静的解析による並列論理型言語の実行最適化

大野 和彦†，中島 浩†，富田 眞治†

†　:京都大学 工学部

並列論理型言語は非数値分野の問題記述に適しているが、動的なオーバヘッドが大きいため実行効率に問題がある。そこで本研究では、静的解析を用いた最適化手法を提案する。本手法はプロセスを処理単位とするため、効率的な解析・最適化が可能である。本論では、モード・タイプ解析によるコントロールフローおよびデータフロー情報の抽出手法と、これを利用したプロセス内の最適化による逐次実行の効率化、およびプロセス間入出力の最適化による物理的な通信の削減について述べる。

Optimization of Concurrent Logic Language with Static Analysis

Kazuhiko OHNO†, 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, nakasima, tomita}@kuis.kyoto-u.ac.jp

Although concurrent logic language is suitable for symbol processing, its execution is inefficient due to the dynamic overhead. We propose an optimization scheme using static analysis. Because each process is dealt independently, the analysis and optimization are efficient. In this paper, we describe derivation scheme of control-flow and data-flow information with mode-type analysis. Using this information, we also discuss optimization of inner-process execution and reduction of inter-process messages.

1 はじめに

現在研究が進みつつある並列計算機は、数値計算だけでなく、記号処理や人工知能といった非数値分野の利用も期待されている。しかし、今のところ研究の主流である手続き型言語の並列化コンパイラでは、後者の分野で求められる要件を満たしておらず、計算機の能力を十分に発揮できない。

並列論理型言語は論理的に並列な実行モデルを持つため、計算機の物理構成から独立したプログラムを自然に記述できる。また、暗黙の通信 ・同期によりアルゴリズム記述に専念できること、抽象度の高い記述により複雑なデータ構造が容易に扱えることなど、非数値分野の記述に適している。

しかしながら、現在実装されている並列論理型言語は、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、並列実行の単位となるプロセスはユーザが指定すると仮定し、プロセス単位の解析 ・最適化を行う。このため、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] によるコントロールフロー ・データフローの抽出を行う。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を行う。また、プロセッサ間通信になりうるプロセス間入出力を最適化し、物理的な通信の削減を図る。

以下、2 章では最適化の対象となる並列論理型言語 KL1 について説明し、3 章および 4 章で、それぞれ静的解析および最適化の手法について述べる。最後に 5 章で、まとめを行う。

2 並列論理型言語 KL1 の概要

本節では、今回提案する最適化手法の対象となる、並列論理型言語 KL1 の概要および特徴について説明する [3], [4]。

2.1 プログラム構造

KL1 は Flat GHC に基づいて ICOT で設計された言語であり、以下の形式で表される。

$$\begin{aligned} H_1 \ :- \ & G_{11}, \ldots, G_{1m_1} | B_{11}, \ldots, B_{1n_1} \\ & \vdots \\ H_l \ :- \ & G_{l1}, \ldots, G_{lm_l} | B_{l1}, \ldots, B_{ln_l} \end{aligned}$$

各行は節またはクローズと呼ばれる。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。

あるゴール H が与えられたとき、H と H_i との同一化、およびガード部 $G_{11}, \ldots, G_{1m}$ の実行を並列に行うことができる。ただし、H を具体化することはできず、ガードゴールに書けるのは組み込み述語のみである。続いて、ガード部が成功したクローズのうち一つが選択され、そのボディ部 $B_{l1}, \ldots, B_{ln}$ が実行される。ボディゴールには組み込み述語とユーザ定義述語を記述できる。

KL1 の述語は、述語名と引数の数で同定される。以下、stack/3 のように、“述語名/引数の個数” の形で述語を表記する。また、述語 p/n であるクローズヘッドの集合およびボディゴールの集合を、それぞれ $H_{p/n}$, $B_{p/n}$ で、それらの第 m 引数 $(1 \leq m \leq n)$ の集合を、それぞれ $H_{p/n,m}$, $B_{p/n,m}$ で表す。

