2001年度 修士論文

DKLIC 処理系における分散論理変数の資源管理

Resource Management for Distributed Logic Variables of DKLIC

提出日: 2002年2月5日
指導: 上田 和紀 教授

早稲田大学大学院 理工学研究科 情報科学専攻
学籍番号: 600P031-4

高木 祐介

目 次

第1章　はじめに **1**
1.1　本論文の構成 1

第2章　分散プログラミングの現状と問題点 **2**
2.1　プロセス (Processes) 2
2.2　ソケット (Sockets) 3
2.3　RPC (Remote Procedure Call) 3
2.4　ORB (Object Request Broker) 4
2.5　Mobile Agents 4
2.6　本質的な並行性 (Concurrency) 5
2.7　無視できない遅延時間 (Latency) 7
2.8　部分的な故障 (Fault Tolerance) 7
2.9　保安性 (Security) 7

第3章　分散 KL1 言語の挑戦 **9**
3.1　KL1 言語とは 9
3.2　KL1 言語の概要 9
3.3　言語核に組み込まれた同期と自動排他制御 10
3.4　通信路 (Channels) 11
3.5　Type / Mode System 12
3.6　KLIC 処理系 12
3.7　関連研究 13

第4章　DKLIC 処理系の目標 **14**
4.1　本研究が前提とする分散環境 14
4.2　DKLIC 処理系の構成 14
4.3　DKLIC 全体の仕様 15
4.4　本研究の目標と範囲 15

第5章　プロトコル DKLICP **18**
5.1　二者通信のプロトコル 18
5.2　分散論理変数のライフサイクル 19
5.3　誰が ID を再利用するか 21

5.4 三者通信のプロトコル 22

第6章 dklicio の設計 **25**
6.1 分散論理変数 distributed 25
6.2 ID manager (Name Server) 26
6.3 変数表オブジェクト variable_table 27

第7章 dklicio の実装 **31**
7.1 主な開発環境 31
7.2 分散論理変数オブジェクト 31
7.3 変数表オブジェクト 34
7.4 変数ネームサーバ 36
7.5 モジュール dklicio 36

第8章 DKLIC 処理系の評価と考察 **38**
8.1 既存の試作からの改良点 38
8.2 KLIC 並列実装との比較 38
8.3 中継通信の排除 39
8.4 Oz との比較 39
8.5 通信例外処理 39
8.6 ストリームへ連続した ID を割り当てて返事をバッファする . . 40
8.7 線形で非 mobile なストリームの ID 即時再利用 40
8.8 strictly linear ID allocation 41
8.9 今後の課題 41

第9章 まとめ **43**

参考文献 **45**

付録A 補足 **48**
A.1 モードなしのプログラムを等価なモード付きプログラムへ書き換える 48
A.2 DKLIC-XMLP: inter-operability 49

付録B DKLIC 処理系の仕様と実装 **50**
B.1 dklicio 50
B.2 変数ネームサーバ 50
B.3 変数表オブジェクト 50
B.4 fusesusp 50

付 録 C　DKLIC 処理系の応用例 **51**
C.1　Remote Predicate Call 51
C.2　Chat Server 51
C.3　検索サーバ 51
C.4　Mobile Agent 51

第1章　はじめに

本章では、本論文の概要を述べる。

普及している手続き型言語では、プロセス群の分散、同期、通信プロトコルの実装といった分散プログラミングに不可欠な処理の記述が非常に面倒である。

本研究では、動的プロセス生成、自動同期、非同期メッセージ通信を宣言的に記述する KL1 言語に基づき、既存の KLIC 処理系を拡張して宣言型ソケット通信の機能を提供する DKLIC 処理系を実装した。このことによって、直観的でなかった分散処理の記述を宣言的に行えるようにした。

分散論理変数の直結や廃棄を検出することによって、既存の DKLIC 処理系では不可能であった二重送信や通信の中継を排除し、分散論理変数の ID を早期に再利用することを可能とした。

また、KL1 言語に蓄積された高度な静的解析の技術を分散プログラミングに適用することを可能とした。それだけでなく、既存の静的解析の技術である型・モード・線形性体系に明確には現れない通信路の確立（分散論理変数の open）という概念を明確に示した。分散プログラミングにおいては、素性の知れない通信データやモバイルコードに対して静的に検証を行えることは非常に重要である。

また、静的意味論を変えずに様々な通信手法を試みることは今後のネットワークの負荷を軽くする研究に役立つであろう。通信手法によるネットワークの負荷の変化を定量比較することは、本研究においても今後の課題である。

1.1　本論文の構成

本章では、本論文の概要を述べた。

第 2 章では、分散プログラミングの現状と問題点を述べる。

第 3 章では、KL1 言語の基本と既存の KL1 言語処理系について解説する。

第 4 章では、本研究が前提とする分散環境と本研究の目標を設定する。

第 5 章では、DKLIC が使用する通信プロトコルを決定する。

第 6 章では、DKLIC の仕様と設計を述べる。

第 7 章では、DKLIC を実装する。

第 8 章で、DKLIC を評価し、考察を加える。

第 9 章で、本論文をまとめる。

第2章　分散プログラミングの現状と問題点

本章では、分散プログラミングとそのための環境、機構、ツール、言語について述べる。

複数のプロセスが協調処理を行う分散アプリケーションの開発は、逐次アプリケーションの開発に比べて非常に難しい。既に普及している手続き型言語の分散拡張ではネットワーク透過性が不十分であるため、プログラマは逐次アプリケーションを単純に分散拡張することができない。

