静的解析による並列論理型言語KL1の
通信最適化手法

大野 和彦

平成7年2月15日

Abstract

第五世代計算機計画の核言語として開発された並列論理型言語KL1は、通信·同期が暗黙であることや動的なデータ構造を容易に扱えることなどの利点を持ち、記号処理や人工知能といった非数値分野における問題記述に適している。

KL1の処理系は、従来の研究においていくつかの実装が行われている。最近ではKLICと呼ばれる移植性に優れた処理系が開発され、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されている。しかしこれらの実装は実行効率の面で手続き型言語に及ばず、実行方式の改良が必要とされている。

とくにメッセージ交換型並列計算機を対象とした場合、頻発する細粒度通信のオーバヘッドが実行速度低下の大きな要因になっている。これは、動的なデータ構造を扱うためデータ構造中の通信部分が静的に決定できず、データが必要になった時点で保持者に対し要求メッセージを送信し、これに対してデータを返信するという形で通信が行われるためである。したがって、大きな構造データでは要素の参照毎に通信が行われてしまう。

そこで本研究では、粒度の高い効率的な通信を実現する一括送信手法を提案する。本手法は、静的解析情報を利用して受信側が参照するデータのみを一度に送ることにより、不要なデータを送信することなく通信回数を削減する。

本手法は、対象プログラムの静的解析、一括送信コードの生成、およびKL1ランタイムの拡張からなる。

静的解析では、各ノードに配置される並列プロセス毎にモード·タイプ解析を行う。この解析では、各変数が実行時にとり得るデータ型や具体化·参照を行う箇所が明らかになる。この結果、同一化により他の並列プロセスと共有される変数vについて、そのプロセスが参照·具体化のどちらを行うか、また実行時にとり得るvのデータ型のうち、どの部分を参照するかが判明する。

一括送信コードの生成では、静的解析情報から、vのうちそのプロセスが参照する要素のみを一度に送信するコードを生成する。このコードはKL1プログラムの形で生成されるため、処理中のデータ操作にKL1ランタイムのメモリ管理機能などを利用できる。

拡張したKL1ランタイムでは、vの具体化を行うプロセスにおいて、生成した一括送信コードをKL1ゴールとして実行する。このため、KL1ランタイムのカーネル部を変更することなく一括送信を実装できる。

我々が移植したAP1000版KLIC上を対象として、本手法を実装し性能評価を行った。評価に用いたプログラムはノード間でスタック操作を行う`stack`、数字の並びを推測するゲームである`mastermind`、およびN-queen問題を解く`nqueen`の3種類である。

この結果、本手法による一括送信コード生成では、スタック操作命令列やゲーム盤の状況を示したリストなどノード間送信が行われる構造データに関して、適切なデータのみを送信するコードが得られた。

また、一括送信を実装したランタイム上で実行し、従来の実装と比較を行った。この結果、通信が頻発する`stack`および`mastermind`では約2倍の速度向上が得られた。また、演算時間に比べ通信量の少ない`nqueen`についても、最大8%の速度向上が得られた。一方、物理通信回数については、いずれも1/4以下に減少するという良好な結果が得られた。

Efficient Message Communication of Concurrent Logic Programming Language KL1 Based on Static Analysis

Kazuhiko OHNO

Feb. 15, 1995

Abstract

The concurrent programming language KL1 was designed as a kernel language of the Fifth Generation Computer Systems project. This language has some advantages that communications and synchronizations are implicit, and handling dynamic data structure is simple. As a result, it is appropriate for nonnumerical processing.

Several implementations of KL1 has been made in previous researches. KLIC, one of such implementations, has high portability, which is promising for the use on several parallel computers and workstations. However, the executing efficiency of these implementations are inferior to the parallel programming languages based on procedural language, thus improvement of the execution method is required.

On message-passing multiprocessors, frequent occurrence of fine grained communication is a major factor of slow execution. This is because of the demand-driven communication due to the handling of dynamic data structure. Therefore, messages are required for each element of the structure to send a large data structure.

In this paper, we propose a packing-send method, which makes the grain of message larger and results in efficient communications. By using information of static analysis, this method packs data which receiver process refers to, and reduces number of communications without sending unnecessary data.

In the first step, each parallel process is analyzed by mode-type analysis, which extracts possible data types and input/output directions of variables. With the variable v which is shared with other parallel processes, direction of communications and required elements of the possible data structures are identified.

In the second step, packing-send program code, which creates a message consisted of required elements, is generated. Because this code is a KL1 program, the memory management function of KL1 runtime routine is available in the data manipulation of packing-send.

On the modified KL1 runtime routine, the generated code is executed as KL1 goals in the process which creates data structure of v. Therefore, the modification of the KL1 runtime routine kernel is not necessary in implementation of packing-send method.

We made implementation and evaluation of this method on KLIC, which we ported on AP1000. The programs used for evaluation are `stack` which makes inter-node manipulation of stack, `mastermind` which guesses a sequence of numbers, and `nqueen` which solves the N-queen problem.

By applying our method to these programs, we obtained proper packing-send program code. We also executed these programs on modified KLIC runtime routine and compared the result with that of unmodified runtime routine. As a result, we obtained 2 times speedup with `stack` and `mastermind`, on which frequent communications occur. We obtained a maximum of 8% speedup with `nqueen`, on which operation time dominate execution time. The number of communications is reduced to 1/4 or less of its initial value.

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。

これに対し、第五世代計算機計画の核言語として開発された並列論理型言語 KL1 は、論理的に並列な実行モデルと暗黙の通信 ·同期により計算機の物理構成から独立したアルゴリズムを自然に記述できる、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、といった特徴を持ち、非数値分野のプログラム記述に適している。

同計画で開発された並列推論マシン PIM 上には KL1 処理系が実装され、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。また、最近では移植性の高い処理系 KLIC が開発され、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。こうしたことから将来的には多くの並列計算機環境上で、これらのソフトウェアの利用や KL1 の記述能力を生かした並列プログラム開発が期待できる。

しかしながら、KLIC など従来の実装では、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。とくにメッセージ交換型並列計算機を対象とした場合、頻発する細粒度通信のオーバヘッドが速度低下の大きな要因になる。これは、動的なデータ構造を扱うため、データ中の通信対象部分が実行時に決定し、必要になると要求 ·返信という形で通信が行われるためである。

そこで本研究では、通信対象となるデータ型や通信方向を静的解析し、この解析情報を利用して粒度の高い効率的な通信を行う一括送信手法を提案する。また、KLIC 処理系は本研究室において並列計算機 AP1000 上に移植されており、これを用いた本手法の性能評価についても述べる。

本手法では、まずモード ·タイプ解析 [1], [2], [3] により、実行時に生じるデータの型や依存関係をコンパイル時に抽出する。続いてこの情報を利用し、必要なデータのみを一括送信するコードを生成する。実行時には拡張した KLIC ランタイムがこのコードを利用することにより、物理通信の粒度を上げ、通信回数の削減を図る。

以下、まず 2 章で KL1 と KLIC について説明する。続いて 3 章および 4 章で、今回提案する手法およびその性能評価について、5 章で本手法の問題点と解決方法について、それぞれ述べる。最後に 6 章で、まとめを行う。

2　背景

本章では、並列論理型言語 KL1 [4], [5] およびその処理系 KLIC [6], [7] について簡単に説明する。

2.1　並列論理型言語 KL1

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処理をリダクションと呼ぶ。このリダクション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 $G_{i_1}, \ldots, G_{i_m}$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 $B_{j_1}, \ldots, B_{j_n}$が並列実行される。

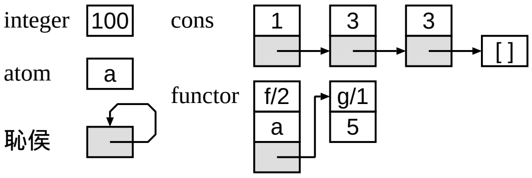


図 1: KL1 のデータ構造

ボディゴール B_{j_k}で、ある引数 vを参照しようとした場合、vがまだ値を持たない未具体化変数なら B_{j_k}は実行を中断し、vが具体化されて値を持つと実行を再開する。これをゴールのサスペンションと呼び、この機構により自動的にゴール間の依存関係が保たれるようになっている。

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。セルは以下のような型を持つ (図 1)。

参照ポインタ (reference) 1 セルに他のセルへのポインタが格納される。変数はこれで表現され、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、あるいは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。

アトミック (atomic) 処理の最小単位となる型であり、1 セルに値が直接格納される。アトム (atom) および整数 (integer) はこの型で表現される。

構造データ (structure) 複数のセルからなり、次のようなものがある。

コンス (cons) car, cdr の連続した 2 セルからなる。プログラムでは [car|cdr] と表記される。car および cdr は任意型をとることができ、リストなどの動的データ構造を表現するのに用いられる。

たとえば線形リストは、car セルにリスト要素の具体値 (へのポインタ)、cdr セルに他の cons セルへのポインタまたは空リストを表すアトム [] を格納することで表す。

ファンクタ (functor) ファンクタ名と引数の個数 n を格納したセルに n 個の引数格納用セルが続き、後者には具体値 (へのポインタ) が格納される。

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。

2.1.3　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。

2.1.2 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。

例えば図 2に示したプログラムは、プロセス間でスタック操作を行うプログラムである。実

```
1 main
 :- drive(10000,S),stack(S,none)@node(1).
2 drive(M,S)
 :- M=:=0 | S=[].
3 drive(M,S)
 :- M=\=0 | S=[push(M)|S0],S0=[pop(N)|S1],
           N1:=N-1,drive(N1,S1).
4 stack([]          ,D       )
 :- terminate(D).
5 stack([push(X)|S],D       )
 :- stack(S,p(X,D)).
6 stack([pop(X)|S]  ,p(Y,D1))
 :- X=Y,stack(S,D1).
7 terminate(D).
```

※行先頭の節番号は、説明のために付加。

図 2: プログラム stack

行によりスタックプロセス stack と操作プロセス drive が生成され、前者の第 1 引数と後者の第 2 引数が共有変数となる。drive は共有変数 S を [push(M),pop(N),S1] で具体化し、S1 を新たな第 2 引数として自己再帰を行うことで、次々にリスト要素を生成する。一方 stack は第 1 引数の car 部分を参照し、cdr 部分を新たな第 1 引数として自己再帰を行うことで、リスト要素を順に参照する。

この動作により、drive は操作命令 push(N) および pop(N) をストリームの要素として stack に送り、stack は push(N) に対しては N の値を構造データの形で順に格納し、pop(N) に対しては N をスタックトップの値で具体化する。

2.2 KL1 処理系

従来の研究において、いくつかの KL1 処理系が実装されているが、基本的な方針は似通っている。そこで本節では、KLIC を例に説明する。

KLIC は、(財) 新世代コンピュータ技術開発機構 (ICOT) で開発された KL1 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。

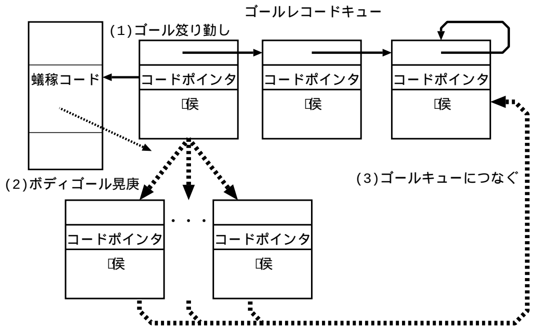


図 3: ゴールの実行

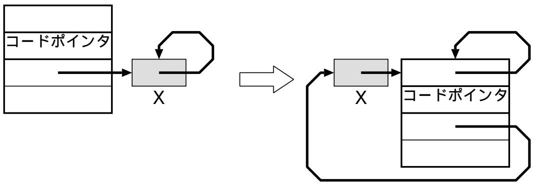


図 4: 変数へのゴールフック

2.2.1 逐次実行方式

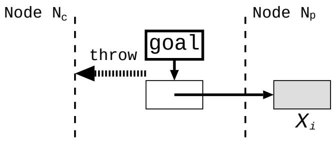
ある節が選択された時の変数操作やガード / ボディゴール呼び出しなどは、直接あるいは KLIC ランタイムルーチンの呼び出しによりこれらの処理を行うような、C 言語プログラムに変換される。

実行時のゴールの実体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。このゴールレコードや実行時に生成されるデータは、ヒープ上に格納され、コピー型ガーベジコレクションを用いた動的なメモリ管理が行われる。

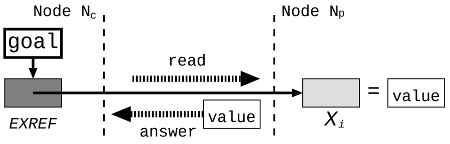
プログラムの実行は、実行キューにつながれたゴールレコードを取り出して対応するコードを実行し、その結果得られたボディゴールのレコードをキューにつなぐことで行われる (図 3)。

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、2.1.1 節で述べたようにゴールのサスペンションが生じる。具体的には、X が指示するキューにそのゴールレコードがつながれ (図 4)、実行キューから別のゴールが取り出されて実行される。これを、ゴールが変数 X にフックすると表現する。