2.2 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。セルは以下の型を持つ。

reference 他のセルへの参照ポインタである。KL1 の変数はこれで表現され、値に関して特定の型は持たない。変数は初期状態では未具体化であり値を持たず、同一化により対象へのポインタが格納される。具体化、すなわち具体値との同一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具体値を読み出すことができるようになる。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。

atom, integer 処理の最小単位となるシンボルおよび整数であり、いずれも 1 セルで直接値を格納する。両者を合わせて atomic 型と呼ぶ。

cons cdr, car の連続した 2 セルからなり、cdr は次の cons へのポインタを持つ。リスト構造 list 型は、この cons の連鎖で表現される。car にはリストの要素が格納されるが、これは atomic 型の他、別の cons や functor へのポインタの場合もある。

nil 空リストを表現するのに使用される。

functor 構造体であり、functor 名と引数の個数 n を格納したセルに、n 個の連続したセルが続く。これらには各引数が格納され、cons の car

と同様に atomic 型や cons ・functor へのポインタなどが格納される。

他に string, vector などのデータ型があるが、議論を簡略化するため、本論では扱わない。

2.3 組み込み述語

KL1 の組み込み述語のうち、本論に関係するものを説明する。

atom(X), integer(X), list(X), functor(X)
引数 X が、述語名の型であるかをチェックする。

X op Y 算術比較を行う述語であり、op は >,<,>=,=<,=:=,=\=のいずれかである。両辺には integer 型の引数をとり、算術式を記述できる。以下、算術比較述語と呼ぶ。

X := Y 算術式 Y を計算し、その結果で X を具体化する。以下、算術代入述語と呼ぶ。

X = Y X と Y を同一化する。以下、同一化述語と呼ぶ。

2.4 プロセスとストリーム

KL1 では、述語が自分自身を再帰的に呼び出すことによって、ある条件が満たされる間、存在し続けるようなプロセスを実現する。プロセスの内部状態は、呼び出し時の引数を現状態として次状態を生成し、これを再帰呼び出しの新しい引数とすることで保持している。

2.2 節で述べたように KL1 の変数は一度具体化されると値を変更できないため、ストリームと呼ばれるリストを用いてプロセス間の情報交換を行う。

ストリーム通信では、送信側が受信側と共有している変数を、car が送りたいメッセージ、cdr が新しい変数であるような cons セルで具体化する。cdr の変数に同様な具体化を繰り返すことで、次々にメッセージを送ることができる。

受信側は、car,cdr 共に未具体化の変数であるような cons セルと共有変数を同一化することによって、car 側変数の具体値としてメッセージを受信できる。また、cdr 側変数を次の共有変数として再帰呼び出しを行うことで、次々にメッセージを受けとることができる。

なお、本論では、プロセスとして動作する述語に加え、その述語から呼び出されてサブルーチン的に処理を行う述語も含め、全体を一つのプロセスプログラムと考える。

```
[1] stack([push(D)|I],O,S)
     :- stack(I,O,[D|S]).
[2] stack([pop|I],O,S)
     :- O=[A|NO], pop(A,S,NS),
        stack(I,NO,NS).
[3] stack([pop(N)|I],O,S)
     :- O=[L|NO], pop(N,L,S,NS),
        stack(I,NO,NS).
[4] stack([reverse(N)|I],O,S)
     :- O=[L|NO], rev:rev(R,L),
        pop(N,R,S,NS), stack(I,NO,NS).
[5] stack([],O,_) :- O=[].
[6] pop(A,[X|S],NS) :- A=answer(X), NS=S.
[7] pop(A,[],NS) :- A=empty, NS=[].
[8] pop(N,L,S,OS)
     :- N>0
     |  L=[X|NL], pop(X,S,NS),
        NN:=N-1, pop(NN,NL,NS,OS).
[9] pop(0,L,S,OS) :- L=[], OS=S.
```

図 1: スタックプロセス stack のプログラム

3 静的解析の流れ

KL1 プログラムの動作を解析するには、コントロールフローおよびデータフローの追跡が必要である。

今回提案する手法では、プロセス単位の局所的な解析を行う。現段階においては、プログラム中におけるプロセスの範囲、およびそのプロセスの初期ゴールは、ユーザから明示されるとする。

本論では具体例として、スタックとして機能するプロセス stack のプログラム (図 1) を取り上げる。なお、説明のため行頭に番号を付加している。

このプロセスは、第一引数のストリームにスタック操作コマンドを要素とするリストを受けとり、第二引数のストリームにその結果を出力する。スタックは第三引数にリストとして保持している。スタック要素の型は任意である。

