの値が参照されることを示す。また、$m(v_i) = out$ とは、$goal/n$ 内で v_iが具体化されることを示す。

ただし、述語 p/n について、ボディゴールとして現れる呼び出し側と定義節のヘッドとの間で、対応する引数のモードを一致させるため、クローズヘッドに関してはモードを逆に考える。

タイプ　その変数がどのようなデータ型に具体化され得るかを示す。

以下、$t(v_i) = \{atom, integer, \texttt{[]}, [t(car)|t(cdr)],$ ファンクタ名 $(t(arg_1), \ldots, t(arg_n))\}$ により、v_iのタイプを表す。複数のタイプをとり得る場合は、それらの集合で表す。

4.1　対象となるプログラム形式

解析の対象として、KLIC コンパイラがコンパイルの途中で生成する中間表現を利用する。この表現では算術演算式などのマクロが展開され、すべてのゴールは $goal($ $X_1, \ldots, X_n)$ の形で表されている。また、クローズヘッドの引数は、すべて変数である。さらに、`p(X) :- X=Y,Y=Z | q(Z)` における `Y` のような無意味な変数は、削除されているものとする。

節の一般形は、次のようになる。

$$g_H(V_H) : -G_G(V_H, V_G, V_B) \mid G_B(V_G, V_B)$$

各記号の意味は、以下の通りである。

g_H	クローズヘッド
G_G	ガードゴール集合
G_B	ボディゴール集合
V_H	ヘッド変数集合
V_G	ガード変数集合
V_B	ボディ変数集合

V_H, V_G, V_Bは、以下のように定義する。

1. v_iがクローズヘッドの引数なら、$v_i \in V_H$

2. `=`$(v_i$, $e(v_j^1, \ldots, v_j^n))$, $v_i \in V_H$ なら、$v_j^1, \ldots, v_j^n \in V_G$

3. 1,2 に該当しない場合、$v_i \in V_B$

ここで、$f(v_i^1, \ldots, v_i^n)$ は $v_i^1, \ldots, v_i^n$を含む構造データを表し、$e(v_i^1, \ldots, v_i^n)$ は $f(v_i^1, \ldots, v_i^n)$ または $v_i^1, \ldots, v_i^n$自身を表すものとする。

4.2　解析方法

4.2.1　節内解析

まず、各節毎に局所的な解析を行い、モードおよびタイプを決定する。

```
main :- drive(10,S),stack(S,none)@node(1).
drive(M,S) :- M=:=0 | S=[].
drive(M,S) :- M=\=0 |
  S=[push(M)|S0],S0=[pop(N)|S1],
  N1:=N-1,drive(N1,S1).
stack([]          ,D      ) :- terminate(D).
stack([push(X)|S],D      ) :- stack(S,p(X,D)).
stack([pop(X)|S] ,p(Y,D1)) :- X=Y,stack(S,D1).
terminate(D).
```

図 1: スタックプログラム

モード解析　節内の変数について、次の制約条件が成立する。

1. 同一化を除く組込述語の引数については、述語名と引数の位置で決定される。
   ただし、引数が $v_i \in V_H \cup V_G$ の場合、$m(v_i) = in$ である。
2. ガードゴール $g_i \in G_G$ について、
   (a) $=(v_i, atomic)$
   $\rightarrow m(v_i) = in$
   (b) $=(v_i, f(v_j^1, \ldots, v_j^n))$
   $\rightarrow m(v_i) = in$
   (c) $=(v_i, v_j)$, $v_i, v_j \in V_G \cup V_B$
   $\rightarrow m(v_i) = m(v_j) = in$
   (d) $=(v_i, v_j)$, $v_i \in V_H$, $v_j \in V_G$
   $\rightarrow m(v_i) = \overline{m(v_j)}$
3. ボディゴール $g_i \in G_B$ について、
   (a) $=(v_i, atomic)$, $v_i \in V_G \cup V_B$
   $\rightarrow m(v_i) = out$
   (b) $=(v_i, f(v_j^1, \ldots, v_j^n))$, $v_i \in V_G \cup V_B$
   $\rightarrow m(v_i) = out$
   (c) $=(v_i, v_j)$, $v_i, v_j \in V_G \cup V_B$
   $\rightarrow m(v_i) = \overline{m(v_j)}$
4. ある $v_i \in V_G \cup V_B$ の全出現のうち、一つだけが out で他はすべて in である。
5. $m(v_i) = in$ である出現がガード部にある場合、$m(v_i) = out$ である出現がガード部に存在する。