第 2.1 節、第 2.2 節、第 2.3 節、第 2.4 節、第 2.5 節で、既存の分散プログラミング機構を挙げる。

2.1　プロセス (Processes)

マルチプロセスとは、時分割システム (Time Sharing System) と呼ばれる種類のオペレーティングシステムによって導入された機構である [27]。この種のオペレーティングシステムは複数のプロセスを短時間でスイッチしながら実行することによって、それらのプロセスが同時に実行されているような錯覚を利用者に与える。

例えば、コンピュータ利用者は Web をブラウズしながら、CD を聴きながら、チャットしながら、文書を作成したいであろう。また例えば、銀行の ATM は複数の利用者が複数箇所で同時に利用することができなければならない。

このような複数の副プログラムを全て実行するような巨大な単一のプログラムを作成すれば、望みの処理を提供することはできる。しかし、そのような巨大なプログラムを作成することは難しい。複数のプログラムを独立に作成し、同時に実行することができれば、プログラマは容易に個々のプログラムを作成し、ユーザは複数のプログラムを実行することができる。巨大な一枚岩プログラムを作成する必要はない。

すなわち、**プロセス**とは個々のプログラムの実行の状態である。

2.2 ソケット (Sockets)

複数のプロセスは通信によって同期しながら協調処理を行う。例えば、Web で拾った文書を論文中で引用する時、Web ブラウザと文書作成ソフト間でデータを通信することができれば、同じ内容をユーザが打ち込み直す必要はない。

Unix オペレーティングシステムでは、プロセス間通信はファイル、パイプなど**ストリーム** *(stream)* と呼ばれるインタフェイスを介して行われる。すなわち、プロセスの入出力はストリームのインタフェイスを与えられている。

ソケットはプロセス間通信に用いられるストリームの一種である。サービスを要求するクライアントとサービスを提供するサーバから成る、クライアント/サーバモデルに基づいている。

パイプは共通の親プロセスを持つプロセス間の通信にのみ用いられるが、ソケットはクライアントがホストとポート番号によってサーバを指名することによって接続を確立する。ホストを指定することから分かるように、サーバプロセスはローカルホスト内にある必要はなく、遠隔ホスト内にあっても良い。

ソケットは BSD Unix に由来するが、C 言語のソケットインタフェイスは System V 系 Unix や Windows へも移植された。Java ではソケットインタフェイスが言語レベルで定義されている。

例えば、Wnn などの漢字変換サーバはソケットを通じて接続された日本語入力フロントエンドに対してサービスを提供する。

また例えば、X Window System のサーバは GUI エンジンであり、X を応用するクライアント群の依頼に応じて GUI を生成する。

他にも、FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol), HTTP (Hyper Text Transfer Protocol) など多くのプロトコルがソケットを応用している。

2.3 RPC (Remote Procedure Call)

RPC や Java RMI は、手続きや関数を呼び出す形式で遠隔のサービスを受ける機構である。ソケットへの入力が手続き・関数呼出しの引数として、ソケットからの出力が関数の結果として、透明化される。透明化が不完全であるのは、オブジェクトに対して参照・逆参照の操作を明示するせいである。

NIS (Network Information Service), NFS (Network File System) は RPC を応用している。

2.4 ORB (Object Request Broker)

CORBA (Common ORB Architecture) や Jini は分散オブジェクトを管理し、遠隔のサービス（オブジェクト）に対する要求を仲介する機構である。オブジェクト指向におけるオブジェクトとは状態を持った計算単位であり、プロセスに似た概念である。

RPC との本質的な差は、分散オブジェクトの登録・検索 (Name Service) による capability の取得にある。すなわち、クライアントはサーバがどのホストにあるかなどの位置情報を検索サービスを通じて知ることができる。

デスクトップ環境 GNOME (GNU Network Object Model Environment) は CORBA を応用している。

広義の検索サービスは非常に広い範囲で様々に応用されている。以下に検索サービスの例をいくつか挙げる。

DNS (Domain Name System) はホスト名を name として検索を行う検索サーバを含んでいる。DNS ユーザは望みのホストの IP アドレスを知っている必要はない。DNS ユーザがホスト名を指定するだけで、DNS サーバがそのホスト名を持つホストの IP アドレスを教えてくれる。

IRC (Internet Relay Chat) システムはチャンネルを name として検索を行う検索サーバを含んでいる。IRC ユーザはチャットサーバが実際にはどのホストの何番のポートで接続を受け付けるかを知っている必要はない。IRC ユーザがチャットに加わりたいチャンネルを指定するだけで、検索サーバがそのチャンネルを稼働しているチャットサーバを検索し、自動的に接続してくれる。