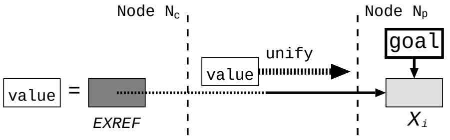
のちに X が具体化されると、つながれていたゴールは再び実行キューに戻される。この機構



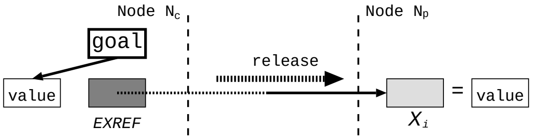
(a) ゴール索軒



(b) read/answerメッセージ



(c) unifyメッセージ



(d) releaseメッセージ

図 5: ノード間通信

により、ゴールの実行スケジューリングを動的に行いながら、変数の定義・参照順序が保証されるようになっている。

2.2.2 並列実行方式

KLIC を含めた KL1 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、

$$goal(X_1, \ldots, X_n)\texttt{@node}(N_c)$$

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 throw メッセージによりノード 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 X_i を指す外部参照ポインタが N_c に送られる。(図 5a)。

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。N_p は answer メッセージにより、要求された値を送信する (図 5b)。一方、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 (図 5c)。

前者の場合は answer メッセージ受信後、その外部参照ポインタによる参照がなくなったことを知らせる release メッセージが N_c から N_p に送信される (図 5d)。これは、read メッセージ送信後から answer メッセージ受信までの間に N_c において X_i の具体化が起こる可能性があるため、N_c が answer メッセージを受信したことを確認するまで N_p 上の X_i をガーベジと判定してはならないためである。後者の場合は unify メッセージ到着により N_c からの参照がなくなったと判断できるので、release メッセージは不要である。

また、X_i の値として他の変数 Y_j を含んだ構造データが送信されることがある。この場合もゴール送信時と同様に、送信ノード上の変数 Y_j を指す外部参照ポインタが受信ノードに作られる。

answer 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 にフックし、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。

2.2.3 Generic object

他の KL1 処理系にはない特徴として、KLIC には基本的なデータ型の他に generic object と呼ばれるデータ構造が用意されている。これは、KLIC ランタイムのカーネル部分を変更することなく、新たなデータ型などを拡張するためのものである。generic object は、それが表すデータの値など内部情報を格納するデータ領域と、メソッドと呼ばれるデータ操作用の C コードから成る。

generic object は data, generator, consumer の 3 種類が用意されており、後者の 2 種は一種の未具体化変数として扱われる。consumer ではその object に対する具体化が行われると`unify`メソッドが起動する。また generator では、値が参照されると`generate`メソッドが起動するほか、具体化が行われると consumer と同様に

`unify` メソッドが起動する。

2.3 AP1000 版 KLIC

現在公開されている KLIC 処理系はワークステーション上で汎用並列通信ライブラリである PVM を用いて動作する。我々はこの KLIC を並列計算機 AP1000 上へ移植しており [8]、本研究での性能評価には、この AP1000 版 KLIC を用いた。

AP1000 はメッセージ交換型の並列計算機であり、トーラス網で結合した最大 1024 個のセル、およびホストとなるワークステーションから構成される [9]。

現在のセル用 OS は、タイムスライス型のマルチプロセスがサポートされていない。そのため AP1000 版 KLIC では、複数ノードを 1 セルに割り当てられないという制限がある。このため、KL1 の論理的なノードは、物理的なセルと一対一対応する。また、割り込み通信もサポートされていないため、KLIC ランタイムが 1 リダクション毎にポーリングを行うことで、メッセージ到着を検出している。

3 通信最適化手法

3.1 手法の概要

3.1.1 KL1 分散処理系の問題点

KL1 では、論理的に並列な実行モデルや暗黙の通信・同期により、計算機の物理構成を意識することなく並列プログラムを容易に記述できる。KL1 分散処理系では、実行時にランタイムシステムが必要に応じた通信を行うことや、サスペンションによりゴール間の依存関係を保つことで、これらの利点を実現している。しかしながら、これらの動的処理のオーバヘッドにより、手続き型言語をベースとした並列言語処理系などと比較すると、実行速度が劣るという欠点がある。

例えば 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。しかし上記のオーバヘッドは解決できておらず、なかでもメッセージ交換型並列計算機を対象とした場合、細粒度通信の頻発が大きな速度低下の要因となっている。

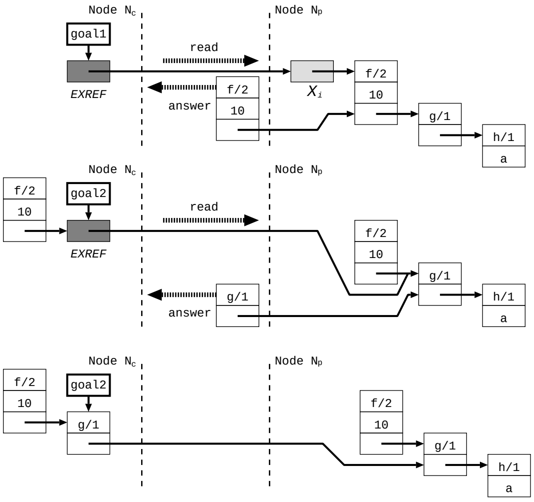


図 6: 構造データの送信

これは、動的なデータ構造やゴールスケジューリングのため、データ中の通信対象部分や定義・参照を行うノードが一般的には実行時まで判明せず、必要になった時点で送信を要求するという要求駆動的なデータのやりとりを行うためである。このため、次のような問題点を生じている。

1. 1 個のデータを受けとるという 1 論理メッセージのために、データ要求 (read)、返信 (answer)、および外部参照ポインタの解放 (release) と、3 個の物理メッセージが必要となる。
2. リストやファンクタなどの構造データを扱う場合、どの部分を参照するかが判らないため、個々の要素が参照される毎に通信を行ってしまう。例えばノード N_p 上の構造データ `f(10,g(h(a)))` について、ノード N_c 上でこのデータの `f(10,g(X))` という部分が参照されると、図 6のように 2 回に分けて送信が行われる。この結果、通信が細粒度化する。

問題点 2については、最初の要素が参照された時点で、構造データ全体を送信してしまうという実装も考えられる。この場合、通信回数は減らすことができるが、送信した構造データ内

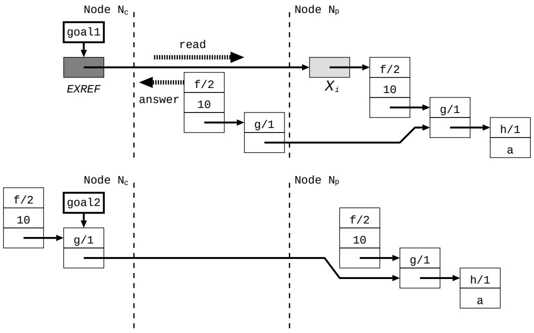


図 7: 構造データの一括送信

に受信側で参照しない要素があった場合、その部分の通信時間および受信側で占めるメモリが無駄になる。例えば図 6の例では、ノード N_cでは参照しない `h(a)` まで送信してしまう。

3.1.2 通信最適化の方針

3.1.1 節で述べた問題点を改善するため、本研究では次のような一括送信手法を提案する。

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を生成する場合を考える。P_pが P_cからのデータ要求メッセージを受信したとき、構造データ中のすでに具体化された部分のうち P_cが必要とする部分のみをすべて送るようにすれば、余分なデータを送信することなくメッセージ粒度を上げることができ、通信回数が減少する。これにより、のちのデータ要求回数を減らす、もしくはなくすことが可能である。

図 6の例では、N_p上の構造データである `f(10,g(h(a)))` のうち、N_cが参照する部分 `f(10,g(X))` を一度に送信する。このとき `h(a)` については、たとえこの部分が具体化されていても送信を行わない (図 7)。この結果、無駄なデータを送ることなく送信回数を 2 回から 1 回に減らすことができる。

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、タイプ解析と呼ばれる手法により、実行時にある変数がとり得るデータ型 (タイプ) を静的に決定できる。また、モード解析と呼ばれる手法により、各出現毎に変数の入出力方向 (モード) を静的に解析できる。

これらの情報を利用し、上記のようなデータ送信を行う KL1 コードを生成する。以下、このコードを一括送信コードと呼ぶ。また、一括送信コードより実行時に生成されるゴールを、一括送信ゴールと呼ぶ。

一括送信コードを KL1 で記述することにより、一括送信処理を KL1 ゴールの一つとして扱うことができ、送信する KL1 データの型判定やメモリ管理に KLIC ランタイムの機能を利用できる。また、一括送信ゴールの生成処理や送信データ格納バッファに generic object を用いることで、KL1 ランタイムのカーネルを変更することなく一括送信を実装することができる。

また、モード解析結果により変数の具体化・参照を行うプロセスがそれぞれ P_p, P_c と定まっていれば、read メッセージにより値要求を出した P_cで具体化が起こることはない。したがって release メッセージが不要になり、問題点 1が改善できる。

以下、まず 3.2節で上記のモード・タイプ情報を抽出する解析手法について述べる。次に 3.3 節で、一括送信を実現する KLIC ランタイムの拡張について説明する。最後に 3.4節で、拡張版ランタイムで利用可能な一括送信コードを解析結果より生成する手法について述べる。

なお、以下の説明では例として図 2に示したプログラム `stack` を用いる。

3.2 静的解析

3.2.1 入力プログラム

一般に論理型言語では、複数のプロセス $P_1, \ldots, P_n$が同一化により変数 vを共有している場合に、任意の P_iが vを具体化するようなプログラムを書くことができる。しかし KL1 では、すでに具体化された vを再び具体化しようとした場合には、プログラム全体が停止してしまう。このため、文法上は上記のようなプログラムを記述できるものの、一般に KL1 プログラムの多くでは vの具体化を行うプロセスは P_iのうちどれか一つであり、残りのプロセスはすべて参照を行う。このようなプログラムでは、プログラム中の具体化・参照箇所を一意に決定することができる。これを well-moded なプログラムと呼び、本手法ではこの性質を満たすプ

ログラムを対象とする。

具体的な解析対象としては、KLIC コンパイラがコンパイル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。

節の一般形は、次のようになる。

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

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_B	ボディ変数集合

以後の記述では述語 $p(v_1, \ldots, v_n)$ について、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 $H_{p/n}$, $B_{p/n}$、p/n の第 i 引数の集合をそれぞれ $H_{p/n,i}$, $B_{p/n,i}$ と表記する。また、$f(v_1, \ldots, v_m)$ で変数 $v_1, \ldots, v_m$ を含む構造データを表すことにする。

3.2.2 並列プロセス分割

KL1 ではプラグマにより並列実行するゴールを指定し、それ以外の部分は単一ノード上で実行される。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、そこから同一ノード上で実行される部分を、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 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、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 I と呼ぶ。

入力プログラムを並列プロセスに分割するアルゴリズムは、以下のようになる。

並列プロセス初期ゴール

1. $G_1 =$ {`main/0`}, $pmax = 1$ とする。
2. プログラム中の各 `@node` プラグマ付きゴール g/n 毎に、
   $\forall G_i \neq \{g/n\}$ $(i = 1, \ldots, pmax)$
   なら、
   $pmax \leftarrow pmax+1$, $G_{pmax} = \{g/n\}$
   とする。

並列プロセス

$i = 1, \ldots, pmax$ について、

1. $g_i \in G_i$ を初期ゴールとする並列プロセスを、節の集合 P_i で表す。初期値として、$P_i = \emptyset$ とする。
2. G_i からゴールを一つ取りだし g/n とする。
   g/n をクローズヘッドに持つ節の集合を $C_{g/n}$ とする。
3. $P_i \cap C_{g/n} = \emptyset$ なら $P_i \leftarrow P_i \cup C_{g/n}$ とし、$C_{g/n}$ のユーザ定義ボディゴールのうち `@node` プラグマのないものを G_i に入れる。
4. $G_i = \emptyset$ なら終了。そうでなければ 2へ。

プログラム `stack` の場合、`@node` プラグマ付きのゴールは `stack/2` だけであるから、`main/0` を初期ゴールとし `drive/2` で再帰するプロセス (節 1,2,3)、および、`stack/2` を初期ゴールとし自己再帰を繰り返すプロセス (節 4,5,6,7) に分割される。

3.2.3 モード解析

モードとは、あるゴールにおける変数の出現について、その位置で入力、出力のいずれであるかを表したものである。本論では変数 v_i のモードを $m(v_i)$ で表し、in, out, $unknown$ のいずれかをとるものとする。ここで、$goal(v_1, \ldots, v_n)$ 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。$m(v_i) = out$ とは、$goal/n$ 内で v_i が具体化されることを示す。$m(v_i) = unknown$ とは、プロセス内情報だけではモードが一意に決定できないことを示す。

また、$m(v_i)$ の逆モードを表す記号として、$\overline{m(v_i)}$ を用いる。逆モードは以下のように定義する。

$\overline{in} = out,\ \overline{out} = in,$
$\overline{unknown} = unknown$

クローズヘッド h については、h と同一化しうるボディゴールの対応した引数のモードと一致するものとする。

以下、具体的なモード解析手順を述べる。なお、式の記述では変数を v_i などとしているが、本節における解析結果で具体的な変数を指す場合は、節毎の変数番号を用いて `V(0)` のように表記することにする。