クローズヘッドの引数はガード部で具体化できないから、2a, 2b, 2c が成立する。また、ガード部での同一可能性テストを除き、同一化述語は最終的に一方の値で他方を具体化するから、2d, 3a, 3b, 3c が成立する。さらに、節内で複数のゴールが同じ変数を具体化できないから 4が、ガードで入力待ちのゴールを実行せずにボディゴールの実行はできないから 5 が、それぞれ成立する。

```
main
:- drive(10,V(0:b,_,_)),
   stack(V(0:b,_,_),none)@node(1).
drive(V(2:h,i,0),V(3:h,o,[]))
:- =(V(2:h,i,0),V(0:g,o,i)),
   =(V(3:h,o,i),V(1:g,i,[])),
   equal(V(0:g,i,i),0)
 | =(V(1:g,o,[]),[]).
drive(V(6:h,i,i),V(7:h,o,*1))
:- =(V(6:h,i,i),V(0:g,o,i)),
   =(V(7:h,o,*1),V(1:g,i,*1)),
   not_equal(V(0:g,i,i),0)
 | =(V(1:g,o,*1),
     [push(V(0:g,i,i))|V(2:b,i,*2)]),
   =(V(2:b,o,*2),
     [pop(V(3:b,o,i))|V(4:b,_,_)]),
   subtract(V(3:b,i,i),1,V(8:b,o,i)),
   =(V(5:b,o,i),V(8:b,i,i)),
   drive(V(5:b,i,i),V(4:b,_,_)).
stack(V(1:h,i,[]),V(2:h,_,_))
:- =(V(1:h,i,[]),[]),
   =(V(2:h,_,_),V(0:g,_,_))
 | terminate(V(0:g,_,_)).
stack(V(3:h,i,*3),V(4:h,_,_))
:- =(V(3:h,i,*3),
     [push(V(0:g,_,_))|V(1:g,_,_)]),
   =(V(4:h,_,_),V(2:g,_,_))
 | stack(V(1:g,_,_),p(V(0:g,_,_),V(2:g,_,_))).
stack(V(4:h,i,*4),V(5:h,i,*5))
:- =(V(4:h,i,*4),
    [pop(V(0:g,_,_))|V(1:g,_,_)]),
   =(V(5:h,i,*5),p(V(2:g,_,_),V(3:g,_,_)))
 | =(V(0:g,_,_),V(2:g,_,_)),
   stack(V(1:g,_,_),V(3:g,_,_)).
terminate(V(1:h,_,_))
:- =(V(1:h,_,_),V(0:g,_,_)).
```

タイプは頭文字で表記。
変数は `V(番号:種別, モード, タイプ)` で表記。
種別は `h,g,b` がそれぞれヘッド、ガード、ゴール変数。
また、$*n$ の部分は省略した。

図 2: 節内解析結果

タイプ　以下の制約条件が成立する。

1. 同一化を除く組込述語では、述語名と引数の位置により決定される。
2. 具体化では、その具体値の型になる。
3. 変数同士の同一化、あるいは変数を含む構造データとの同一化の場合、out モード引数のタイプは、in モード引数のタイプになる。

スタックプログラムの節内解析結果を、図 2に示す。“_” の部分はこの段階で解析できなかったが、これはこの節内では具体化も参照もされないことを表す。

4.2.2　並列プロセス分割

節内解析済みのプログラムを、並列プロセスに分割する。

並列プロセスは、各並列プロセス初期ゴールに次の手順を適用することで得られる。

1. ゴール集合 $G = \{$ プロセス初期ゴール $\}$,
   節集合 $C_P = \emptyset$ とする。
2. $G = \emptyset$ なら終了。
3. G からゴールを 1 個取り出し g とする。
4. ヘッドが g である節集合 C_g について $C_g \cap C_P = \emptyset$ なら、各 $c_i \in C_g$ について、
   (a) $c_i \rightarrow C_P$
   (b) c_i の各ボディゴール $\rightarrow B$
   ただし、`@node` プラグマ付きのものは無視する。
5. 2へ

4.2.3 プロセス内解析

節内解析の結果を元に、各プロセス内のモード・タイプを決定する。

述語 $p(v_1, \ldots, v_n)$ について、プロセス内のクローズヘッド p/n およびボディゴール p/n の集合をそれぞれ $H_{p/n}$, $B_{p/n}$、また、第 i 引数の集合をそれぞれ $H_{p/n,i}$, $B_{p/n,i}$ とすると、以下の制約条件が成立する。