Gnutella などの Peer-to-Peer システムは、ファイル名を name として検索を行う検索サーバを含んでいる。Gnutella ユーザは望みのファイルが実際にはどのホストにあるのかを知っている必要はない。Gnutella ユーザが望みのファイル名を指定するだけで、検索サーバがそのファイルを持つホストを検索し、自動的にファイルをダウンロードしてくれる。

2.5 Mobile Agents

モバイルエージェント (Mobile Agents) あるいは単にエージェントは、実行状態を含むプロセスの移動によって通信を透明化する機構である。RPC や ORB との差は、通信されるメッセージが単なるデータではなく自律的に計算・移動を行うプロセスである点にある。

Java Bytecode, Macromedia Flash, Dynamic HTML などが非常に機能の制限されたエージェントであると考えると、オンラインショッピングはエージェントの応用である。

また、Web 上で統計や検索のための情報を自律的に収集する Web Robots は正にエージェントである。

以上で、プロセス、プロセス間通信、RPC、ORB、エージェントについて述べた。これら既存の分散プログラミング機構は C や Java などの手続き型言語に基づいており、参照のネットワーク透過性が不十分である。

次に、分散プログラミングに特徴的ないくつかの問題領域について述べる。

2.6 本質的な並行性 (Concurrency)

分散環境においては、非分散環境においては見られない本質的な並行性がプログラミングの重要な要素となる。

例えば、ある銀行口座に対して複数箇所で同時に振り込み・引き出しが行われることがあるであろう。もし、口座に対して操作を行う端末 (ATM) がこの世に一つしか存在しなければ、物理的制約からその口座に対する操作は自然に順序付けられるであろう。しかし、そのように並行処理を行えない銀行システムは実際には使い物にならない。

このような分散アプリケーションに対する要求は、サービスを分散させながら、しかしデータは共有したいと言う要望から生ずる。ある銀行口座に対する振り込み・引き出し処理は東京でも大阪でも行えるべきである。しかし、その口座の残高は東京でも大阪でも共有されているべきである。東京で振り込んだ金額を大阪で引き出すことができないような銀行システムは使い物にならない。

共有データに対して並行処理が行われる場合、一般にそれらの処理のトランザクション管理が必要である。例えば、残高 10 万円の口座に対して、1 万円の振り込みと 2 万円の引き出しが並行に行われるものとする。差し引き残高は $10+1-2=9$ 万円であるべきである。しかし、振り込み処理が

1. 残高を読み込む 10 万円
2. 振り込む $10+1$ 万円
3. 差し引きを書き込む 11 万円

一方、引き出し処理が

1. 残高を読み込む 10 万円
2. 引き出す $10-2$ 万円
3. 差し引きを書き込む 8 万円

のように行われ、振り込みの各操作と引き出しの各操作が交互にスイッチしながら実行されると、9 万円であるべき差し引き残高は 11 万円か 8 万円のいずれかになる。

これらの処理を正しく行うために、振り込み・引き出し処理はそれぞれ不可分 (atomic) でなければならない。すなわち、振り込み処理が先に始まったら、差し引きを書き込んでからでなければ引き出し処理を始めてはならない。逆もまた同様である。

共有データに対する各処理を不可分にするため、一般に**排他制御**が利用される。各処理はロック、セマフォ、モニタなどと呼ばれる操作権を取得してからでなければ実行を始めることが許されない。これらの操作権は常にただ一つの処理に対してのみ与えられる。実行を終えた処理は操作権を返却する。返却された操作権は、その操作権を予約していた、あるいはその後に操作権を求めてきた他の処理に与えられる。Java 言語の synchronized は排他制御を実現するものである。

プログラムを実行するためのありとあらゆる部分処理を不可分にすれば、プログラムが（少なくとも並行処理に関して）誤っていないことは保証できる。しかし、このようなプログラムは並行プログラムではなく、逐次プログラムに等価である。従って、並行プログラミングにおいては排他制御による不可分な処理を最小限に留めることが望ましい。

しかし、プログラムの排他制御部分をたった一行減らしただけで並行処理が誤りになることは非常に多い。例えば、上記の振り込み処理は次のように不可分であるべきである。

1. 残高に対する操作権を取得する
2. 残高を読み込む 10 万円
3. 振り込む $10 + 1$ 万円
4. 差し引きを書き込む 11 万円
5. 残高に対する操作権を返却する

しかし、あるプログラマが残高を読み込む操作は残高を書き換える操作でないので不可分ではないと考え、排他制御部分の外に出してしまったとする。この場合、

1. 振り込み操作が残高を読み込む 10 万円
2. 引き出し操作が残高に対する操作権を取得する
3. 引き出し操作が残高を読み込む 10 万円
4. 引き出し操作が引き出す $10 - 2$ 万円
5. 引き出し操作が差し引きを書き込む 8 万円
6. 引き出し操作が残高に対する操作権を返却する
7. 振り込み操作が残高に対する操作権を取得する