入力可能なコマンドには、以下のものがある。

push(D) 値 D をスタックトップに入れる。

pop スタックトップの値を取り出して出力する。

pop(N) スタックトップから N 個の値を取り出し、それらを長さ N のリストとして出力する。

reverse(N) 同様に N 個取り出し、逆向きのリストにして出力する。

具体的な操作はプロセス内の他の述語をサブルーチン的に呼び出して行うが、reverse(N) だけはプロセス外の述語 rev/2 を呼び出す。

3.1 コントロールフロー解析

初期ゴールが与えられた時、プログラムに次の手順を適用して実行制御の流れを追跡し、ゴール間の呼びだし関係木を作成する。以下、これをコントロールフローグラフ (CFG) と呼ぶ。

- 引数の内容は無視し、述語名および引数の個数のみで同一化を行う。
- クローズヘッドが呼び出されると、その節のボディゴールはすべて呼び出される。したがってクローズヘッドは、ボディゴールを子とする AND 節点で表す。
- 実行時においては、ボディゴール p/n は $H_{p/n}$ のいずれかを呼び出すが、ここでは呼び出される節を決定できない。したがってボディゴールは、$H_{p/n}$ を子とする OR 節点で表し、各クローズヘッドについて、それぞれ処理を続行する。
- ボディゴールが組み込み述語の場合は、同一化処理を行わず、ボディゴールは葉で表される。
- ボディゴールが p/n であるとき、すでに CFG 中に $H_{p/n}$ が存在していれば、そのボディゴールの子は新たに生成せずに、$H_{p/n}$ へのリンクとなる。したがって、再帰呼び出しの場合はループを生じることになる。

以上より、CFG は AND-OR グラフにループ・合流が含まれた形になる。

初期ゴール `stack(L,R,[])` として図 1のプログラムより得た CFG を、図 2に示す。

3.2 モード・タイプ解析

ここでは、モードと呼ばれる変数の入出力方向、および、タイプと呼ばれる変数の値の型を解析する。

KL1 では、変数はすべて述語の引数として現れ、その名前は節毎にローカルである。ボディゴールが節を呼び出す際に引数間で同一化が行われることにより、異なる節の変数が同一視され、いずれかが具体化された時点で、他の節でもその値を読めるようになる。また、一度具体化された値は (同一化されている全変数について) 変更できないという特徴がある。

したがって、CFG 上で変数の具体化や値の参照を行っているゴールを見つけ、他の変数との同一化をたどることにより、各述語でのモード・タイプを解析することができる。

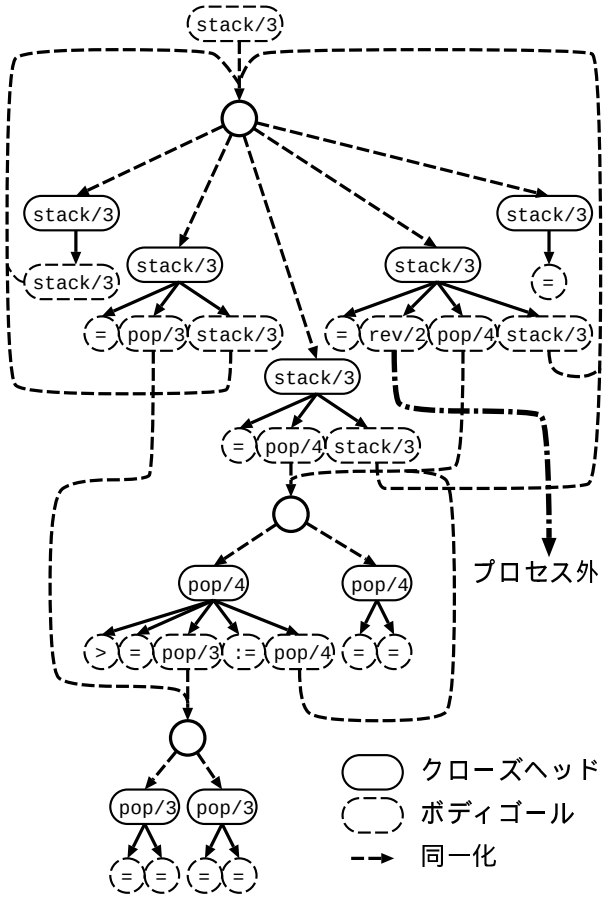

図 2: コントロールフローグラフ (CFG)