モード $mode = \sum_{x_i \in H_{p/n,i} \cup B_{p/n,i}} m(x_i) - \{\_\}$ とすると、$\{\forall x_j | x_j \in H_{p/n,i} \cup B_{p/n,i}, m(x_j) = \_\}$ について $m(x_j) = mode$

タイプ $type = \sum_{x_i \in H_{p/n,i} \cup B_{p/n,i}} t(x_i) - \{\_\}$ とすると、$\{\forall x_j | x_j \in H_{p/n,i} \cup B_{p/n,i}, t(x_j) = \_\}$ について $m(x_j) = type$

ただし、タイプ中に $\forall x_i \in H_{p/n,i} \cup B_{p/n,i}$ を含む場合、その部分を記号+で置き換える。これは、再帰的なデータ構造を表す。

スタックプログラムのプロセス内解析結果を、図 3 に示す。

プロセス main/0

```
main
:- drive(10,V(0:b,o,*6)),
   stack(V(0:b,i,*6),none)@node(1).
drive(V(2:h,i,0),V(3:h,o,[]))
:- =(V(2:h,i,0),V(0:g,o,i)),
   =(V(3:h,o,i)V(1:g,i,[])),
   equal(V(0:g,i,i),0)
 | =(V(1:g,o,[]),[]).
drive(V(6:h,i,i),V(7:h,o,*1))
:- =(V(6:h,i,i),V(0:g,o,i)),
   =(V(7:h,o,*1),V(1:g,i,*1)),
   not_equal(V(0:g,i,i),0)
 | =(V(1:g,o,*1),
     [push(V(0:g,i,i))|V(2:b,i,*2)]),
   =(V(2:b,o,*2),
     [pop(V(3:b,o,i))|V(4:b,i,*6)]),
   subtract(V(3:b,i,i),1,V(8:b,o,i)),
   =(V(5:b,o,i),V(8:b,i,i)),
   drive(V(5:b,i,i),V(4:b,o,*6)).
        *1=[push(i))|[pop(V(i))|+]]
        *2=[pop(V(i))|+]]
        *6={[] , [push(V(i))|[pop(V(i))|+]]}
```

プロセス stack/2

```
stack(V(1:h,i,[]),V(2:h,i,*5))
:- =(V(1:h,i,[]),[]),
   =(V(2:h,i,*5),V(0:g,o,*5))
 | terminate(V(0:g,i,*5)).
stack(V(3:h,i,*3),V(4:h,i,*5))
:- =(V(3:h,i,*3),
     [push(V(0:g,_,_))|V(1:g,o,*7)]),
   =(V(4:h,i,*5),V(2:g,o,*5))
 | stack(V(1:g,i,*7),
         p(V(0:g,_,_),V(2:g,i,*5))).
stack(V(4:h,i,*4),V(5:h,i,*5))
:- =(V(4:h,i,*4),
     [pop(V(0:g,_,_))|V(1:g,o,*7)]),
   =(V(5:h,i,*5),
     p(V(2:g,_,_),V(3:g,o,*5)))
 | =(V(0:g,_,_),V(2:g,_,_)),
   stack(V(1:g,i,*7),V(3:g,i,*5)).
terminate(V(1:h,i,*5))
:- =(V(0:g,o,*5),V(1:h,i,*5)).
        *3=[push(_)|*7]
        *4=[pop(_)|*7]
        *5=p(_,+)
        *7={[] , [push(_)|+] , [pop(_)|+]}
```