8. 振り込み操作が振り込む 10 + 1 万円
9. 振り込み操作が差し引きを書き込む 11 万円
10. 振り込み操作が残高に対する操作権を返却する

という誤った処理が行われる可能性がある。

また、複数の処理が複数の共有データに対する操作権を一部ずつ取得し、お互いにもう一部の操作権を待ち続けるデッドロックという現象を避けることにも、プログラマは注意しなければならない。

2.7 無視できない遅延時間 (Latency)

分散環境においては、計算機内部の記憶装置に比較して数千倍、数万倍の通信遅延が起きる。

衛星を通じて地球の反対側と通信することもあるであろう。通信手段が光速度を越えない限り、物理的に距離の遠い分散通信は非常に大きな遅延時間を伴う。

一般に、分散通信の遅延時間は無視できない程度に大きいと考えるべきである。

2.8 部分的な故障 (Fault Tolerance)

分散環境においては、非分散環境に比較してシステムの一部が故障することが多い。例えば、イーサネットケーブルにつまずいて断線することはマザーボードの基盤上の配線が断線することよりも多いであろう。

また例えば、プロバイダを通じてインターネットに接続された小規模の LAN は独自の DNS サーバを稼働していないことが多いであろうが、プロバイダの DNS サーバへの接続を失敗することは大した事故ではない。

分散環境においては、このような部分的故障への対処が必要である。

オペレーティングシステムにおいては、一つのプロセスの故障によってシステム全体が故障すべきではない。同様に、ネットワーク上の一つのホストや通信路の故障によってネットワーク全体が、あるいはネットワークに接続された健全なホストが故障すべきではない。

従って、分散アプリケーションにおいて全てのプロセスは他のプロセスが故障しないことを期待すべきではない。

2.9 保安性 (Security)

分散環境においては、保安性が重視される。

携帯情報端末に住所や暗証番号などの個人情報が満載されていても、その携帯端末を落として他人に利用されたり、その端末からネットワークに個人情報が送信されたりしない限り、それらの個人情報が他人に洩れる心配はない。
しかし、端末にそれらの個人情報を登録する目的は、役所への手続きやオンラインショッピングを簡単に済ませるためであろう。即ち、個人情報はネットワークに送信されることが前提となる。一方で、それらの個人情報が権利のない他人に洩れることがあってはならない。
分散アプリケーションは、本来の目的とする相手に十分な情報を通信しなければならない。一方で、それ以外の相手に個人的な情報を漏らしてはならない。

本章では、プロセス、プロセス間通信、RPC、ORB、エージェントについて述べた。これら既存の分散プログラミング機構は C や Java などの手続き型言語に基づいており、参照のネットワーク透過性が不十分である。また、分散プログラミングに特徴的ないくつかの問題領域について述べた。
次章では、本章で述べた分散プログラミングにおける問題点を解決するために本研究が基礎としている KL1 言語と既存の KL1 言語処理系について述べる。

第3章　分散 KL1 言語の挑戦

前章では、分散プログラミングの現状と問題点について述べた。本章では、それらの問題点を解決するために本研究が基礎としている KL1 言語と既存の KL1 言語処理系について述べる。

3.1　KL1 言語とは

KL1 言語とは、CLP (Concurrent Logic Programming), CCP (Concurrent Constraint Programming), CBC (Constraint-based Concurrency) などと呼ばれるプログラミングパラダイムに基づく言語（計算モデル）族のうちで、最も単純なものである。

KL1 言語は最も単純な並行論理型言語の一つである。述語呼び出しによって並行プロセスを動的生成し、論理変数を通信路として非同期プロセス間通信を行う。論理変数を含むメッセージを通信路に流すことによって、論理ネットワークを動的構成することができる。

ユーザや遠隔プロセスもまた、並行プロセスの一種であると考えられる。従って、並行プロセスと非同期プロセス間通信を用いる KL1 言語のプログラミングスタイルは、分散プロセスによる協調処理の記述に適していると言える。

3.2　KL1 言語の概要

KL1 プログラムは述語の集合から成り、**述語** *(predicate)* は節の集合から成る。**節** *(clause)* は次のような形をしている。

$$Head \text{ :- } Guard \mid Body \text{ .}$$

ここで、*Head* は一階述語論理の原子論理式であり、*Guard* と *Body* は原子論理式のマルチ集合である。*Head* と *Guard* はその節が実行されるための条件を表し、*Body* 内の原子論理式は**述語呼出し**、すなわち**ゴール** *(goal)* を表す。

例えば、銀行口座を表す述語は次のように書ける。

```
account(Request, State) :- Request = [balance(Balance)|Request1] |
  Balance := State,
```

```
  account(Request1, State).
account(Request, State) :- Request = [exchange(Amount)|Request1] |
  State1 := State + Amount,
  account(Request1, State1).
```