モード *mode* は、次のように表現される。

in そのゴールより下で値の参照を行うため、親節点のゴールより具体化された値を受けとる必要がある。

out そのゴールより下で具体化が生じており、その値を親節点へと送る。

unknown 解析の結果、その引数のモードは決定できなかった。

don't care その引数は無視される。このタイプの引数については、以後の解析で無視する。

mode1|*mode2*|... モードは *mode1*, *mode2*,... のいずれかを取りうる。

また、タイプ *type* は、次のように表現される。

atom, int, nil それぞれ対応する具体値の型。

cons(*type1*,*type2*) *type1* および *type2* を、それぞれ car および cdr として持つ cons セル。リストは cons セルの連鎖で表されるから、この型になる。

表 1: 組み込み述語のモード ・タイプ

組み込み述語	モード	タイプ
型チェック *type*(X)	X:in	*type*
算術比較 X *op* Y	X,Y:in	X,Y:integer
算術代入 X := Y	X:out,Y:in	X,Y:integer
算術式	すべて in	すべて integer
具体化 X=具体値	X:out	X:具体値の型
同一化 X=Y	X:in,Y:out	X,Y は同じ型
	または	
	X:out,Y:in	X,Y は同じ型

func(*name*/*arity*, *type1*, *type2*, ..) 名前 *name*, 要素数 *arity*, 要素の型 *type1*, *type2*, ... であるような functor。

unknown 解析の結果、その引数の型は決定できなかった。

don't care その引数は無視される。このタイプの引数については、以後の解析で無視する。

type1|*type2*|... モードは *type1*, *type2*,... のいずれかを取りうる。

モードタイプ解析のアルゴリズムについて、以下に述べる。

3.2.1 節内解析

まず、各節毎に局所的な解析を行う。

1. 組み込み述語の引数について、表 1のように決定する。
2. 引数が具体値なら、モードは in、タイプはその具体値の型になる。ただし具体値が cons や func の場合、中身に含まれる変数については不定である。
3. ある節内の変数X について、
   (a) あるボディゴールの out モード引数なら、その節のクローズヘッドおよび他のボディゴールの引数については、それぞれ out および in モードになる。
   (b) クローズヘッドの in モード引数なら、その節のすべてのボディゴールについて in モードになる。
   (c) タイプはすべて同じ型になる。

stack プログラムにこのアルゴリズムを適用した結果を、図 3に示す。

```
[1] stack(i,u,u) :- stack(u,u,i).
  D :u  I :u  O :u  S :u
[2] stack(i,u,u)
     :- u=u, pop(u,u,u), stack(u,u,u).
  I :u  O :c({A},{NO})  S :u  A :u  NO:u  NS:u
[3] stack(i,u,u)
     :- u=u, pop(u,u,u,u), stack(u,u,u).
  N :u  I :u  O :c({L},{NO})  S :u
  L :u  NO:u  NS:u
[4] stack(i,u,u)
     :- u=u, rev:rev(u,u),
        pop(u,u,u,u), stack(u,u,u).
  N :u  I :u  O :c({L},{NO})  S :u
  L :u  R :u  NO:u            NS:u
[5] stack(i,o,d) :- o=i.
  O :n
[6] pop(o,i,u) :- o=i, u=u.
  A :f(answer/1,{X})  X :u  S :u  NS:u
[7] pop(o,i,o) :- o=i, o=i.
  A :a  NS:n
[8] pop(i,u,u,u) :- i>i
     |  u=u, pop(u,u,u), o:=i, pop(i,u,u,u).
  N :i  L :c({X},{NL})  S :u  OS:u
  NL:u  NS:u            NN:i
[9] pop(i,o,u,u) :- o=i, u=u.
  L :n  S :u  OS:u
```

※ モード ・タイプ名は頭文字で表記
※ {X}は、変数X のタイプを表す

図 3: 節内解析結果

3.2.2 節間解析

同一化したボディゴールとクローズヘッドでは、モードおよびタイプは同じになる。しかし、静的解析において同一化対象は一意に決定できないから、同一化の可能性がある相手のモード ・タイプの OR 表現で表す。

$B_{p/n}$中の任意のボディは$H_{p/n}$中の任意のクローズヘッドを呼びだし可能であり、このとき対応する引数が同一化する。CFG の節点を述語から任意の引数に置き換えることにより、この引数の同一化グラフが得られる。ただし、節内解析でモードが判明している引数からは有向枝、モードが不明な引数からは無向枝を出す。