図 3: プロセス内解析結果

4.3 データ依存解析

決定されたモードを利用し、プロセス内のデータ依存関係を解析する。

変数 v_i が依存する変数集合 $dep(v_i)$ は、以下のように決定できる。

1. $goal(v_1, \ldots, v_n) \in G_g \cup B_g$ に対して、$V_i = \sum_{m(v_j)=in} v_j, V_o = \sum_{m(v_j)=out} v_j$ とすると、$\forall v_k \in V_o$ に対し、$dep(v_k) \leftarrow V_i$
2. ガードゴールで in の変数集合を V_{gi}, $goal(v_1, \ldots, v_n) \in B_g, V_o = \sum_{m(v_j)=out} v_j$ とすると、$\forall v_k \in V_o$ に対し、$dep(v_k) \leftarrow V_{gi}$
3. 述語 $p(v_1, \ldots, v_n)$ に対して $v_i = in$ のとき、$dep(v_i \in H_{p/n,i}) \leftarrow B_{p/n,i}$, $v_i = out$ のとき、$dep(v_i \in B_{p/n,i}) \leftarrow H_{p/n,i}$
4. $v_i \in dep(v_j)$ のとき、$dep(v_j) \leftarrow dep(v_i)$

```
=(V(1:g,o,*1),[push(V(0:g,i,i))|V(2:b,i,*2)]),
=(V(2:b,o,*2),[pop(V(3:b,o,i))|V(4:b,i,*6)]),
                              ⇓
=(V(2:b,o,*2),[pop(V(3:b,o,i))|V(4:b,i,*6)]),
=(V(1:g,o,*1),[push(V(0:g,i,i))|V(2:b,i,*2)]),
```

図 4: プロセス内最適化

2の依存関係は、ガードが終了しないとボディが実行されないことによる。

5 最適化

前章で述べた静的解析による情報を利用し、次のような最適化を行う。

5.1 プロセス内最適化

現在の KLIC では、節内のガード／ボディゴール実行は元プログラムの記述順にスケジューリングされる。したがって、モード解析結果を元にゴールをデータフロー順にソートすれば、サスペンション回数を最小化できる。

また、構造データを具体化する際に内側から具体化していくことにより、変数の生成と同時に具体化することができ、いったん未具体化変数を生成してから改めて参照ポインタを付け直す処理が不要になる。

各節内の各ゴール g_n について、次のように半順序関係を定める。

1. v_i が g_i で in, g_j で out のとき、$g_i > g_j$
2. $g_i > g_j$ かつ $g_i < g_j$ のとき、$g_i = g_j$

この関係で昇順になるように、節内のゴールを並べ直す。

ここで、$g_i = g_j$ とは g_i と g_j が互いに相互依存関係にあることを表している。このような場合は並べ替えを行わない。

スタックプログラムの例では、図 4のように変更することにより、`V(2)` を生成と同時に具体化できる。

5.2 プロセス間最適化

ノード N_c 上のプロセス P_p/m がノード N_c 上にプロセス P_c/n を生成する場合を考える。

このとき行われるゴール送信では、以下のように引数を送る。

1. in モード引数
   引数データのうち、すでに生成されている部分について、P_c/n が参照するネストレベルまでを送信する。
2. out モード引数
   引数は必ず未具体化の変数であるので、この変数への外部参照ポインタを送信する。

5.2.1 in モード引数の送信

in モードであるような各引数について、P_c/n が参照するネストレベルとは、プロセス P_c/n の解析結果よりタイプの判明している部分までである。そこで、P_c/n の第 i 引数送信コードとして、以下のようなデータ送信コード $\texttt{reply\_}P_c\_n\_i(V)$ を生成する。

1. V が未具体化のとき、V への外部参照ポインタをバッファに格納。
2. アトミックデータの場合、値をバッファに格納。
3. タイプに出現するファンクタ $name(V_1, \ldots, V_l)$ の場合、$name/m$ をバッファに格納し、各引数 V_i について、
   (a) タイプ不明なら、実体は不用なので外部参照ポインタを格納。
   (b) タイプ判明なら、$\texttt{reply\_}P_c\_n\_i(V_i)$ を再帰呼び出し。
4. タイプに出現するリスト $[V_{car}\,|\,V_{cdr}]$ の場合、
   (a) $\texttt{reply\_}P_c\_n\_i(V_{car})$ を再帰呼び出し。
   (b) $\texttt{reply\_}P_c\_n\_i(V_{cdr})$ を再帰呼び出し。
5. その他の場合、V への外部参照ポインタを格納。

プロセス P_p/m はゴール送信時、このコードを呼び出して引数を送信バッファに格納する。

また、この時点で未具体化部分があると、P_c/n には外部参照ポインタ X が送られ、のちにその部分が必要になった時点で read メッセージにより改めて要求される。この場合は X を引数としてこのコードを呼び出すことで、その時点までに具体化された部分をバッファに格納し、返信メッセージを作成する。