この述語は account という名前を持ち、引数の個数 (arity) が二つであるため、account/2 述語と呼ばれる。

State は口座の残高を表す。この口座に対する処理要求は Request へメッセージを書き込むことによって表現する。例えば、balance/1 メッセージは現在の残高を参照する。引数 Balance に現在の残高が返される。また、exchange/1 メッセージはこの口座に対して Amount 円の振り込み（負数なら引き出し）を行う。

exchange/1 メッセージの処理が第 2 章の不可分な振り込み・引き出し処理に相当することに注意せよ。balance/1 メッセージの処理は不可分な振り込み・引き出し処理の一部である読み込み操作のみを行うような処理である。

3.3 言語核に組み込まれた同期と自動排他制御

account/2 述語の振り込み処理は、手続き型のプログラムのように残高を表す変数 State を直接書き換える代わりに、書き換え後の残高を表す変数 State1 を導入している。一般に、**単一代入変数**を持つ純関数型言語や論理型言語で状態を書き換える時には、このように次状態変数を導入する。

State の各状態は口座残高の履歴であり、Request は口座に対する操作の履歴 (journal) であると考えることができる。

KL1 言語では、全てのゴールは並行に実行されるが、*Head* や *Guard* で未定義変数の具体値を読み込むと、その節の実行は具体値が決定するまで中断することになっている。

例えば、

```
  State1 := State + Amount,
```

が実行されるまで State1 の具体値は決定されない。従って、exchange/1 メッセージの直後に他のメッセージが到達しても、exchange/1 メッセージの処理を終えなければ直後のメッセージの処理は始まらない。一方、balance/1 メッセージは残高を書き換えないので、balance/1 メッセージとその直後のメッセージは並行に処理される。

Java 言語では、共有データに対する不可分な処理を synchronized ブロックで囲むことによって排他制御を行う。このような手続き型言語による分散プログラミングは、不可分な処理を見誤ることによって不正な処理を行う危険がある。

一方、KL1 言語において不正な並行処理を行うプログラムはほとんどの場合、State と State1 を取り違えるなどして変数を書き誤ったものである。KL1 言語

では静的解析の技術が非常に発達しており、Kima [9] という解析系はこのような変数の書き誤りを自動的に検出し、修正案を提示することができる。

また、既に具体値の決定した単一代入変数に対して異なる具体値を書き込もうとすると失敗 (unification failure) が起きる。このような動的な失敗は、誤ったプログラムが停止するという最悪の場合の保証であって、プログラマはこれに頼るべきではない。3.5 で述べるモード付けによって静的な排他制御を行うべきである。

3.4 通信路 (Channels)

KL1 言語の変数は単一代入であるだけでなく、first-class の**論理変数**である。すなわち、論理変数は具体値の決定していない未定義状態のまま、**項** *(term)* と呼ばれる KL1 言語のデータ表現木の一部として他のプロセスに渡すことができる。

例えば、balance/1 メッセージの発信者は残高を知らないので、引数 Balance を未定義のまま account/2 述語に渡す。account/2 述語は引数 Balance に残高を書き込むことによって返事をする。Balance のように結果を返すための変数を返信箱 (Reply Boxes) と呼ぶ。

また例えば、account/2 述語へ exchange/1 メッセージを送ることは、引数 Request に具体値 `[exchange(Amount)|Request1]` を書き込むことによって表現するが、このとき続くメッセージ列 Request1 はまだ未定義であろう。Request のように徐々に書き込まれてゆくメッセージ列を**ストリーム** *(stream)* と呼ぶ。

論理変数は通信路である。例えば、次の append/3 述語はリスト X と Y を連結したリスト Z を返す。

```
append(X, Y, Z) :- X = [] |
  Z = Y.
append(X, Y, Z) :- X = [A | X1] |
  Z = [A | Z1],
  append(X1, Y, Z1).
```

クライアントがこの述語を呼び出すと append/3 プロセスが生成される。関数型言語では X の中身が完全に確定した時に Z を全て確定する。しかし、この append/3 プロセスは X の中身が不確定ならば中断し、X の中身が徐々に確定するにつれて Z の中身を前から確定していく。

すなわち、append/3 プロセスは要求駆動型のサーバである。X と Y はクライアントからサーバへの通信路であり、Z はサーバからクライアントへの通信路である。また、X, Y, Z の要素は通信路に流れるメッセージである。

X, Y, Z に流れるメッセージが整数や文字列などの単純なものであれば、Unix 入出力ストリームや RPC で簡単に実装することが可能である。

しかし、KL1 言語では未定義状態の論理変数を含む複雑なメッセージを通信路に流すことによって、新たな通信路を動的生成することが可能である。複雑な通信プロトコルの宣言的記述を許すために、分散 KL1 言語処理系は論理変数の輸出入と分散単一化を実現しなければならない。

3.5 Type / Mode System

本節では、KL1 言語における高度な静的解析の体系について解説する。

型体系は、純関数型言語において研究が盛んである。KL1 言語ではプログラマが変数の型を明示的に宣言することはなく、従って、KL1 プログラムの変数には型がない (type-less) が、プログラムの静的解析による型再構成が可能である。

例えば、account/2 述語の State 引数は、

```
State1 := State + Amount,
```

から分かるように数値型を持つ。また、Request 引数は

```
account(Request, State) :- Request = [exchange(Amount)|Request1] |
```