節内解析　まず各節毎に局所的な解析を行う。節内の変数について以下の制約条件を適用し、モードを決定する。

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。

2. ガードに出現する`=/2` 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) $=(v_i, v_j),\ v_i \in V_H,\ v_j \in V_B$
    $\rightarrow m(v_i) = \overline{m(v_j)}$
    (d) $=(v_i, v_j),\ =(v_k, v_j),$
    $v_i, v_k \in V_H,\ v_j \in V_B$
    $\rightarrow m(v_i) = m(v_k) = in,\ m(v_j) = out$

3. ボディに出現する`=/2` について
    (a) $=(v_i, atomic),\ v_i \in V_B$
    $\rightarrow m(v_i) = out$
    (b) $=(v_i, f(v_{j_1}, \ldots, v_{j_n})),\ v_i \in V_B$
    $\rightarrow m(v_i) = out$
    (c) $=(v_i, v_j),\ v_i, v_j \in V_B$
    $\rightarrow m(v_i) = \overline{m(v_j)}$

4. ある v_i の全出現について、
    (a) クローズヘッドでの出現が in モードのとき、ボディゴールでの出現はすべて in
    (b) クローズヘッドでの出現が out モード、またはクローズヘッドに出現しないとき、ボディゴールでの出現のうち一つだけが out で残りはすべて in

    ただし、条件 2dに該当する同一化ゴール 2 個に関しては、両者の v_j を合わせて一つの出現と考える。

5. $m(v_i) = in, v_i \in V_B$ 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、条件 2a, 2b, 2d, 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、条件 2c, 3a, 3b, 3c が成立する。さらに、節内では複数のゴールが同じ変数を具体化することはないから、条件 4が成立する。最後に、ボディ変数 v_k を参照するガードゴール g_i は、他のガードゴール g_j により v_k が具体化されないかぎり成功しない。ガードゴールがすべて成功しないとボディゴールは実行できないから、その節が実行可能であるためには、このような g_j が存在しなければならない。したがって条件 5 が成立する。

例えばプログラム `stack` のプロセス `main/0` では、`equal/2` は 2 個の整数値が一致するかどうか判定する組み込み述語であるから、節 2 の `equal(V(0),0)` において $mode($`V(0)`$) = in$ となる (条件 1)。さらに`=(V(2),V(0))` において、節 2 中で `V(0)` の出現は 2 箇所しかないから条件 4 より $mode($`V(0)`$) = out$、条件 2c より $mode($`V(2)`$) = in$ となる。

プログラム `stack` の節内モード解析結果を、図 8に示す。

プロセス内解析　続いてプロセス全体について、クローズヘッド・ボディゴール間の同一化によるモード波及を解析する。

3.2.1 節で述べたように well-moded を仮定しているので、互いに同一化可能なクローズヘッド・ボディゴール間で、対応する引数のモードが異なることはない。そこで、$H_{p/n,i}, B_{p/n,i}$ の

プロセス `main/0`

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

プロセス `stack/2`

```
1 stack(V(1:i),V(2:?))
  :- =(V(1:i),[]),=(V(2:?),V(0:?))
   | terminate(V(0:?)).
2 stack(V(3:i),V(4:?))
  :- =(V(3:i),[push(V(0:?))|V(1:?)]),
     =(V(4:?),V(2:?))
   | stack(V(1:?),p(V(0:?),V(2:?))).
3 stack(V(4:i),V(5:i))
  :- =(V(4:i),[pop(V(0:?))|V(1:?)]),
     =(V(5:i),p(V(2:?),V(3:?)))
   | =(V(0:?),V(2:?)),stack(V(1:?),V(3:?)).
4 terminate(V(1:?))
  :- =(V(1:?),V(0:?)).
```

変数は `V(番号:モード)` で表記。
モード　i: *in*,　o: *out*,　?: *unknown*

図 8: `stack` 節内モード解析結果

プロセス `main/0`

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

プロセス `stack/2`

```
1 stack(V(1:i),V(2:i))
  :- =(V(1:i),[]),=(V(2:i),V(0:o))
   | terminate(V(0:i)).
2 stack(V(3:i),V(4:i))
  :- =(V(3:i),[push(V(0:?))|V(1:o)]),
     =(V(4:i),V(2:o))
   | stack(V(1:i),p(V(0:?),V(2:i))).
3 stack(V(4:i),V(5:i))
  :- =(V(4:i),[pop(V(0:?))|V(1:o)]),
     =(V(5:i),p(V(2:?),V(3:o)))
   | =(V(0:?),V(2:?)),stack(V(1:i),V(3:i)).
4 terminate(V(1:i))
  :- =(V(1:i),V(0:o)).
```

変数は `V(番号:モード)` で表記。
モード　i: *in*,　o: *out*,　?: *unknown*

図 9: `stack` プロセス内モード解析結果

各要素中にモードが判明しているものが一つでもあれば、他の要素も同モードに決定できる。したがって、

$$
\begin{aligned}
& m(\exists v_j \in H_{p/n,i} \cup B_{p/n,i}) \\
& \quad = mode(\neq unknown) \\
& \Rightarrow m(\forall v_k \in H_{p/n,i} \cup B_{p/n,i}) \\
& \quad = mode(\neq unknown)
\end{aligned}
$$

となる。この結果、新たにモードが判明した v_k については、それを含む節について再び節内解析を行う。

この過程を繰り返し、モードの新規判明がなくなれば解析は終了する。この段階で決定できなかったモードに関しては *unknown* となる。

例えばプログラム `stack` のプロセス `stack/2` では、節3の節内解析によりクローズヘッドの第2引数が *in* モードであることが判明している。したがって、プロセス中のすべての `stack/2` において第2引数が *in* モードに決定できる。

プログラム `stack` の最終的なモード解析結果を、図 9に示す。

3.2.4　タイプ解析

2.1.2 節で述べたように、KL1 の変数は型を持たない。したがってタイプとは、ある変数が実行時にどのようなデータ型に具体化され得るかを示したものである。ただし、同じ変数が実行パスにより異なるデータ型に具体化されるようなプログラムも考えられるから、静的解析では具体値の型が一意に決まるとは限らない。そこで、タイプは可能性のある具体値の型の集合で表す。

変数は節毎にローカルであり、一度具体化さ

れた変数は値を変更できないから、同じ節内で同一変数が違う値をとることはない。したがって、節内の出現数にかかわりなく、変数毎に 1 個のタイプを考えればよい。

変数 v_i のタイプを $T(v_i) = \{t_1, \ldots, t_m\}$ で表すと、各 t_k は以下のいずれかの値をとる。なお、式の記述では前節と同様に変数を v_i などで表すが、本節における解析結果で具体的な変数を指す場合は、プロセス内の節番号および節毎の変数番号を用いて `V(1,0)` のように表記することにする。

1. $atom$, $integer$, `[]`
   それぞれアトム、整数、空リストを表す。空リストはアトムの一種であるが、ここでは独立したタイプとみなす。
2. $v_j(m_j)$
   構造データの要素として現れる変数を表す。m_j はモードであり、$in, out, unknown$ のいずれかをとる。$v_j(m_j)$ が実際にとるタイプは、$T(v_j)$ の値である。
   なお、解析結果で具体的な変数を指す場合は、プロセス内の節番号および節毎の変数番号を用いて、$\mathtt{V(1,0,}in\mathtt{)}$ のように表記する。
3. $[t_{car} \,|\, t_{cdr}]$, ファンクタ名 $(t_1, \ldots, t_n)$
   それぞれコンスおよびファンクタを表す。構造データのタイプは、car, cdr や各引数を各々のタイプに置き換えた形で表現する。
4. $unknown$
   解析の結果、最終的に判明しなかったタイプを表す。

また、b_j が具体値の時、$T(b_j)$ は b_j の型を表すものとする。例えば、

$T(10) = \{integer\}$,

$T(\mathtt{move(left)}) = \{\mathtt{move}(atom)\}$

である。

さらに、同一化による同値関係を、集合 $U_1, \ldots, U_M$ で表すことにする。U_k の各要素は変数 v_i、もしくは集合 $\hat{U}_l = \{u_{l_1}, \ldots, u_{l_m}\}$ であり、u_{l_j} は変数、またはボディゴールの引数に直接現れる具体値である。後者は、実行時に選択される節により、$\hat{U}_l$ の要素のいずれかが U_k 内の他の変数と同一化することを表している。以後、U_k を同一化集合、$\hat{U}_l$ を同一化候補集合と呼ぶ。

以下、具体的なタイプ解析手順を述べる。

節内解析　モード解析と同様に、まず各節毎に局所的な解析を行う。

節内の各ゴールに対し、次の操作を適用する。

1. 組み込み述語 $goal(\ldots, v_i, \ldots)$

   $\Rightarrow T(v_i) = \{$ その位置に来るべき型 $\}$

2. $v_i = b_m$　(b_m は具体値)

   $\Rightarrow T(v_i) = T(b_m)$
   ただし b_m が変数 v_j を含む構造データの場合、3.2.3 節の結果より、その部分の型は $v_j(m_j)$, $m_j = \overline{m(v_j)}$ とする。これは v_i 中の v_j に対応する部分について、モード $m(v_j) = in$ ならここで具体化され、モード $m(v_j) = out$ ならここで参照されるからである。

3. $v_i = v_j$

   $\Rightarrow v_i \in U_k$, $v_j \in U_l$ となる同一化集合 U_k, U_l がすでに両者とも存在すれば、これらを融合する。すなわち
   $U_k \leftarrow U_k \cup U_l$
   とし、U_l は削除する。
   U_k のみ存在する場合は、
   $U_k \leftarrow U_k \cup \{v_i, v_j\}$
   とする (U_l のみ存在する場合も同様)。
   どちらも存在しなければ $M \leftarrow M + 1$ とし、新たに $U_M = \{v_i, v_j\}$ を生成する。

上記の操作により判明しなかった $T(v_j)$ については、初期値として $T(v_j) = \emptyset$ とする。

例えばプログラム `stack` のプロセス `main/0` 節 2 では、操作 1, 2 により、それぞれ

$T(\mathtt{V(2,0)}) = \{integer\}$,

$T(\mathtt{V(2,1)}) = \{\mathtt{[]}\}$,

が判明する。また、操作 3により、

$U_1 = \{\mathtt{V(2,0)}, \mathtt{V(2,2)}\}$,

プロセス `main/0`
$T(\mathtt{V(1,0)}) = \{\}$
$T(\mathtt{V(2,0)}) = \{integer\}$
$T(\mathtt{V(2,1)}) = \{\mathtt{[]}\}$
$T(\mathtt{V(2,2)}) = \{\}$
$T(\mathtt{V(2,3)}) = \{\}$
$T(\mathtt{V(3,0)}) = \{integer\}$
$T(\mathtt{V(3,1)}) = \{\mathtt{[push(V(3,0,o))|V(3,2,o)]}\}$
$T(\mathtt{V(3,2)}) = \{\mathtt{[pop(V(3,3,i))|V(3,4,o)]}\}$
$T(\mathtt{V(3,3)}) = \{integer\}$
$T(\mathtt{V(3,4)}) = \{\}$
$T(\mathtt{V(3,5)}) = \{\}$
$T(\mathtt{V(3,6)}) = \{\}$
$T(\mathtt{V(3,7)}) = \{\}$
$T(\mathtt{V(3,8)}) = \{integer\}$

$U_1 = \{\mathtt{V(2,0),V(2,2)}\}$
$U_2 = \{\mathtt{V(2,1),V(2,3)}\}$
$U_3 = \{\mathtt{V(3,0),V(3,6)}\}$
$U_4 = \{\mathtt{V(3,1),V(3,7)}\}$
$U_5 = \{\mathtt{V(3,5),V(3,8)}\}$

プロセス `stack/2`
$T(\mathtt{V(1,0)}) = \{\}$
$T(\mathtt{V(1,1)}) = \{\mathtt{[]}\}$
$T(\mathtt{V(1,2)}) = \{\}$
$T(\mathtt{V(2,0)}) = \{\}$
$T(\mathtt{V(2,1)}) = \{\}$
$T(\mathtt{V(2,2)}) = \{\}$
$T(\mathtt{V(2,3)}) = \{\mathtt{[push(V(2,0,?))|V(2,1,i)]}\}$
$T(\mathtt{V(2,4)}) = \{\}$
$T(\mathtt{V(3,0)}) = \{\}$
$T(\mathtt{V(3,1)}) = \{\}$
$T(\mathtt{V(3,2)}) = \{\}$
$T(\mathtt{V(3,3)}) = \{\}$
$T(\mathtt{V(3,4)}) = \{\mathtt{[pop(V(3,0,?))|V(3,1,i)]}\}$
$T(\mathtt{V(3,5)}) = \{\mathtt{p(V(3,2,?),V(3,3,i))}\}$
$T(\mathtt{V(4,0)}) = \{\}$
$T(\mathtt{V(4,1)}) = \{\}$

$U_1 = \{\mathtt{V(1,0),V(1,2)}\}$
$U_2 = \{\mathtt{V(2,2),V(2,4)}\}$
$U_3 = \{\mathtt{V(3,0),V(3,2)}\}$
$U_4 = \{\mathtt{V(4,0),V(4,1)}\}$

変数は `V`（節番号, 節内の変数番号）または `V`（節番号, 節内の変数番号, モード）で表記。
モード　i: *in*,　o: *out*,　?: *unknown*