5.2.2 単方向／遅延送信

5.2.1節で示した送信コードでは、ゴール送信時や read 返信時に構造データがほとんど具体化されていない場合も、外部参照ポインタの形で送信してしまう。

単方向送信　プロセス `stack/2` のように、リストを入力とするプロセスでリストの末尾到達がプロセス終了条件の場合、リストの各要素を必ず参照することが判明している。したがって、新たなリスト要素が具体化された時点で、read を待たずに P_c/n が送信を行ってよい。

これを実現するには、5.2.1節の 1で、具体化されたら再び送信を行うようなゴールを、V につないでおけばよい。

遅延送信　構造データの具体化部分が少ない場合は、ある程度具体化が進むまで送信を送らせることにより、一回の送信量を大きくすることができる。

これを実現するには、作成したメッセージのサイズをカウントし、ある閾値を超える前に 5.2.1節の 1 に到達した場合は送信を行わず、具体化されたらメッセージ作成を再開するゴールを V につないでおけばよい。

ただし、P_p/m と P_c/n の間でデータの相互依存関係があった場合、デッドロックを起こす可能性がある。4.3節で述べた $dep(v_i)$ に、P_p/m の並列プロセス初期ゴールまたは他の並列プロセス初期ゴール呼び出しの *in* モード引数が含まれていない場合は、v_i の具体化は P_p/m 内で完結でき、デッドロックの危険なしに上記の送信方法が可能になる。

5.2.3　*out* モード引数の送信

スタックプログラムの pop(V) のように *in* モード引数中に含まれるものも含め、引数に現れる *out* モード変数は、P_c/n で具体化されしだい、unify メッセージにより具体化要求が P_p/m に送られる。したがって現在の実装でも、効率的な単方向通信により、具体化された値が P_p/m に送信される。しかし、構造データの場合、P_p/m 側がどのネストレベルまで必要とするか不明なので、1 レベルだけしか送信されない。そこで、*out* モード変数については 5.2.1節で述べたようなコードを P_p/m について生成しておき、P_c/n でこれを用いて unify メッセージを作成する。

以上の方法により生成したプロセス `stack/2` のデータ送信コードを、図 5に示す。

6　おわりに

本論では、並列論理型言語 KL1 の静的解析手法と、それを利用した KL1 処理系 KLIC の最適化手法について述べた。

本手法ではプロセス単位の処理を行うことにより処理コストを小さくすると同時に、プロセス内の逐次実行効率化とプロセス間の通信量削減を実現している。

現在、KLIC コンパイラに、本手法による解析・最適化フェーズを実装中である。また、我々は AP1000 上に KLIC 処理系の実装を進めており、この上で本手法の性能評価を行う予定である。

```
reply_stack_2_1(V0)
:- unbound(V0)
 | put_buffer(variable,V0).
otherwise.
reply_stack_2_1([])
:- put_buffer(atom,[]),
reply_stack_2_1([V0|V1])
:- put_buffer(list),
   reply_stack_2_1(V0),
   reply_stack_2_1(V1).
reply_stack_2_1(push(V0))
:- put_buffer(functor,push,1),
   put_buffer(variable,V0).
reply_stack_2_1(pop(V0))
:- put_buffer(functor,pop,1),
   put_buffer(variable,V0).
otherwise.
reply_stack_2_1(V0)
:- put_buffer(variable,V0).

reply_stack_2_2(V0)
:- unbound(V0)
 | put_buffer(variable,V0).
otherwise.
reply_stack_2_2(p(V0,V1))
:- put_buffer(functor,p,2),
   put_buffer(variable,V0),
   reply_stack_2_1(V1).
otherwise.
reply_stack_2_1(V)
:- put_buffer(variable,V).
```

図 5: `stack/2` のデータ送信コード

謝辞

日頃より御討論いただく京都大学工学部情報工学教室 富田研究室 の諸氏に感謝致します。

参考文献

[1] Kazunori Ueda and Masao Morita. Message-oriented parallel implementation of moded flat GHC. In *Proc. Intl. Conf. on FGCS'92*, pp. 799–808, 1992.

[2] A. K. Bansal and L. Sterling. An abstract interpretation scheme for logic programs based on type expression. In *Proc. Intl. Conf on FGCS'88*, pp. 422–429, 1988.

[3] K. Ueda. Guarded horn clauses:a parallel logic programming language with the concept of a guard. Technical report, ICOT, 1986.

[4] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, 1990.

[5] Takashi Chikayama, Tetsuro Fujise, and Daigo Sekita. A portable and efficient implementation of KL1. In *Proc. 6th Intl. Symp. PLILP'94*, pp. 25–39, 1994.