から分かるように、コンス（リストセル）型を持つ。

このように変数が特定の型の具体値のみをとることは良いプログラムの性質であり、型付き (well-typed) と言われる。前述の自動誤り修正系 Kima はプログラムの型付きでない部分を報告し、修正案を提示する。

モード体系は、変数に対する読み書きのモード（方向）に関する体系である。

例えば、balance/1 メッセージの引数 Balance の書き手は balance/1 メッセージの発信者でなく、account/2 述語である。従って、account/2 述語にとって Balance のモードは書き込み (out) である。逆に、balance/1 メッセージの発信者にとって Balance のモードは読み込み (in) である。

このように同一の変数に対するモードは書き手と読み手の間で逆転する。変数への書き手がただ一人であることは良いプログラムの性質であり、モード付き (well-moded) と言われる。自動誤り修正系 Kima はプログラムのモード付きでない部分を報告し、修正案を提示する。

静的に排他制御されていないモードなしのプログラムの多くは、それと等価な計算を行なうモード付きの（静的に排他制御された）プログラムに書き換えることができる。モードなしのプログラムを等価なモード付きのプログラムに書き換える方法については、付録 A.1 を参照せよ。

3.6 KLIC 処理系

KLIC 処理系は既存の KL1 言語処理系の一つである。

KLIC 処理系は同一の機種、オペレーティングシステムを持つ複数のホストで動作する PVM (Parallel Virtual Machine) 上で並列実行を行う機能を持っている。しかし、分散環境においてはネットワークを構成する個々の機械に同一の機種、オペレーティングシステムを強制することは非現実的である。

KLIC 処理系の並列実装におけるホスト間通信は、変数の読み手が書き手へ具体値を要求するプル型で行われる。プル型の lazy なメッセージ通信は、読み手がいない変数のためのメッセージが不要であり、通信量が最適であるという性質を持つ。反面、プル型では一つの変数の読み込みに対して往復で二つのメッセージを必要とする。この性質は、通信遅延の大きな分散環境においては許容できない性質である。

KLIC 処理系の分散機能は、変数を含まない基底項を入出力できる手続き型のソケットのみである。従って、本研究が目標とする宣言型分散プログラミングのためには、変数を含む項を入出力できる宣言型のソケットを実装する必要がある。

3.7 関連研究

並行論理型言語として、KL1 言語の他に Concurrent Prolog [21], Janus [22], DRL [23] などがある。

Janus ではメッセージ `^/1` によってプログラマがモードを明示する。その点を除けば Janus は KL1 言語に非常に近く、Concurrent Prolog や KL1 と同じ言語の異なる実装（方言）と考えて良い。

Oz [25] は並行制約に基づくマルチパラダイム言語である。Oz は静的解析を目指しておらず、単一代入変数だけではプログラミングに不十分であると主張している。[24], [26] では、手続き型言語に論理変数を導入する方法を論じている。Oz の分散実装である Mozart は KLIC 処理系の並列実装を参考にしている。

以上で、宣言型分散プログラミングを可能にする KL1 言語の現状と既存の KL1 言語処理系の不足な点について述べた。次章では、宣言型分散プログラミングを可能にするために既存の KL1 言語処理系に欠けている機能を分析する。また、それらの機能を実装するに当たっての目標を定める。

第4章　DKLIC 処理系の目標

前章では、宣言型分散プログラミングを可能にする KL1 言語の特徴と既存の KL1 言語処理系の不足な点について述べた。本章では、宣言型分散プログラミングを可能にするために KL1 言語が備えるべき機能を分析する。また、その機能を実現するための目標を設定する。

4.1 本研究が前提とする分散環境

DKLIC 処理系が前提とする分散環境は次の通りである。

1. ネットワークに接続されたホストが動的に増減し、全順序を付けるなどの全域的な把握は困難である。また、全てのホストで同じ時刻を示すような全域的な時計は存在しない。

   KLIC の並列実装はホストに全順序が付けられていることを前提としているため、広域分散環境にそのまま適用することはできない。

2. 各ホストの機種やオペレーティングシステムは統一されていない。

   KLIC の並列実装は全てのホストで機械語コードを共有するため、非均質な分散環境にそのまま適用することはできない。

3. ホスト間の通信は予測できない遅延時間を要する。

   KLIC 処理系の並列実装は変数の具体値を lazy に受信するプル型のメッセージ通信を行うが、一つの変数の読み込みに往復二つのメッセージが必要なプル型通信は通信遅延の大きな分散環境において許容できるものではない。

4. 各ホスト間は TCP/IP 接続されており、パケットの欠損や追い越しが起きることはない。

4.2 DKLIC 処理系の構成

DKLIC 処理系は、宣言型分散プログラミングを可能にする分散 KL1 言語処理系である。

本研究が基盤とするインターネットでは、歴史的に実装が重視される。また、単純さは良いソフトウェアの性質である。このことから、DKLIC 処理系は単純な実装を重視する。DKLIC の実装を単純にするため KLIC を下位処理系とし、KL1 言語を実装言語とする。