図 10: `stack` 節内タイプ解析結果

$$U_2 = \{\mathtt{V(2,1),V(2,3)}\}$$

が判明する。

プログラム `stack` の節内タイプ解析結果を、図 10に示す。

プロセス内解析　プロセス全体について、ボディゴール・クローズヘッド間の同一化によるタイプ波及を解析する。

アルゴリズムは、以下のようになる。

1. $\forall v_h \in H_{p/n,m},\ \forall v_b \in B_{p/n,m}$に対して、
   $\hat{U}_h = H_{p/n,m},\ \hat{U}_b = B_{p/n,m}$とし、
   $\forall U_h \ni v_h,\ \forall U_b \ni v_b$について、
   $U_h \leftarrow U_h \cup \{\hat{U}_b\},$
   $U_b \leftarrow U_b \cup \{\hat{U}_h\}$
   として、同一化候補集合$\hat{U}$を各Uに加える。ただしv_hに対し該当するU_hがない場合は、$M \leftarrow M+1,\ U_M = \{v_h\} \cup \{\hat{U}_b\}$とする。$v_b$についても同様である。

2. U_kに含まれる各変数のタイプの集合T_kを求める。以下これを同一化タイプ集合と呼ぶ。T_kは次のように定義できる。

   $$T_k = \textstyle\bigcup_{v_i \in U_k} T'(v_i),$$
   $$T'(v_i) = \begin{cases} \{v_i\} & \text{if } T(v_i) = \emptyset \\ T(v_i) & \text{otherwise} \end{cases}$$

   また、同一化候補集合$\hat{U}_l \in U_k$についても、同様に同一化候補タイプ集合
   $\hat{T}_l = \bigcup_{\forall u_m \in \hat{U}_l} T'(u_m)$
   を求め、
   $T_k \leftarrow T_k \cup \{\hat{T}_l\}$
   とする。ただし、
   $u_m = f(\ldots, v_j, \ldots),\ m(v_j) = m_j$
   のとき、
   $T(u_m) = f(\ldots, v_j(\overline{m_j}), \ldots)$
   とする。

3. Uは同値関係要素の集合であるから、$v_i, v_j, \hat{U}_l \in U_k$のとき、$T(v_i)$と$T(v_j)$、また$\hat{T}_l$の少なくとも一要素と$T(v_i)$が、それぞれ同一化可能でなければならない。
   そこで、同一化タイプ集合T_kの各要素に対し、以下の操作を適用する。

   (a) 同一要素は一方を削除する。すなわち、
   $t_i, t_j \in T_k,\ t_i = t_j$のとき、
   $T_k \leftarrow T_k - \{t_j\}$
   $\hat{T}_l \in T_k, t_i, t_j \in \hat{T}_l,\ t_i = t_j$ のとき、
   $\hat{T}_l \leftarrow \hat{T}_l - \{t_j\}$

   (b) $v_i, v_j \in U_k$のとき、v_i, v_jは同タイプとなるから、
   $t_i, v_j \in T_k,\ t_i \neq \forall v_m$のとき、
   $T(v_j) = \{t_i\},\ T_k \leftarrow T_k - \{v_j\},$
   また、$\forall \hat{T}_l (\ni v_j)$ について、

$$\hat{T}_l \leftarrow \hat{T}_l \cup T(v_j) - \{v_j\}$$

とする。

(c) 二つの構造データ $t_{n_1}, t_{n_2} \in T_k$ の場合、両者は同一化可能でなければならない。したがって、

$$f(\ldots, t_i, \ldots) \in T_k,$$
$$f(\ldots, v_j(m_j), \ldots) \in T_k,$$
$$t_i \neq \forall v_m(m_m)$$

のとき、

$$T_k \leftarrow T_k - \{f(\ldots, v_j(m_j), \ldots)\},$$
$$T_l(\ni v_j) \leftarrow T_l \cup t_i - \{v_j\}$$

とする。また、

$$f(\ldots, v_i(m_i), \ldots) \in T_k,$$
$$f(\ldots, v_j(m_j), \ldots) \in T_k$$

のとき、

v_i, v_j について、節内解析の操作 3 と同様に U の更新を行う。

(d) 同一化候補タイプ集合中の 2 つの要素について、引数の一つのみが異なる構造データである場合は、両者を一つにまとめ異なる部分をタイプ集合で表す。すなわち、

$$f(\ldots, t_i, \ldots), f(\ldots, t_j, \ldots) \in \hat{T}_l$$

のとき、

$$\hat{T}_l \leftarrow \hat{T}_l \cup \{f(\ldots, \{t_i, t_j\}, \ldots)\}$$
$$-\{f(\ldots, t_i, \ldots), f(\ldots, t_j, \ldots)\}$$

とする。

(e) 同一化候補タイプ集合 $\hat{T}_l \in T_k$ の要素が一つになった場合、

$v_i \in T_k$ は必ずこれと同タイプになる。したがって、

$$\hat{T}_l \in T_k,\ \hat{T}_l = \{t_i\}$$

のとき、

$$T_k \leftarrow T_k \cup \{t_i\} - \{\hat{T}_l\}$$

とする。

$$\hat{T}_l \in T_k,\ \hat{T}_l = \{v_i\}$$

のとき、

$$T_k \leftarrow T_k \cup \{v_i\} - \{\hat{T}_l\}$$

とし、$\exists T_m(\ni v_i)$ が存在すれば、

$$T_k \leftarrow T_k \cup T_m$$

として T_m, U_m は削除する

4. 3での各操作で更新された U_k, T_l があれば、それぞれ 2, 3から繰り返す。

   新規情報が得られなくなったら、5へ。

5. T_k の要素が v_i または $\hat{T}_{l_j}$ の場合、v_i のタイプはこの節内で一意に決定されない。したがって、$\hat{T}_{l_j}$ の各要素のうち、型が判明しているものの集合で表す。

(a) $\exists v_i, \in T_k,\ \exists \hat{T}_{l_j} \in T_k$ のとき、

$$T(v_i) = \bigcup_{\forall \hat{T}_{l_j} \in T_k} \hat{T}_{l_j}$$

とする。

(b) 決定した $T(v_i)$ を、他の $T(v_j)$ に波及させる。すなわち、

$\forall v_j$ について $v_i \in T(v_j),\ T(v_i) \neq \emptyset$ なら、

$$T(v_j) \leftarrow T(v_j) \cup T(v_i) - \{v_i\}$$

(c) $T(v_i)$ について、まだ v_j が含まれている場合、この部分についてはタイプ不明とする。すなわち、

$\exists v_j \in T(v_i)$ なら、

$$T(v_i) \leftarrow T(v_i) \cup \{unknown\}$$
$$- \bigcup_{\forall v_j \in T(v_i)} \{v_j\}$$

とする。

6. 5の適用によってもタイプが判明しなかった v_i については、プロセス内情報だけではタイプ不明である。したがって、

$T(v_i) = \emptyset$ である $\forall v_i$ について、

$$T(v_i) = \{unknown\}$$

$\forall f(\ldots, v_i(m_j), \ldots) \in T(\forall v_m)$ について、

$$f(\ldots, v_i(m_j), \ldots)$$
$$\rightarrow f(\ldots, unknown, \ldots)$$

とする。

7. $\forall T(v_i)$ について、3a, 3d と同様に集合要素の削減・融合を図る。つまり、

$t_k, t_l \in T(v_i),\ t_k = t_l$ のとき、

$$T(v_i) \leftarrow T(v_i) - \{t_l\}$$

$f(\ldots, t_k, \ldots), f(\ldots, t_l, \ldots) \in T(v_i)$ のとき、

$$T(v_i) \leftarrow T(v_i) \cup \{f(\ldots, \{t_k, t_l\}, \ldots)\}$$
$$-\{f(\ldots, t_k, \ldots), f(\ldots, t_l, \ldots)\}$$

とする。

以上のアルゴリズムをプログラム `stack` のプロセス `stack/2` に適用すると、

プロセス main/0

$U_1 = \{\texttt{V(2,0)},\texttt{V(2,2)}, \{integer,\texttt{V(3,5)}\}\}$
$U_2 = \{\texttt{V(2,1)},\texttt{V(2,3)}, \{\texttt{V(1,0)},\texttt{V(3,4)}\}\}$
$U_3 = \{\texttt{V(3,0)},\texttt{V(3,6)}, \{integer,\texttt{V(3,5)}\}\}$
$U_4 = \{\texttt{V(3,1)},\texttt{V(3,7)}, \{\texttt{V(1,0)},\texttt{V(3,4)}\}\}$
$U_5 = \{\texttt{V(3,5)},\texttt{V(3,8)}, \{\texttt{V(2,2)},\texttt{V(3,6)}\}\}$
$U_6 = \{\texttt{V(1,0)}, \{\texttt{V(2,3)},\texttt{V(3,7)}\}\}$
$U_7 = \{\texttt{V(3,4)}, \{\texttt{V(2,3)},\texttt{V(3,7)}\}\}$
$T_1 = \{integer\}$
$T_2 = \{\texttt{[]}, \{\texttt{V(1,0)},\texttt{V(3,4)}\}\}$
$T_3 = \{integer\}$
$T_4 = \{\texttt{[push(V(3,0,o))|V(3,2,o)]},$
$\quad \{\texttt{V(1,0)},\texttt{V(3,4)}\}\}$
$T_5 = \{integer\}$
$T_6 = \{\texttt{V(1,0)},$
$\quad \{\texttt{[]}, \texttt{[push(V(3,0,o))|V(3,2,o)]}\}\}$
$T_7 = \{\texttt{V(3,4)},$
$\quad \{\texttt{[]}, \texttt{[push(V(3,0,o))|V(3,2,o)]}\}\}$

プロセス stack/2

$U_1 = \{\texttt{V(1,0)},\texttt{V(1,2)}, \{\texttt{V(4,1)}\},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,i))}, \texttt{V(3,3)}\}\}$
$U_2 = \{\texttt{V(2,2)},\texttt{V(2,4)},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,i))}, \texttt{V(3,3)}\}\}$
$U_3 = \{\texttt{V(3,0)},\texttt{V(3,2)}\}$
$U_4 = \{\texttt{V(4,0)},\texttt{V(4,1)}, \{\texttt{V(1,0)}\}\}$
$U_5 = \{\texttt{V(1,1)}, \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$
$U_6 = \{\texttt{V(2,3)}, \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$

$U_7 = \{\texttt{V(3,4)}, \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$
$U_8 = \{\texttt{V(2,1)}, \{\texttt{V(1,1)},\texttt{V(2,3)},\texttt{V(3,4)}\}\}$
$U_9 = \{\texttt{V(3,1)}, \{\texttt{V(1,1)},\texttt{V(2,3)},\texttt{V(3,4)}\}\}$
$U_{10} = \{\texttt{V(3,5)},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,i))}, \texttt{V(3,3)}\}\}$
$U_{11} = \{\texttt{V(3,3)},$
$\quad \{\texttt{V(1,2),V(2,4),V(3,5)}\}\}$
$T_1 = \{\texttt{V(1,0)},\texttt{V(1,2)},\texttt{V(4,0)},\texttt{V(4,1)},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,o))}, \texttt{V(3,3)}\}\}$
$T_2 = \{\texttt{V(2,2)},\texttt{V(2,4)},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,o))}, \texttt{V(3,3)}\}\}$
$T_3 = \{\texttt{V(3,0)},\texttt{V(3,2)}\}$
$T_5 = \{\texttt{[]}, \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$
$T_6 = \{\texttt{[push(V(2,0,?))|V(2,1,i)]},$
$\quad \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$
$T_7 = \{\texttt{[pop(V(3,0,?))|V(3,1,i)]},$
$\quad \{\texttt{V(2,1)},\texttt{V(3,1)}\}\}$
$T_8 = \{\texttt{V(2,1)},$
$\quad \{\texttt{[]},\texttt{[push(V(2,0,?))|V(2,1,i)]},$
$\quad \texttt{[pop(V(3,0,?))|V(3,1,i)]}\}\}$
$T_9 = \{\texttt{V(3,1)},$
$\quad \{\texttt{[]},\texttt{[push(V(2,0,?))|V(2,1,i)]},$
$\quad \texttt{[pop(V(3,0,?))|V(3,1,i)]}\}\}$
$T_{10} = \{\texttt{p(V(3,2,?),V(3,3,i))},$
$\quad \{\texttt{p(V(2,0,?),V(2,2,o))}, \texttt{V(3,3)}\}\}$
$T_{11} = \{\texttt{V(3,3)},$
$\quad \{\texttt{V(1,2),V(2,4),}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}\}$

変数はV(節番号, 節内の変数番号）またはV(節番号, 節内の変数番号, モード）で表記。

モード　i: *in*, o: *out*, ?: *unknown*

図 11: stack 同一化集合解析結果

$$H_{stack/2,1} = \{\texttt{V(1,1)},\texttt{V(2,3)},\texttt{V(3,4)}\}$$
$$B_{stack/2,1} = \{\texttt{V(2,1)},\texttt{V(3,1)}\}$$