この同一化グラフを利用して、以下のようにモード ・タイプを決定する。

1. $H_{p/n,m}$, $B_{p/n,m}$の判明しているモード ・タイプについて、それぞれ和集合をとる。これを $HM_{p/n,m}$, $HT_{p/n,m}$, $BM_{p/n,m}$, $BT_{p/n,m}$と、それぞれ表記する。

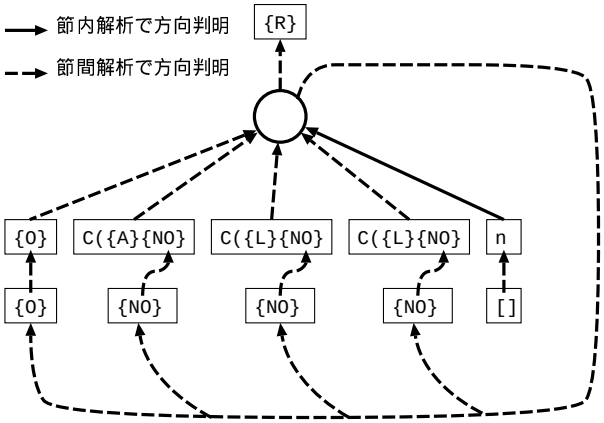

図 4: 同一化グラフ

```
[4] stack(i,o,i)
     :- o=i, rev:rev(i,o),
        pop(i,o,i,o), stack(i,o,i).
  N :i
  I :c(f(push/1,{D})|a|f(pop/1,i)
     |f(reverse/1,i),{I})|n
  O :c({L}{O})
  S :c({D},{S})|n
  L :u
  R :c(f(answer/1,{D})|a,{R})|n
  NO:c(f(answer/1,{X})|a|{L}{NO})|n
  NS:{S}
```

図 5: stack の節 [4] 最終解析結果

2. $BM_{p/n,m}$に unknown が含まれていなければ、クローズヘッド $h \in H_{p/n,m}$のうちモードが不明なものは、$BM_{p/n,m}$になる。
   ただしこの場合、h のうちモードが決定しているものは $BM_{p/n,m}$のどれかと同一化可能でなければならない。これにより、cons や func の要素のモードが決められる場合がある。
   $HM_{p/n,m}$に unknown が含まれていない場合も、同様にして $b \in B_{p/n,m}$のモードを決定する。

3. $BM_{p/n,m}$および $HM_{p/n,m}$が共に unknown を含む時は、h,b のうちモードが不明なものは、$BM_{p/n,m}$と $HM_{p/n,m}$の和集合になる。