KLIC 処理系の分散機能は手続き型ソケットのみであるため、DKLIC 処理系を実現するためには宣言型ソケットを実装する必要がある。宣言型ソケットを実現している既存の処理系は存在しない。

DKLIC 処理系は変数の具体値を eager に送信するプッシュ型のホスト間メッセージ通信を行なう。プッシュ型の実装を提供すれば、プル型のホスト間通信をシミュレートする応用も記述できるという理由もある。

実装を単純にするため、DKLIC 処理系の機能はいくつかの階層に分割して提供する。個々の機能を提供する各階層、各サーバはできる限り単純に実装し、Unix のコマンドのように組み合わせることによって包括的な機能を提供するシステムを構成する。

4.3 DKLIC 全体の仕様

DKLIC 処理系およびその応用は次のような階層を成す。

1. 接続サーバ dklicio は宣言型ソケットを確立する bind/2, connect/2 述語を提供する。

2. 検索サーバ naming (ORB) は次の二つの述語あるいはメッセージを処理する。register はユーザ定義サーバを登録する。lookup は登録されたサーバを検索する。

3. エージェント空間 exec (Mobile Agents / Fault Tolerance) は KL1 言語によって記述されたモバイルエージェントを実行、制御する機能を提供する。

4. ユーザ定義サーバ. 下位の接続サーバや検索サーバなどを利用して、分散 KL1 プログラマは独自のサーバを定義することができる。

5. アプリケーション. 下位の接続サーバ、検索サーバ、ユーザ定義サーバなどを利用して、KL1 プログラマは分散アプリケーションを記述することができる。

4.4 本研究の目標と範囲

本研究においては、例外処理は目標としない。

本研究では DKLIC 処理系の最下位サーバである dklicio を実装する。検索サーバその他の上位サーバは本研究の目標ではなく、応用例とする。

DKLIC 処理系の基本サーバ dklicio は、以下のような宣言型分散プログラミングを可能にする。

```
name(NS, Table) :- NS = [register(Name,Server) | NS1] |
  generic:new(merge, Clients, Server),
  Table1 = [(Name,Clients) | Table],
  name(NS1, Table1).
name(NS, Table) :- NS = [lookup(Name,Result) | NS1] |
  lookup(Name, Result, Table, Table1),
  name(NS1, Table1).

lookup(Name, Result, Table, Table_) :- Table = [] |
  Result = not_found(Name),
  Table_ = Table.
lookup(Name, Result, Table, Table_) :- Table = [(N,S)|Table1], N=Name |
  S = {Client, S1},
  Result = found(Client),
  Table_ = [(N,S1) | Table1].
otherwise.
lookup(Name, Result, Table, Table_) :- Table = [(N,S)|Table1] |
  Table_ = [(N,S) | Table1_],
  lookup(Name, Result, Table1, Table1_).
```

この name サーバは、register/2 メッセージによってサーバへの通信路 Server に名前 Name を付けて表に登録する。また、lookup/2 メッセージによって指定された Name を表の中で検索する。Name が見つかった場合、Name と名付けられたサーバへの通信路 Server へとマージされる通信路のうち一本を返す。

この返された通信路 Client に対してクライアントはサーバへの要求メッセージを書き込んで良い。複数のクライアントからサーバが並行に受け取るメッセージ列は、一本の通信路 Server にマージされている。

クライアント、name サーバ、サーバがそれぞれホスト A, B, C にあるとき、DKLIC 処理系の既存の試作ではクライアントから name サーバを通じてサーバへ接続された通信路に流れるメッセージは、name サーバが中継を終了してもホスト B を中継して通信されていた。

本研究では、未定義変数の直結や廃棄を検出することによって、このような中継を排除することを目標とする。

本章では、宣言型分散プログラミングを可能にするために KL1 言語が備える

べき機能を分析した。また、その機能を実現するための目標を設定した。次章では、DKLIC 処理系の通信プロトコル DKLICP を解説する。

第5章　プロトコル DKLICP

前章では、KL1 言語による分散プログラミングを実現するために未定義変数を含む KL1 項を送受信する必要があることを明らかにし、DKLIC 処理系の目標を設定した。本章では、論理変数を含む KL1 項を送受信するために dklicio が使用する通信プロトコル DKLICP について解説する。

次に DKLICP 言語の形式化の階層を挙げる。

1. 正則文法 $lex : String \to Token^*$. KL1 言語の字句解析と同一である
2. 文脈自由文法 $parse : Token^* \to Term^*$. KL1 言語の構文解析と同一である
3. 文脈依存文法 $unmarshal =_{\mathrm{def}} marshal^{-1} : (String \to)Term^*$. 分散論理変数・輸出入表が定義する
4. 意味論.（ユーザ定義）サーバによって定義される

DKLIC 処理系の実装を単純にするため、$lex, parse$ を KL1 言語と共有することによって、KLIC 処理系の既存パーザをそのまま利用する。

他言語との inter-operability のために正則文法と文脈自由文法を XML 言語に対応させる例については、付録 A.2 を参照せよ。

5.1　二者通信のプロトコル

1. まず、クライアントがサーバへ文字列 “DKLICP 2002-01 *Host*:*Port*\n” を送信する。ここで、*Host*:*Port* はクライアントプロセスを実行する DKLICP 対応サーバの位置情報である。