プロセス main/0

$T(\texttt{V(1,0)}) = \{\texttt{[]},$
$\quad \texttt{[push(V(3,0,o))|V(3,2,o)]}\}$
$T(\texttt{V(2,0)}) = \{integer\}$
$T(\texttt{V(2,1)}) = \{\texttt{[]}\}$
$T(\texttt{V(2,2)}) = \{integer\}$
$T(\texttt{V(2,3)}) = \{\texttt{[]}\}$
$T(\texttt{V(3,0)}) = \{integer\}$
$T(\texttt{V(3,1)}) = \{\texttt{[push(V(3,0,o))|V(3,2,o)]}\}$
$T(\texttt{V(3,2)}) = \{\texttt{[pop(V(3,3,i))|V(3,4,o)]}\}$
$T(\texttt{V(3,3)}) = \{integer\}$
$T(\texttt{V(3,4)}) = \{\texttt{[]},$
$\quad \texttt{[push(V(3,0,o))|V(3,2,o)]}\}$
$T(\texttt{V(3,5)}) = \{integer\}$
$T(\texttt{V(3,6)}) = \{integer\}$
$T(\texttt{V(3,7)}) = \{\texttt{[push(V(3,0,o))|V(3,2,o)]}\}$
$T(\texttt{V(3,8)}) = \{integer\}$

プロセス stack/2

$T(\texttt{V(1,0)}) = \{\ \texttt{p(V(2,0,?),V(2,2,o))}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}$
$T(\texttt{V(1,1)}) = \{\texttt{[]}\}$
$T(\texttt{V(1,2)}) = \{\ \texttt{p(V(2,0,?),V(2,2,o))}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}$
$T(\texttt{V(2,0)}) = \{unknown\}$
$T(\texttt{V(2,1)}) = \{\texttt{[]},$
$\quad \texttt{[push(V(2,0,?))|V(2,1,i)]},$
$\quad \texttt{[pop(V(3,0,?))|V(3,1,i)]}\}$
$T(\texttt{V(2,2)}) = \{\texttt{p(V(2,0,?),V(2,2,o))},\texttt{V(3,3)}\}$
$T(\texttt{V(2,3)}) = \{\texttt{[push(V(2,0,?))|V(2,1,i)]}\}$
$T(\texttt{V(2,4)}) = \{\texttt{p(V(2,0,?),V(2,2,o))},\texttt{V(3,3)}\}$
$T(\texttt{V(3,0)}) = \{unknown\}$
$T(\texttt{V(3,1)}) = \{\texttt{[]},$
$\quad \texttt{[push(V(2,0,?))|V(2,1,i)]},$
$\quad \texttt{[pop(V(3,0,?))|V(3,1,i)]}\}$
$T(\texttt{V(3,2)}) = \{unknown\}$
$T(\texttt{V(3,3)}) = \{\ \texttt{p(V(2,0,?),V(2,2,o))}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}$
$T(\texttt{V(3,4)}) = \{\texttt{[pop(V(3,0,?))|V(3,1,i)]}\}$
$T(\texttt{V(3,5)}) = \{\texttt{p(V(3,2,?),V(3,3,i))}\}$
$T(\texttt{V(4,0)}) = \{\ \texttt{p(V(2,0,?),V(2,2,o))}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}$
$T(\texttt{V(4,1)}) = \{\ \texttt{p(V(2,0,?),V(2,2,o))}$
$\quad \texttt{p(V(3,2,?),V(3,3,i))}\}$

変数はV(節番号, 節内の変数番号）またはV(節番号, 節内の変数番号, モード）で表記。

モード　i: *in*, o: *out*, ?: *unknown*

図 12: stack プロセス内タイプ解析結果

であるから、1より

$$U_8 = \{\texttt{V(2,1)}, \{\texttt{V(1,1)},\texttt{V(2,3)},\texttt{V(3,4)}\}\}$$
$$U_9 = \{\texttt{V(3,1)}, \{\texttt{V(1,1)},\texttt{V(2,3)},\texttt{V(3,4)}\}\}$$

などとなる。したがって、2より

$$\begin{aligned} T_8 = \{&\texttt{V(2,1)},\\ &\{\texttt{[]},\\ &\ \texttt{[push(V(2,0,}unknown\texttt{))|V(2,1,}in\texttt{)]},\\ &\ \texttt{[pop(V(3,0,}unknown\texttt{))|V(3,1,}in\texttt{)]}\}\} \end{aligned}$$

$$\begin{aligned} T_9 = \{&\texttt{V(3,1)},\\ &\{\texttt{[]},\\ &\ \texttt{[push(V(2,0,}unknown\texttt{))|V(2,1,}in\texttt{)]}, \end{aligned}$$

$$\mathtt{[pop(V(3,0,}unknown\mathtt{))|V(3,1,}in\mathtt{)]}\}\}$$

となり、5より

$$T(\mathtt{V(2,1)}) = T(\mathtt{V(3,1)}) =$$
$$\{\mathtt{[]},$$
$$\mathtt{[push(V(2,0,}unknown\mathtt{))|V(2,1,}in\mathtt{)]},$$
$$\mathtt{[pop(V(3,0,}unknown\mathtt{))|V(3,1,}in\mathtt{)]}\}\}$$

が得られる。

プログラム `stack` 全体の同一化集合およびプロセス内タイプの解析結果を、それぞれ図 11, 図 12に示す。

3.3 KLIC ランタイムの拡張

本節では、3.2 節で述べた解析情報を利用して一括送信を行うような、KLIC ランタイムの拡張について述べる。

以後の説明では、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を生成する場合を考える。P_pが throw メッセージにより N_cに送るボディゴール (P_cの初期ゴール) を $goal(v_1,\ldots,v_n)$ とする。

また、プロセス P_cにおいて、すべてのクローズヘッド $goal(v_{k_1},\ldots,v_{k_n})$ の第 i 引数の集合を A_iで表す。つまり $A_i = H_{goal/n,i}$である。この A_iを用いて、

$$T(A_i) = \bigcup_{\forall v_j \in A_i} T(v_j),$$
$$m(A_i) = m(\forall v_j \in A_i)$$

と定義する。

このとき、プログラム全体が well-moded であることを仮定しているから、P_cの解析結果より判明した $m(A_i)$ は P_pの解析結果により判明した $m(v_i)$ と一致しているものとする。

3.3.1 送信モード

各 v_iについて、$m(v_i) = unknown$ なら送信方向が不明であるから、従来と同様に一要素づつの送信を行う。$m(v_i) = in$ なら P_p側が、$m(v_i) = out$ なら P_c側が、それぞれ一括送信を行う。ただし、v_iの取る具体値がアトミックデータの場合には一括送信を行う意味がないので、余分なオーバヘッドを避けるため、従来と同じ送信方法をとる。

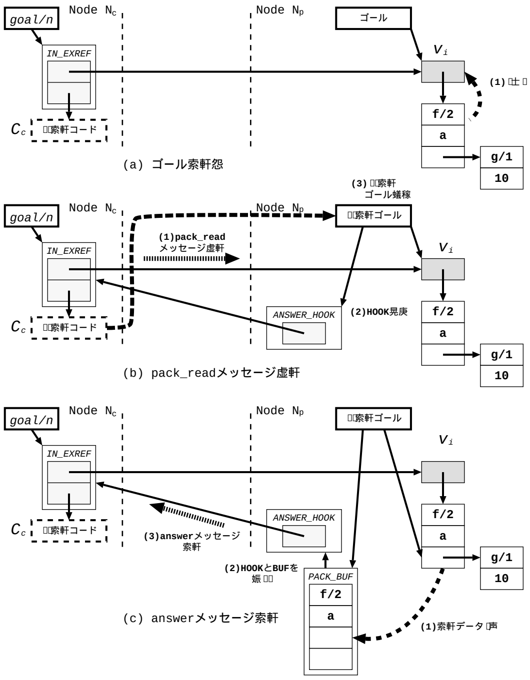


図 13: 送信モード **in** のときの処理

そこで、各引数について送信モードを以下のように定義する。

in $m(v_i) = in$、かつ $T(A_i)$ が構造データ型を含む場合。

out $m(v_i) = out$、かつ $T(v_i)$ が構造データ型を含む場合。

normal $m(v_i) = unknown$、あるいは $T(v_i)$ または $T(A_i)$ が $unknown$ またはアトミック型の場合。

3.3.2 一括送信処理

$goal/n$ の各引数 v_iについて、送信モードにより以下の処理を行う。

送信モード **in** のとき
v_iは P_pの側で構造データに具体化されるから、後に P_cから read メッセージにより値が要求されたとき、P_pが一括送信を行わなければならない。このとき使用する一括送信コード C_cはゴール $goal/n$ 第 i 引数に対し P_cの解析結果より生成されたものになる。

そこで、throw メッセージによりゴール $goal/n$ を送信する際に、送信モード **in** 用の外部参照ポインタである generator object `IN_EXREF` を N_cに送る。`IN_EXREF` は内部情報として C_cへのポインタを持つ。(図 13(a))。

P_cにおいて v_iの参照が生じた場合、通常は `EXREF` が N_pに read メッセージを送信するが、`IN_EXREF` は pack_read メッセージを送信する。このとき、`IN_EXREF` が保持していた一括送信コードへのポインタ C_cが pack_read メッセージに格納される。N_pでは pack_read メッセージの受信により、C_cで指定された一括送信ゴールをゴールキューの先頭につなぎ、返信処理を行う consumer object `ANSWER_HOOK` を作成する。返信先の位置などは、このとき `ANSWER_HOOK` に格納される (図 13(b))。

一括送信ゴールは、送信対象 v_iおよび `ANSWER_HOOK` を引数にとる。v_iが未具体化なら具体化されるまで待ち、v_iの具体値を一括送信するのに必要なバッファサイズを数え、一括送信バッファである data object `PACK_BUF` を生成する。続いて v_iの具体値をたどりながら、`PACK_BUF` のメソッドにより具体値の各要素を `PACK_BUF` に格納する。格納が完了すると `PACK_BUF` を `ANSWER_HOOK` と同一化して終了する。

この最後の操作により `ANSWER_HOOK` の `unify` メソッドが起動し `PACK_BUF` に格納された内容を answer メッセージとして一括送信する (図 13(c))。

送信モード **out** のとき

v_iは P_cの側で構造データに具体化されるから、ゴール送信時には必ず未具体化変数である。したがって、P_cにより具体化が行われ unify メッセージが P_pに送信される時、一括送信を行わなければならない。このとき使用する一括送信コード C_pは、送信モード **in** の場合と逆に、ゴール $goal/n$ 第 i 引数に対し P_pの解析結果より生成されたものになる。

そこで送信モード **in** の場合と同様、throw メッセージによりゴール $goal/n$ を送信する際に、送信モード **out** 用の外部参照ポインタである generator object `OUT_EXREF` を N_cに送る。`OUT_EXREF` は内部情報として C_pへのポインタを持つ (図 14(a))。

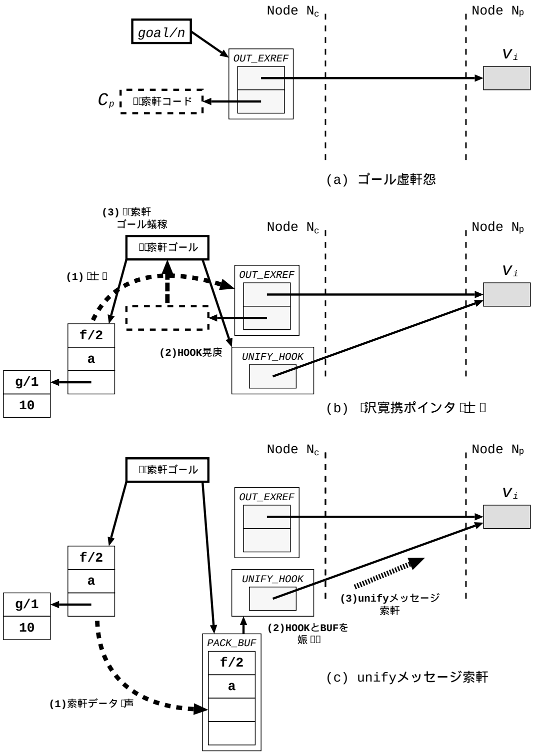


図 14: 送信モード **out** のときの処理

P_cでの v_iの具体化は、P_p上の v_iを指す外部参照ポインタと具体値との同一化になる。従来の送信方法では `EXREF` が unify メッセージにより具体値を P_pに送信するが、この場合は `OUT_EXREF` の `unify` メソッドが起動する。このメソッドでは C_pで指定された一括送信ゴールをゴールキューの先頭につなぎ、送信処理を行う consumer object `UNIFY_HOOK` を作成する (図 14(b))。

一括送信ゴールは送信モード **in** の場合と同様に data object `PACK_BUF` を生成して送信内容を格納し、`PACK_BUF` を `UNIFY_HOOK` と同一化する。この操作により `UNIFY_HOOK` の `unify` メソッドが起動し、バッファに格納された内容を unify メッセージとして一括送信する (図 14(c))。

送信モード **normal** のとき

従来と同様に、2.2.2 節で述べた処理を行う。

3.3.3　未具体化変数を含む構造データの処理

一括送信ゴールが v_iの具体値をバッファ `PACK_BUF` に格納する際、具体値が未具体化変数 v_kを含む構造データの場合がある。この場合、v_kについては今回具体値を送信することはできないから、ゴール送信時の v_iと同様に v_kを指す外部参照ポインタのみを送信することになる。