4. タイプについても、2,3 と同様に決定する。

図 3より得た述語 stack/3 の第二引数の同一化グラフを、図 4に示す。

節間解析の結果、新たにモード ・タイプが判明したものに対しては、それを含む節について節内解析から繰り返す。

stack プログラムの節 [4] について、最終的に得られた各変数のモード ・タイプを図 5に示す。

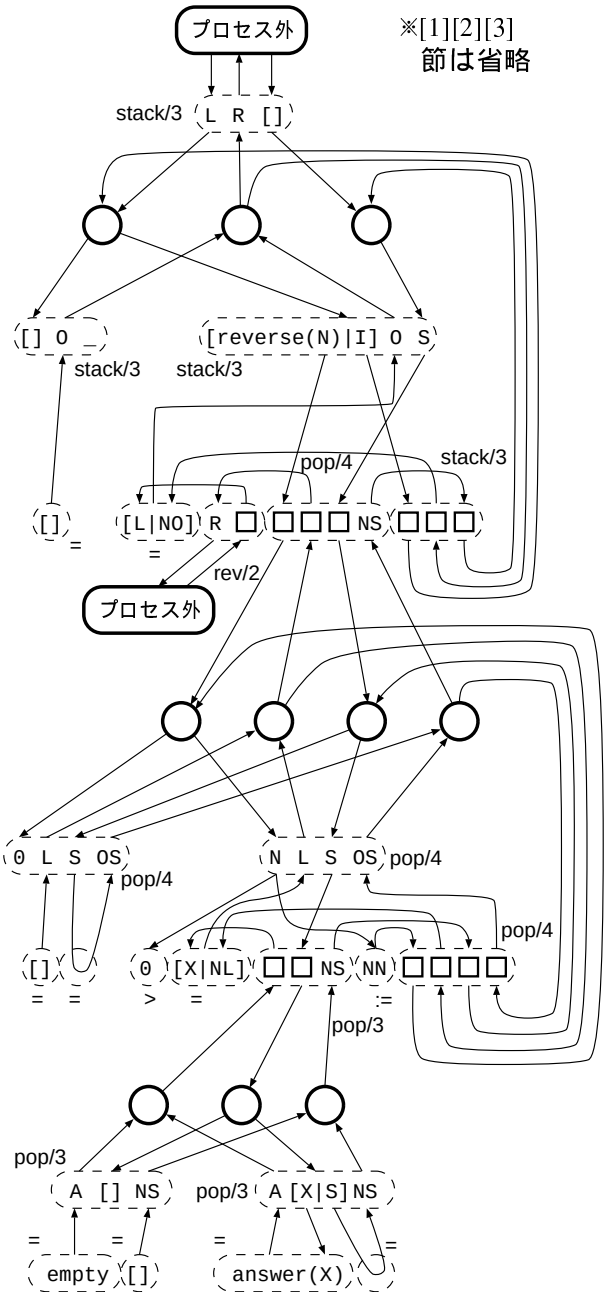

図 6: データフローグラフ (DFG)

3.3 データフローグラフ

タイプ ・モード解析の結果より、変数間の依存関係が有向グラフとして表せる。以後、これをデータフローグラフ (DFG) と呼ぶ。

KL1 の実行において、変数同士の同一化などは両者が未具体化でも可能であり、サスペンシ ョン

を引き起こすのは、未具体化変数の値を参照しようとしたときである。これを防ぐには、参照される引数の値を事前に定義 (具体化) しておかなければならない。したがって、データフロー中で定義・参照が行われる箇所を明らかにする必要がある。

値の参照は、クローズヘッドおよびガードゴールでの具体値との同一化、算術比較述語の両辺、算術代入述語の右辺で生じる。また、値の定義は、ボディでの具体化、算術代入述語の左辺で生じる。

プログラム stack より得た DFG を、図 6に示す。

4 最適化

前章で述べた静的解析から得られる情報を用いて、以下に述べるような最適化が可能になる。

4.1 サスペンション回数の削減

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す。この結果、ゴールの切替のオーバヘッドが生じ、実行効率が低下する。

3.3節で述べた DFG を利用し、データの生成順にゴールを実行すれば、このサスペンション回数を削減することができる。現在、多くの KL1 処理系では、節内のゴールを左優先にスケジューリングしている。したがって、プログラム中における述語の順序を書き換え、定義が参照の前に行われるようにする。

stack プログラムにこの最適化を施した結果を、図 7に示す。

```
[1] stack([push(D)|I],O,S)
     :- stack(I,O,[D|S]).
[2] stack([pop|I],O,S)
     :- pop(A,S,NS), O=[A|NO],
        stack(I,NO,NS).
[3] stack([pop(N)|I],O,S)
     :- pop(N,L,S,NS), O=[L|NO],
        stack(I,NO,NS).
[4] stack([reverse(N)|I],O,S)
     :- pop(N,R,S,NS), rev:rev(R,L),
        O=[L|NO], stack(I,NO,NS).
[5] stack([],O,_) :- O=[].
[6] pop(A,[X|S],NS) :- A=answer(X), NS=S.
[7] pop(A,[],NS) :- A=empty, NS=[].
[8] pop(N,L,S,OS)
     :- N>0
     |  pop(X,S,NS), NN:=N-1,
        pop(NN,NL,NS,OS), L=[X|NL].
[9] pop(0,L,S,OS) :- L=[], OS=S.
```

図 7: サスペンション回数の削減

4.2 構造体送信レベルの最適化

cons や func が複雑にネストした構造体を他のプロセッサに送信する時、構造体の実体をどのレベルまで同時に送るかが問題になる [5]。送信した実体に受信側が参照しない部分があれば余分な通信を行ったことになるが、送信しなかったレベルまで受信側が参照すれば、その時点で新たに送信が必要になる。

静的解析の結果、プロセス内で構造体の実体をどのレベルまで参照するかが判明していれば、この情報を送信側のプロセスが利用することにより、必要十分なデータだけを送信することが可能になる。

stack の例では、前章の静的解析により、stack/3 の第一引数 L は functor または atom を要素とするリストであり、その末尾まで要素を参照することが判明している。したがって、リスト構造体はすべての cons セルを送信する必要があり、car が functor へのポインタの時、functor の名前も送信する必要がある。functor の引数については、push(D) の場合、D の中身は参照されていないから送信する必要はない。一方、pop(N) の場合、N は integer として値を参照しているから、必ず送信しなければならない。