このとき、v_kの具体値についても一括送信受信側のプロセスが参照することが判明している場合は、いずれ v_kが具体化された後で、v_kについての一括送信を行うのが望ましい。そこで、以下の処理を行う。

v_iの送信モードが **in** のとき

$f(\ldots, v_k(mode), \ldots) \in T(A_i)$ とする。$mode$ が in または out で $T(v_k)$ が構造データを含む場合、次のようになる。

$mode = in$ の場合 ：v_kについても引き続き P_cが参照する。したがって P_pは `IN_EXREF` を送信する。この `IN_EXREF` には、P_cの解析結果より生成された、v_kの一括送信コード C'_c へのポインタが格納される。

$mode = out$ の場合 ：v_iとは逆に、v_kは P_cが具体化し、P_pが参照する。この場合 N_p上の変数 v_kに対し、P_cが unify メッセージにより具体値を送信することになる。したがって P_pは `OUT_EXREF` を送信する。この `OUT_EXREF` には、P_pの解析結果より生成された v_kの一括送信コード C'_pへのポインタが格納される。

$mode = unknown$ のとき、および $T(v_k)$ が $unknown$ またはアトミック型のとき、一括送信は不可能もしくは無意味である。したがって v_kについては従来と同じ送信方法をとることとし、特別な処理は行わない。

v_iの送信モードが **out** のとき

$f(\ldots, v_l(mode), \ldots) \in T(v_i)$ とする。$mode$ が in または out で $T(v_l)$ が構造データを含む場合、次のようになる。

$mode = in$ の場合 ：v_lについても引き続き P_pが参照する。ただし、N_c上の v_lを N_p上の外部参照ポインタが指す形になるから、v_lの参照は pack_read および answer メッセージを使ったものになる。したがって P_cは `IN_EXREF` を送信する。この `IN_EXREF` には、P_pの解析結果より生成された v_lの一括送信コード C'_pへのポインタが格納される。

$mode = out$ の場合 ：v_iとは逆に、v_lは P_pが具体化し、P_cが参照する。この場合 N_c上の変数 v_lに対し、P_pが unify メッセージにより具体値を送信することになる。したがって P_cは `OUT_EXREF` を送信する。この `OUT_EXREF` には、P_cの解析結果より生成された v_lの一括送信コード C'_cへのポインタが格納される。

$mode = unknown$ のとき、および $T(v_l)$ が $unknown$ またはアトミック型のとき、v_iの送信モードが **in** のときと同様に、特別な処理は行わない。

例えばプログラム `stack` では、`stack/2` 第 1 引数のリストにおいて cdr 部分が未具体化という状況が発生する。この場合 cdr 部分のモードも同様に in であるので、N_p上で新たに生成された `IN_REF` を指す外部参照ポインタが N_cに送信される。したがって、P_cで cdr 部分の参照が生じると、再び一括送信が行われる。

3.4　一括送信コードの生成

3.3 節で述べたように、一括送信ではデータ受信側の解析結果より生成した一括送信コードを利用する。初期ゴール $init(v_1^h, \ldots, v_m^h)$ で `@node` プラグマ付きのゴール $goal(v_1^b, \ldots, v_n^b)$ を含むプロセス Pを考えると、3.2 節の静的解析結果において $m(v_i^h) = in$ である $\forall v_i^h$および $m(v_j^b) = out$ である $\forall v_j^b$は、他プロセスが具体化した値を P内で参照する。また、$m(v_i^h) = out$ である $\forall v_i^h$および $m(v_j^b) = in$ である $\forall v_j^b$について、v_i^h, v_j^bのタイプが $v_k(out)$ を含む場合、v_kについてはやはり具体値を P内で参照する。

したがって、これらについてそれぞれ一括送信コードを生成すればよい。

3.4.1 一括送信コードの内容

一括送信ゴールは送信対象 v_i、および返信処理を行う consumer object ANSWER_HOOK または UNIFY_HOOK(以下、両者を合わせて HOOK と呼ぶ) を引数にとり、以下の処理を行う。

一括送信量の計算 一括送信ゴールはデータ格納中にサスペンドする可能性があるため、一括送信バッファPACK_BUF を KL1 の data object で実現することにより、送信毎に動的な確保を行っている。このため、PACK_BUF 生成前に、必要なバッファサイズを計算する必要がある。
この計算は後述の送信データ格納と同様に、v_iの具体値の構造をたどりながら必要なデータサイズを累計すればよい。

一括送信バッファ data object PACK_BUF を生成する。 PACK_BUF の生成用メソッド new はバッファサイズ S および HOOK への参照ポインタ H を引数にとり、生成された PACK_BUF への参照ポインタ B を返す。

送信データ格納 v_iの具体値構造をたどりながら、送信するデータをバッファに格納する。PACK_BUF には、put_atomic など各データ型格納用のメソッドが用意されている。

バッファ内容送信 PACK_BUF の send メソッドを呼び出す。send メソッドでは、PACK_BUF 自身を H が指す HOOK と同一化する。これによって HOOK の unify メソッドが起動し、PACK_BUF のバッファ内容を物理的に送信する。

3.4.2 送信データ格納処理

KLIC では、送信側が (処理関数へのポインタ, 送信データ) の組でメッセージを作成し、受信側は受信メッセージ中に与えられたポインタの関数を呼び出すことでメッセージ処理を行う。

データ要求側プロセスが参照するデータ型を T_Rとすると、T_Rの各要素 t_kに対し、以下のような送信データ格納節を生成する。各節はクローズヘッドの引数として、対象となるデータ構造への参照ポインタ V、および一括送信バッファ PACK_REF への参照ポインタ B をとり、ガード部において V の具体値が t_kであるかどうかチェックする。

1. t_kがアトミック型 $(atom, integer, \texttt{[]})$
   メソッド put_atomic により、アトミック型受信関数へのポインタ、および V の具体値をバッファに格納する。

2. t_kがコンス型
   メソッド put_cons によりコンス型受信関数へのポインタをバッファに格納する。続いて、V の car および cdr をそれぞれ新たな V として再帰呼出を行い、各々を処理する。

3. t_kがファンクタ型
   メソッド put_functor によりファンクタ型受信関数へのポインタ、およびファンクタ名 $(functor/m)$ をバッファに格納する。続いて、V の各引数をそれぞれ新たな V として再帰呼出を行い、各々を処理する。

4. t_kが不明 $(unknown)$
   メソッド put_unused を呼び出す。
   put_unused は V の具体値がアトミックの場合のみ値をバッファに格納し、他の場合は外部参照ポインタ EXREF 受信関数へのポインタ、および V への外部参照ポインタ EXREF を格納する。

5. t_kが $v_j(in)$
   v_jについても、現在この一括送信処理を行っているプロセスが具体化する。したがって、引続きデータ格納処理を行う必要がある。そこで、t_kが $v_j(in)$ の場合は V を引数に v_j の送信データ格納節を呼び出す。呼び出される格納節は、$T(v_j)$ の各要素に対し、本節の手法を適用して生成する。

6. t_kが $v_j(out)$
   今回の一括送信でデータを受信する側のプロセスが、v_jを具体化する。3.3.3 節で述べたように、現在の送信モードにかかわらず v_jの送信モードを **out** にすればよい。そこでメソッド put_out_exref により、OUT_EXREF 受信関数へのポインタ、お

```
packsend_stack_2_1_top(V,H)
 :- packsend_stack_2_1_count_init(V,S),
    packsend_stack_2_1_pack_init(V,S,H).
packsend_stack_2_1_count_init(V,S)
 :- packsend_stack_2_1_count(V,0,S).
packsend_stack_2_1_pack_init(V,S,H)
 :- wait(S) | generic:new(packbuf,B,S,H),
    packsend_stack_2_1(B,V), packsend_end(B).
packsend_end(B)
 :- generic:sendbuf(B).

packsend_stack_2_1_count(V,S0,S1)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | S1 := S0+5.
otherwise.
packsend_stack_2_1_count(V,S0,S1)
 :- V = [] | S1 := S0+2.
packsend_stack_2_1_count(V,S0,S3)
 :- V = [V0|V1] | S1 := S0+1,
    packsend_stack_2_1_cons_count(V0,S1,S2),
    packsend_stack_2_1_count(V1,S2,S3).

packsend_stack_2_1(B,V)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_exref(B,V,0).
otherwise.
packsend_stack_2_1(B,V)
 :- V = [] | generic:put_atomic(B,V).
packsend_stack_2_1(B,V)
 :- V = [V0|V1] | generic:put_cons(B,V),
    packsend_stack_2_1_cons(B,V0),
    packsend_stack_2_1(B,V1).

packsend_stack_2_1_cons_top(V,H)
 :- packsend_stack_2_1_cons_count_init(V,S),
    packsend_stack_2_1_cons_pack_init(V,S,H).
packsend_stack_2_1_cons_count_init(V,S)
 :- packsend_stack_2_1_cons_count(V,0,S).
packsend_stack_2_1_cons_pack_init(V,S,H)
 :- wait(S) | generic:new(packbuf,B,S,H),
    packsend_stack_2_1_cons(B,V),
    packsend_end(B).
```

```
packsend_stack_2_1_cons_count(V,S0,S1)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | S1:=S0+5.
otherwise.
packsend_stack_2_1_cons_count(V,S0,S2)
 :- V = push(V0) | S1:=S0+2, S2:=S1+4.
packsend_stack_2_1_cons_count(V,S0,S2)
 :- V = pop(V0) | S1:=S0+2, S2:=S1+4.

packsend_stack_2_1_cons(B,V)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_exref(B,V,1).
otherwise.
packsend_stack_2_1_cons(B,V)
 :- V=push(V0) | generic:put_functor(B,V),
    generic:put_unused(B,V0).
packsend_stack_2_1_cons(B,V)
 :- V=pop(V0) | generic:put_functor(B,V),
    generic:put_unused(B,V0).

packsend_stack_2_2_top(V,H)
 :- packsend_stack_2_2_count_init(V,S),
    packsend_stack_2_2_pack_init(V,S,H).
packsend_stack_2_2_count_init(V,S)
 :- packsend_stack_2_2_count(V,0,S).
packsend_stack_2_2_pack_init(V,S,H)
 :- wait(S) | generic:new(packbuf,B,S,H),
    packsend_stack_2_2(B,V), packsend_end(B).
packsend_end(B)
 :- generic:sendbuf(B).

packsend_stack_2_2_count(V,S0,S3)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | S1:=S0+5.
otherwise.
packsend_stack_2_2_count(V,S0,S3)
 :- V = p(V0,V1) | S1:=S0+2, S2:=S1+4, S3:=S2+4.

packsend_stack_2_2(B,V)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_exref(B,V,0).
otherwise.
packsend_stack_2_2(B,V)
 :- V = p(V0,V1) | generic:put_functor(B,V),
    generic:put_unused(B,V0),
    generic:put_unused(B,V1).
```

図 15: `stack` 一括送信コード

および v_j への外部参照ポインタ `OUT_EXREF` をバッファに格納する。

また、t_k の値に関わりなく、以下の KL1 節を生成する。

7. 未具体化変数処理節

   6 に該当しない `V` が未具体化の場合、その部分は今回送信することを諦め、次回の pack_read メッセージで再び一括送信を行う。具体的にはメソッド `put_in_exref` により、`IN_EXREF` 受信関数へのポインタ、および `V` への外部参照ポインタ `IN_EXREF` をバッファに格納する。

この手法により生成したプログラム `stack` の一括送信コードを、図 15 に示す。

4 性能評価

本章では 3 章で述べた一括送信手法の性能評価について述べる。

評価用処理系には 2.3 節で説明した AP1000 版 KLIC を用いた。また、3.3 節で述べた KLIC ランタイムの拡張については、本研究室の伊川

が実装を行った [8]。

4.1　評価プログラム

今回の性能評価では、次の 3 種類のプログラムを対象とした。

`stack`(図 2)　2 章で説明の例として用いた、ノード間でスタック操作を行うプログラムである。このプログラムについては 2.1.3 節で述べたので、本節では説明を省略する。

`mastermind`(付録 A)　マスターマインドとは 1 ~ Nまでの任意の数字 L 個の並びを推測するゲームである。ある試行での推測解が正解でなかった場合は、推測解と正解で数字および位置が一致したものの個数、および数字は合っているが位置が異なっているものの個数、のそれぞれがヒントとして与えられ、次の試行での推測解を決定する際の情報として利用できる。

本プログラムでは、このゲームにおいて M 解以内の試行で正解を当てるような推測の並びを、全解探索する。探索の途中で分岐が生じた場合は、ノード番号が隣のノードから順に仕事を割り当てていく。

`nqueen`(付録 B)　N-queen 問題は、チェスの駒の一つクイーンを、互いに相手を取ることのできない位置を占めるよう、$N \times N$の盤面上にN個配置するという問題である。

本プログラムでは、この条件を満たす配置を全解探索し、見つかった解の個数を出力する。上記の条件を満たすには、盤面上の各列には必ずクイーンが 1 個ずつ配置されなければならないという性質がある。これを利用し、盤面の状態を "i 列目のクイーンが存在する行" を表す整数 $(1, \ldots, N)$ を N個並べたリストで表現する。1 列目のクイーンが存在する行により問題を N 個に分割し、各々を各ノードに割り当てて並列探索を行う。

4.2　最適化結果

評価プログラムのそれぞれについて、3 章で述べた手法により一括送信コードの生成を行った。

`stack`　解析の結果、`stack/2` の引数は共に in モードであった。したがってプロセス `stack/2` の解析結果より、図 15の一括送信コードが得られた。`stack/2` 第 1 引数に対し、ファンクタを要素とするリストを一括送信するコードが得られており、このデータ構造に対する一括送信の効果が期待できる。

また、プロセス `main/0` の解析ではファンクタ `push/1` の引数を参照することが判明している。したがって、本来ならこの部分の一括送信コードが生成されるが、この場合は引数のタイプがアトミック (整数) のみであるので、通常の送信が行われる。

`mastermind`
ゴール送信の対象となる `spawn/8` にはモード in および out の引数が共に現れている。したがって本プログラムでは送信モード **in** および **out** の一括送信が行われる。

また、引数のいくつかはタイプがリストであり、一括送信の効果が期待できる。たとえば第 1 引数は整数リストを要素とするリストであり、各プロセスが最初のリスト要素である整数リスト全体を参照することが判明している。このため必要充分な一括送信コードの生成に成功している。

`nqueen`　`mastermind` と同様に引数のいくつかはリストであり、これらに対し一括送信が生じる。しかし探索の分岐毎に負荷分散を行う `mastermind` と異なり、本プログラムでは最初に一度負荷分散を行った後、各ノードがそれぞれ独立した処理を行う。したがって通信は、最初の負荷分散時、および最後に解の個数を集計するときにしか生じない。このため処理全体に占める通信の割合は小さく、一括送信による速度向上はあまり期待できない。

4.3　実行結果

前記の 3 プログラムについて、オリジナルの AP1000 用 KLIC ランタイム上、および一括送信コードを用いた拡張版ランタイム上のそれぞれで実行し、実行時間および物理通信の回数を

表 1: stack 実行結果

	一括送信なし		一括送信あり	
セル	時間 (秒)	通信回数	時間 (秒)	通信回数
1	1.7	0	—	—
2	24.1	130004	10.3	30003

表 2: mastermind 実行結果

	一括送信なし		一括送信あり	
セル	時間 (秒)	通信回数	時間 (秒)	通信回数
1	2.0	0	—	—
2	22.7	190762	11.2	49501
4	15.7	263667	7.9	47081
8	11.3	278663	5.2	48872
16	9.9	290703	4.5	52496
32	8.7	296751	4.3	55245

表 3: nqueen 実行結果

	一括送信なし		一括送信あり	
セル	時間 (秒)	通信回数	時間 (秒)	通信回数
1	42.7	0	—	—
2	24.8	4104	23.2	369
4	12.6	6576	11.9	427
6	9.0	7439	8.7	760
8	8.6	7496	8.0	346
10	7.8	7713	7.4	193
11	5.7	8508	5.6	809

計測した。それぞれの計測結果を、表 1, 2, 3 に示す。また、mastermind および nqueen についてノード数による実行時間の変化を図 16,17 に示す。

なお、各プログラムは 64 セル構成の AP1000 上で、-LSEND -R 0 オプションを指定して実行した。

stack　push, pop の繰り返し回数を 10000 回とした。また、main/0 と stack/2 の 2 プロセスしか存在しないため、セル 1,2 台で評価を行った。

セル 2 台で一括送信ありの場合、一括送信なしの場合に比べ 2.3 倍の速度向上が得られた。物理通信回数は約 1/4 になっており、4.2 節で述べたように stack/2 第 1 引数に対して一括送信の効果が得られている。

また、本プログラムは演算量が少なく、通信時間が実行時間を支配してしまう。このため、上記の通信回数減少がそのまま速度向上に反映されている。

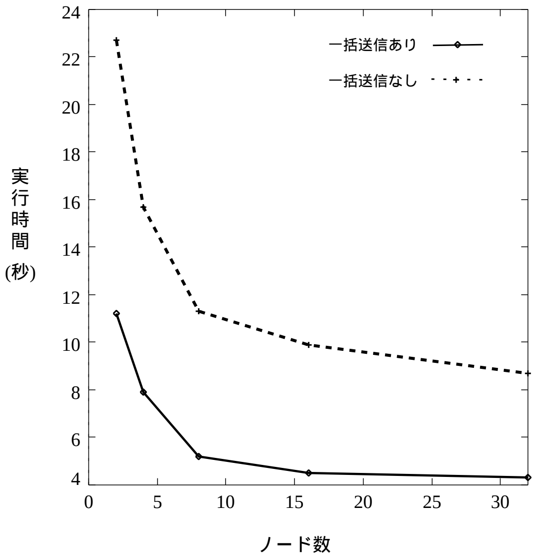


図 16: mastermind 実行時間

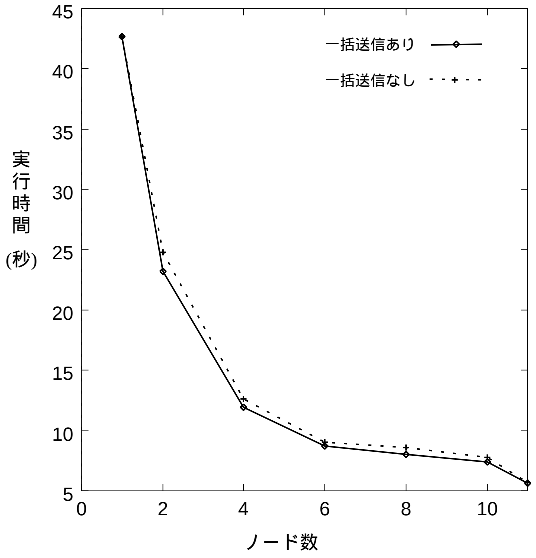


図 17: nqueen 実行時間

mastermind　解く問題は $(L, M, N) = (5, 3, 2)$ とした。

本プログラムでは自分を起点としてノード番号順にゴール送信を行うことで負荷分散を行う。したがって、負荷分散単位である推測解の個数

を超えるノード数で実行すると、負荷にばらつきが生じてしまう。今回の問題では推測解として考えられる組合せが高々32通りなので、セル32台までについて評価を行った。

一括送信により、物理通信回数は約1/4～1/6に減少し、約2倍という良好な速度向上が得られた。これは4.2節で述べたように、`spawn/8`第1引数などについて適切な一括送信コードが得られたためである。

しかしながら、一括送信の有無にかかわらず、台数効果はあまり得られていない。これは、並列探索の負荷分散単位が小さいため、`stack`同様に通信時間が実行時間を支配してしまうためである。

`nqueen`　問題の大きさは$N = 11$とした。

本プログラムでも、負荷分散単位である列の数を超えるノード数では、負荷にばらつきを生じる。そこで11セルまでについて評価を行った。

`mastermind`と異なり負荷分散は最初に一度行うだけであり、通信回数は比較的少ない。このためセル11台の場合、一括送信なしで7.5倍、一括送信ありで7.6倍と、良好な台数効果が得られている。そのかわり一括送信の効果は小さく、通信回数は1/10から1/40と大幅に減少したにもかかわらず、実行速度は2%から8%程度の短縮に留まっている。

4.4　考察

`stack`や`mastermind`のように通信回数が多く実行時間を支配してしまう場合、一括送信により大きな効果が得られた。これらのプログラムは逐次実行した方が高速であり、並列実行に意味がないともいえるが、大規模な並列プログラムの一部にこのような通信の集中が生じる場合も考えられるから、そのようなボトルネックの解消策として一括送信は有効である。また、`nqueen`のように通信量が少なく良好な台数効果が得られるプログラムでも、効果は低いものの速度の改善が得られた。

以上の結果から、一括送信は一般に通信最適化方法として有効であり、対象プログラムの性質にかかわらず速度が改善されることが判明した。

5　問題点と解決方法

本章では、今回提案した一括送信の問題点と解決方法について述べる。

5.1　一括送信実装方式の改良

5.1.1　一括送信バッファの静的確保

現在の実装ではpack_readメッセージを受信する毎に、data object `PACK_BUF`により一括送信データ格納用バッファを動的に確保している。

この方式にはKLICランタイムにあまり手を加えずに済み、余分なメモリを占有する心配がないという利点がある。しかし、必要なバッファサイズを計算するために一度データ構造をたどる必要があり、送信オーバヘッドが増加する。また、確保したバッファは送信終了後ガーベジとなるため、ガーベジコレクション(GC)の回数を増加させる原因になる。

そこで、プログラム実行の最初に固定長の格納用バッファを静的確保する方法が考えられる。この場合、必要バッファサイズの計算は不要であり、GC回数の増加も避けられる。ただし、一括送信ゴールの実行中に新たにpack_readメッセージが到着した場合、それに対する一括送信ゴールを実行してしまうと、格納中の一括送信データを上書きしてしまう。そこでKLICランタイムに一括送信ゴールキューを拡張し、pack_readメッセージ受信により生成された一括送信ゴールは、このキューにつながれて一括送信バッファが空くのを待つようにする。

この方式では格納用バッファ分のメモリが常に占有されてしまう。大規模な構造データも一括送信できるようにするには、このバッファを充分大きく取る必要があり、ヒープなど他の領域を圧迫することになる。しかし大規模なデータの処理にはある程度時間がかかることが予想され、一度に送信するよりは適当なサイズに分割して送る方が、パイプライン効果が見込める可能性がある。また、大規模なデータを一度に送ってしまうと、それらが長期間ヒープ上に存在するため、GC回数を増加させる危険もある。

したがって、格納用バッファサイズを適当に設定し、一括送信コードによるデータ格納中に

このサイズを超えた場合は、具体化済みデータが残っていても即座に送信を行う。適当なバッファサイズについては、今後性能評価を行って決める必要がある。

5.2　送信方式の改良

5.2.1　遅延送信

一括送信では、一括送信ゴールの実行時に具体化されている部分のみが対象となる。したがって、具体化速度に対して pack_read メッセージによる値要求の頻度が高い場合、結果的にほとんどまとめて送信されることがなく、効果が得られない可能性がある。

また、送信モード **out** の引数の場合、`OUT_EXREF` が具体化されると一括送信が行われる。このため、リストを順に具体化するような場合、先頭要素を具体化した時点で一括送信ゴールが実行されてしまい、結果的に従来と同じ送信になってしまう。

これらの場合にも一括送信の効果を上げるには、ある程度の具体化が進むまで一括送信を遅らせる方法が考えられる。具体的には、まず一括送信と同様に送信対象 v_i の具体値をたどりながら、送信データのサイズのみ集計する。このサイズが閾値未満で送信データ中に未具体化変数 v_j が存在する場合、一括送信ゴールを v_j にフックし一括送信を中断する。v_j が具体化されるとサイズ集計を再開し、データサイズが閾値を超えた時点で実際に一括送信ゴールを実行し、一括送信を行う。これにより、メッセージの粒度を常に一定に保つことができる。この送信方法を遅延送信と呼ぶことにする。

遅延送信ではプロセス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危険がある。これを解決するには、KLIC ランタイムを変更し一定期間内にデータ格納が再開しなかった場合はその時点で格納済みのデータのみを送信する、あるいは、プログラム中のデータ依存関係を解析しデッドロックを生じない場合のみ遅延送信を行う、といった方法が必要になる。

5.2.2　単方向送信

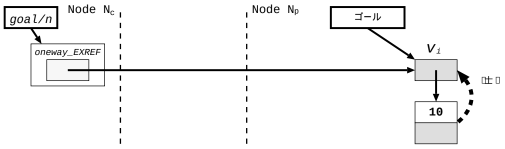
3.1.1 節で述べたように、KLIC で細粒度通信が頻発する理由は、1 論理メッセージのためにデータ要求 (read)、返信 (answer)、および外部参照ポインタ解放 (release) と物理メッセージ 3 個を要することと、構造データの各要素が参照されるたびに通信を行うことである。本論では後者の解決方法として一括送信を提案し、これにより前者の release メッセージも不要になった。しかし、まだ read(pack_read), answer と双方向のメッセージが必要である。

KL1 のストリーム通信では `stack` の場合のように、プロセス P_p が具体化するリストの要素をプロセス P_c が末尾まで次々に参照する形が少なくない。この場合、P_p が具体化する要素は必ず P_c が要求することになる。したがって P_c が他のノードに移動することがなければ、一度ノード N_c 上の P_c が決定すると、P_p は新たな要素を具体化する度に、要求を待たずに N_c に送信することができる。これにより read メッセージが不要になり、物理通信回数を半分にすることができる。この送信方法を単方向送信と呼ぶことにする。

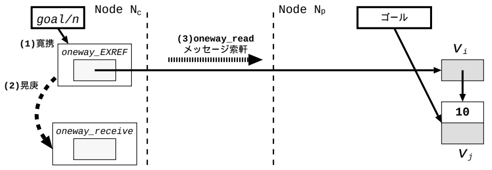
現在の一括送信において送信モード **in** であるような $goal/n$ の引数 v_i が上記の条件を満たすとき、これを送信モード **oneway** とする。このとき、単方向送信は以下に述べる方法で可能になる。また送信モード **out** の場合は、unify メッセージによりもともと単方向の送信が行われている。