このような、タイプに対する参照レベルの情報を出力することにより、対象のタイプにより送信レベルを選択するようなコードを送信側プロセスに付加し、通信を最適化することができる。

4.3 プロセス内消費型データの一括送信

プロセス間通信のオーバヘッドを削減する手段として、ストリームの要素を一つずつ送信せず、ある程度まとめて送る方法がある。しかし、プロセス単位の解析ではプロセス外に対するストリーム通信の相手は不明であり、プロセス外においてこれらのストリーム間に依存関係が存在しうる。したがって、現在のストリーム要素を出力しなければ、次のストリーム要素生成に必要な外部からの入力データが得られない可能性があり、この場合、一括送信はデッドロックを引き起こす。

入力ストリームの要素のうち、プロセス内で値を参照するだけで、これに依存するプロセス外へ

の出力を行わないものは、送信側プロセスが送信を遅らせてもデッドロックを生じる危険がない。このようなデータをプロセス内消費型と呼ぶ。

静的解析の結果、プロセス内消費型のタイプが判明したとする。この場合、該当タイプが連続する間はストリームの送信を抑制し、ある程度データが溜るかプロセス内消費型でないデータがストリームに出力された段階で一括送信するようなコードを、送信側プロセスに付加する。

stack の例では、stack/3 の第一引数 L は要素が push(D) のとき、プロセス外への出力は行わない。したがって、push(D) が連続する限りにおいて、送信側はこのストリームの送信を抑制し、通信回数を削減することができる。

4.4 プロセス内生産型データの一括送信

ストリーム通信を行う場合、リストの要素のうち受信側が参照するネストレベルまでを完全に具体化してからストリームとの同一化を行い、物理的な送信を行った方が効率が良い。しかし、具体化に時間を要する場合は、具体化できたところまでを先に送った方がパイプライン効果が期待できる。

出力ストリームの要素の具体化方法は、一要素に必要とする時間により、以下のように分類できる。

1. 定数時間で具体化できる。
2. 所要時間は実行時に決まるが、プロセス内の情報のみで具体化を完了できる。
3. 具体化過程においてプロセス外の述語を呼び出しており、所要時間はプロセス内解析のみでは不明である。

stack の例では、pop が入力されたとき出力ストリームの要素は定数回の述語呼び出しで具体化できるから、1に当たる。pop(N) が入力されたときは pop/4 の再帰呼び出しが N 回生じるから、2である。reverse(N) が入力されたときはプロセス外の述語 rev/2 を呼び出すから、3に相当する。

1の場合、明らかに具体化完了までストリームとの同一化を遅らせた方がよい。また、3の場合、デッドロックを生じる可能性もあるから、プロセス内で具体化された分を一旦送信する必要がある。

2の場合、デッドロックの危険はないが、所要時間が大きい場合にはパイプライン効果が失われてしまう危険がある。また、4.3節で述べた自己消費型データの出力抑制においても、同様な問題が生じる。

これらの問題の解決策として、ストリームが具体化された段階では物理的な送信を行わずにリストの要素や各々の中身といった具体値のセル数を数え、これが一定値を超えたら物理通信を行う方法が考えられる。非自己消費型や3に相当する要素が現れたり送信側がストリームを閉じたりした場合には、残りの要素を直ちに送信する。

5 おわりに

本論では、並列論理型言語 KL1 を対象とした静的解析法と、それを利用した最適化の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、解析・最適化のコストを小さくすることができる。また、プロセス内は効率的な逐次実行を行い、プロセス間は通信量を削減することにより、効果的な最適化が可能である。

現在、本手法により自動解析・最適化を行うツールを実装中であり、今後の課題は本手法による最適化の効果を実際の処理系上で評価することである。また、プロセス毎の解析結果からプログラム全体の構造を解析し、大域的な最適化を行う手法も検討している。

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] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, Vol. 5, pp. 207–229, 1988.

[2] A. K. Bansal and L. Sterling. An abstract interpretation scheme for logic programs based on type expression. In *Proc. Int. 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] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, pp. 436–451, 1989.