P_p は throw メッセージによりゴール $goal/n$ を送信する際に、モード **in** 用外部参照ポインタ `IN_EXREF` の代わりにモード **oneway** 用外部参照ポインタである generator object `ONEWAY_EXREF` を N_c に送信する (図 18a)。

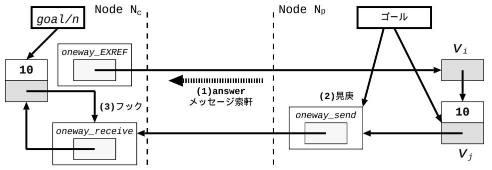
P_c で v_i の参照が生じると、`ONEWAY_EXREF` の `generate` メソッドが起動する。このメソッドは consumer object `ONEWAY_RECEIVE` を生成し、通常の read メッセージの代わりに oneway_read メッセージを N_p に送信する。このとき oneway_read メッセージ中には、単方向送信の送信先として `ONEWAY_RECEIVE` への外部参照ポインタを格納しておく (図 18b)。



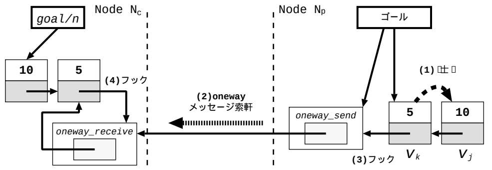
(a) ゴール索軒忽



(b) N_c のoneway_readメッセージ索軒



(c) N_p のanswerメッセージ索軒



(d) N_p のonewayメッセージ索軒

図 18: 単方向送信

N_pでは oneway_read メッセージを受信すると、v_iの具体値を `ONEWAY_EXREF` に answer メッセージとして送信し、consumer object `ONEWAY_SEND` を生成して具体値中の未具体化変数 v_j にフックさせる。`ONEWAY_SEND` は内部情報として、単方向送信先である `ONEWAY_RECEIVE` への外部参照ポインタを持つ。

N_cでは answer メッセージ受信により `ONEWAY_EXREF` の `unify` メソッドが起動する。受信した具体値中の未具体化変数は、本来なら N_p上の v_jへの外部参照ポインタになるが、この `unify` メソッドにより `ONEWAY_RECEIVE` を指すポインタになる。すなわち、`ONEWAY_RECEIVE` がフックする。また、`ONE_WAY_RECEIVE` も内部情報として、このポインタへのポインタを格納する (図 18c)。

P_pで v_jが具体化されると `ONEWAY_SEND` の `unify` メソッドが起動し、v_jの具体値を `ONEWAY_RECEIVE` に oneway メッセージにより送信してから v_j 中の未具体化変数 v_k に自身を再びフックさせる。一方、oneway メッセージを受信した `ONEWAY_RECEIVE` では、自身がフックした変数を受信データで具体化し、再び受信した具体値中の未具体化変数にフックする (図 18d)。

以上の処理により、一度 P_cから read メッセージが送られた後は、oneway メッセージにより次々と具体値が単方向送信される。

単方向送信では一要素毎に送信してしまうから、一論理メッセージあたりの物理通信回数は半減するかわりに一括送信の効果は得られない。これを解決するには遅延送信と組み合わせ、ある程度の具体化が進むたびに単方向の一括送信を行うようにすればよい。

6 おわりに

本論では、並列論理型言語 KL1 の処理系 KLIC を対象とし、静的解析情報を用いた通信最適化手法について述べた。

本手法では、モード・タイプ解析により変数の入出力方向や実行時にとり得る型を静的解析する。次にこの情報より必要なデータのみを一括送信するコードを生成し、実行時に KLIC ランタイムがこのコードを利用することによって、粒度の高い一括送信を行い物理通信回数を削減する。

本論ではこの静的解析手法、一括送信コードの生成手法、および一括送信を行うための KLIC ランタイムの拡張方法について、それぞれ述べた。

また、3 種類のプログラムにより本手法の性能評価を行った。この結果、通信頻度の高い `stack` や `mastermind` では通信回数が 1/4 程度に減少し、約 2 倍の良好な速度向上を得た。また、通信頻度の低い `nqueen` でも、実行時間に対し最大 8%速度が向上した。したがって本手法は、通信頻度の高いプログラムに対して有効であると同時に、通信頻度の低いプログラムに適用しても弊害は生じないといえる。

今回の性能評価に用いたプログラムはいずれも小規模なものであったが、いずれ大規模な実用プログラムにおいて、本手法の性能評価を行う必要がある。また、5 章で述べた改良案の実装および評価も、今後の課題である。

謝辞

本研究の機会を与えてくださった富田眞治教授に、深甚な謝意を表します。

適切な御指導を賜わった中島浩助教授、および数々の有用な御意見を戴いた森眞一郎助手に心から感謝致します。

また、各種資料を提供して下さった (財) 新世代コンピュータ開発機構、および並列計算機 AP1000 の実行環境を御提供頂きました (株) 富士通研究所に感謝致します。

さらに、共同研究者である伊川雅彦氏をはじめとして、日頃御討論頂く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] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, Vol. 5, pp. 207–229, 1988.

[3] 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.

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

[5] 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.

[6] 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.

[7] 仲瀬明彦, 藤瀬哲朗, 近山隆. ポータブル KL1 処理系 KLIC の概要. 情報処理学会第 47 回全国大会 講演論文集, pp. 5–55–5–56, October 1993.

[8] 伊川雅彦. 並列論理型言語処理系 KLIC における通信の高速化. 京都大学特別研究報告書, February 1995.

[9] 富士通株式会社. AP1000 プログラム開発手引書（1）C 言語インタフェース, November 1989.

[10] 中島浩. 並列論理型言語 (KL1) の実装最適化に関する調査研究報告書. Technical report, 財団法人 新世代コンピュータ技術開発機構, March 1994.

[11] 大野和彦, 中島浩, 富田眞治. 静的解析による並列論理型言語の実行最適化. 情処研報 94-PRG-18, pp. 17–24, 1994.

[12] 大野和彦, 伊川雅彦, 森眞一郎, 中島浩, 富田眞治. 並列論理型言語 KL1 の最適化手法. 情処研報 94-OS-67, pp. 183–190, 1994.

A mastermind プログラムリスト

```
main :- solve(3,2,A), display(A).
solve(NG,NC,Out)
 :- code(C), length(C,0,CL), gen(NC,Clist), genGuess(Guesses,Clist,CL),
    spawn_cl0(Guesses,Guesses,CL,[],0,NG,Out,[]).
code(C) :- C=[1,2,1,2,1].
genGuess(Out,L,CL) :- N:=CL-1, addmyself([[]],L,In,[]), genGuess(N,In,L,Out).
genGuess(N,In,L,Out) :- N>0 | N1:=N-1, addmyself(In,L,Temp,[]),
                              genGuess(N1,Temp,L,Out).
genGuess(0,In,_,Out) :- Out=In.
addmyself([I|Is],Myself,O1,O3) :- combo(Myself,I,O1,O2),
                                  addmyself(Is,Myself,O2,O3).
addmyself([]    ,_     ,O1,O2) :- O1=O2.
combo([C|Cs],L,O1,O3) :- O1=[[C|L]|O2], combo(Cs,L,O2,O3).
combo([]    ,_,O1,O2) :- O1=O2.
spawn_cl0(Guesses,Plist,CL,A,N,NG,S0,S2)
 :- nextnode(Node), spawn_cl1(Guesses,Plist,CL,A,N,NG,S0,S2,Node).
spawn_cl1(Guesses,Plist,CL,A,N,NG,S0,S2,Node)
 :- wait(Node) | spawn(Guesses,Plist,CL,A,N,NG,S0,S2)@node(Node).
nextnode(Node) :- builtin:current_node(CC,TC), CL:=(CC+1) mod TC.
spawn([]    ,_    ,_ ,_,_,_ ,S0,S1) :- S0=S1.
spawn([G|Gs],Plist,CL,A,N,NG,S0,S2) :- test(A,G,Plist,CL,A,N,NG,S0,S1),
                                       spawn_cl0(Gs,Plist,CL,A,N,NG,S1,S2).
test([p(OldBulls,OldCows,Tried)|Gs],Guess,Plist,CL,A,N,NG,S0,S1)
 :- eval(Guess,Tried,Bulls,Cows),
    check(OldCows,Cows,OldBulls,Bulls,Gs,Guess,CL,Plist,A,N,NG,S0,S1).
test([]                            ,Guess,Plist,CL,A,N,NG,S0,S1)
 :- code(Code),eval(Guess,Code,Bulls,Cows),
    loop(Bulls,Cows,Guess,CL,Plist,A,N,NG,S0,S1).
eval(X,Y,Bulls,Cows) :- exact(X,Y,Bulls,0,X1,Y1), inexact(X1,Y1,Cows,0).
exact([]    ,T2    ,N,I,S1,S2) :- N=I, S2=T2, S1=[].
exact([H|T] ,[H|T2],N,I,S1,S2) :- I1:=I+1, exact(T,T2,N,I1,S1,S2).
otherwise.
exact([X|T1],[Y|T2],N,I,X1,Y1) :- X1=[X|S1], Y1=[Y|S2], exact(T1,T2,N,I,S1,S2).
inexact(_      ,[]    ,N,I) :- I=N.
inexact([]     ,_     ,N,I) :- I=N.
otherwise.
inexact([Ht|Tt],Choice,N,I) :- delete(Choice,Ht,F,F,Choice,Tt,N,I).
delete([]    ,_,_,_        ,Choice,Tt,N,I) :- inexact(Tt,Choice,N,I).
delete([X|Y] ,X,F,NewChoice,_     ,Tt,N,I) :- I1:=I+1
                                            | F=Y, inexact(Tt,NewChoice,N,I1).
otherwise.
delete([X1|Y],X,F,NewChoice,Choice,Tt,N,I)
 :- F=[X1|Z], delete(Y,X,Z,NewChoice,Choice,Tt,N,I).
check(Sum,Sum,Bulls,Bulls,Gs,Guess,CL,Plist,A,N,NG,S0,S1)
 :- test(Gs,Guess,Plist,CL,A,N,NG,S0,S1).
otherwise.
check(_  ,_  ,_    ,_    ,_ ,_    ,_ ,_    ,_,_,_ ,S0,S1) :- S0=S1.
loop(Bulls,Cows,Guess,CL,Plist,A,N,NG,S0,S1)
 :- Bulls<CL, N1:=N+1, N1<NG
  | spawn_cl0(Plist,Plist,CL,[p(Bulls,Cows,Guess)|A],N1,NG,S0,S1).
loop(CL   ,_   ,_    ,CL,_    ,A,_,_ ,S0,S1) :- S0=[A|S1].
otherwise.
loop(_    ,_   ,_    ,_ ,_    ,_,_,_ ,S0,S1) :- S0=S1.
display(A) :- count(A,N),io:outstream([print(N),nl]).
gen(N,X) :- N>0   | M:=N-1, X=[N|Xs], gen(M,Xs).
gen(N,X) :- N=:=0 | X=[].
length([_|T],X,Y) :- X1:=X+1, length(T,X1,Y).
length([]   ,X,Y) :- X=Y.
count(L,N) :- count(L,0,N).
count([]    ,M,N) :- M=N.
count([X|Xs],M,N) :- M1:=M+1, count(Xs,M1,N).
```

B　nqueen プログラムリスト

```
main :- generate(11,[],L), nqueenD(L,[],X,[]), display(X).
generate(N,T,L) :- N>0  | T1=[N|T], N1:=N-1, generate(N1,T1,L).
generate(N,T,L) :- N=<0 | L=T.
nqueenD([N|T],R,I,O) :- worknode(N,M), nqueenD(M,N,T,R,I,O).
nqueenD([]   ,R,I,O) :- I=O.
worknode(N,M) :- builtin:current_node(_,TC), M:=N mod TC.
nqueenD(M,N,T,R,I,O) :- M>0   | append(T,R,NN), nqueenD(T,[N|R],I,X),
                                c1(N,1,NN,[],[],X,O)@node(M).
nqueenD(M,N,T,R,I,O) :- M=:=0 | append(T,R,NN), nqueenD(T,[N|R],I,X),
                                c1(N,1,NN,[],[],X,O).
c1(T,D,N,[P|R],B,I,O):- T=\=P+D, T=\=P-D | D1:=D+1, c1(T,D1,N,R,B,I,O).
c1(T,D,N,[P|_],B,I,O):- T=:=P+D | I=O.
c1(T,D,N,[P|_],B,I,O):- T=:=P-D | I=O.
c1(T,D,N,[]   ,B,I,O):- queen(N,[],[T|B],I,O).
queen([P|U],C    ,L,I,O) :- append(U,C,N), c1(P,1,N,L,L,I,X),
                            queen(U,[P|C],L,X,O).
queen([]   ,[_|_],_,I,O) :- I=O.
queen([]   ,[]   ,L,I,O) :- I=[L|O].
display(X) :- count(X,N), io:outstream([print(N),nl]).
count(L,N) :- count(L,0,N).
count([X|Xs],M,N) :- M1:=M+1, count(Xs,M1,N).
count([]    ,M,N) :- M=N.
append([A|X],Y,Z) :- Z=[A|Z1], append(X,Y,Z1).
append([]   ,Y,Z) :- Z=Y.
```