   この文字列を受け取ったサーバはそのクライアントへ文字列 “DKLICP 2002-01 server\n” を送信し、以下の規則に従って KL1 項の送受信を開始して良い。

   サーバからの返答文字列を受け取ったクライアントは以下の規則に従って KL1 項の送受信を開始して良い。

   ただし、個々の KL1 項はピリオドと LineFeed から成る文字列 “.\n” によって終端する。

2. 変数以外の項は、KL1 プログラムにおいてその項を表現する文字列そのままで表現される。コンス、ファンクタ、ベクタなどの複合値は、その要素に対して再帰的に本規則を適用する。

3. 変数 X は“ *_ID* ”によって表現される。ここで *_ID* は変数 X に割り当てられた識別子である。

   変数 X にまだ識別子が割り当てられていない場合、この二者通信の間で既存の識別子と重複しない新たな識別子を割り当てる。識別子の割り当てられた変数を分散論理変数と称する。

4. (a) ホスト A 内で分散論理変数 X に具体値 *Term* が書き込まれたら、具体化メッセージ“ *_ID* = *Term*”を送信する。ここで *_ID* は変数 X に割り当てられた識別子である。

   (b) ホスト A 内で分散論理変数 X と Y が直結されたら、直結メッセージ“ *_ID* = *_ID1* ”を送信する。ここで *_ID* , *_ID1* はそれぞれ変数 X, Y に割り当てられた識別子である。また、X は out モードの変数とする。

   in モードの変数同士を直結するのは、読み込みモードの変数に制約を課してある種の書き込みを行うことになるので、モード付きであるという前提に反する。

   従って、直結された二つの変数のうち少なくとも一つは out モードである。X が out モードとなるように X, Y を選び、メッセージを構成せよ。

   (c) ホスト内で分散論理変数 X への参照がなくなったら、削除メッセージ“ *_ID* =Removed”を送信する。ここで *_ID* は変数 X に割り当てられた識別子である。

   これらの具体化、直結、削除メッセージを受信したホスト B は、受信確認として“ *_ID* =Removed” を送信し返す。

   _ID の割当て権がホスト A にある場合、ホスト A はホスト B からの受信確認を受信した後に *_ID* を別の変数に割り当て直して良い。 *_ID* の割当権がホスト B にある場合、ホスト B はメッセージに対する受信確認を送信した後に *_ID* を別の変数に再利用して良い。

5.2 分散論理変数のライフサイクル

分散論理変数のライフサイクルは次の三段階から成る。

1. open. 論理変数に *_ID* を割り当て、輸出（そして輸入）する。

2. use. 分散論理変数を項の一部として送受信する。

3. close. 通信路を閉じ、 $_ID$ を回収する。

具体化メッセージ、直結メッセージ、削除メッセージを受信すると、通信路を閉じて $_ID$ を回収する処理が行われる。このため、これらのメッセージを $CloseMessage(id)$と呼ぶ。

また、 $_ID$ の割り当てられていない非分散論理変数を輸出するとき、通信路を開いて $_ID$ を割り当てる処理が行われる。このように新たな通信路を開くメッセージを$OpenMessage(id)$と呼ぶ。

$OpenMessage(id)$の送信者を $opener(id)$, 受信者を $openee(id)$と呼び、$CloseMessage(id)$の送信者を $closer(id)$, 受信者を $closee(id)$と呼ぶ。

分散論理変数の状態遷移は次のようなものである。

1. $\emptyset(id)$. 利用されたことがない状態である
   (a) $\emptyset(id) \rightarrow opening(id)$
   輸出していない id の $OpenMessage(id)$を遠隔ホストへ輸出する
   (b) $\emptyset(id) \rightarrow opened(id)$
   $OpenMessage(id)$を遠隔ホストから輸入する

2. $opening(id)$. $OpenMessage(id)$を輸出したが、遠隔ホストへ輸入されたことを確認していない状態である
   (a) $opening(id) \rightarrow opened(id)$
   輸出した id を遠隔ホストから受信する
   (b) $opening(id) \rightarrow closing(id)$
   輸出した id の $CloseMessage(id)$を遠隔ホストへ送信する
   (c) $opening(id) \rightarrow closed(id)$
   輸出した id の $CloseMessage(id)$を遠隔ホストから受信する。受信確認を送信する必要がある

3. $opened(id)$. $OpenMessage(id)$を遠隔ホストから輸入した状態である
   (a) $opened(id) \rightarrow closing(id)$
   $CloseMessage(id)$を遠隔ホストへ送信する
   (b) $opened(id) \rightarrow closed(id)$
   $CloseMessage(id)$を遠隔ホストから受信する。受信確認を送信する必要がある

4. $closing(id)$. $CloseMessage(id)$を送信したが、遠隔ホストからの受信確認を受け取っていない状態である

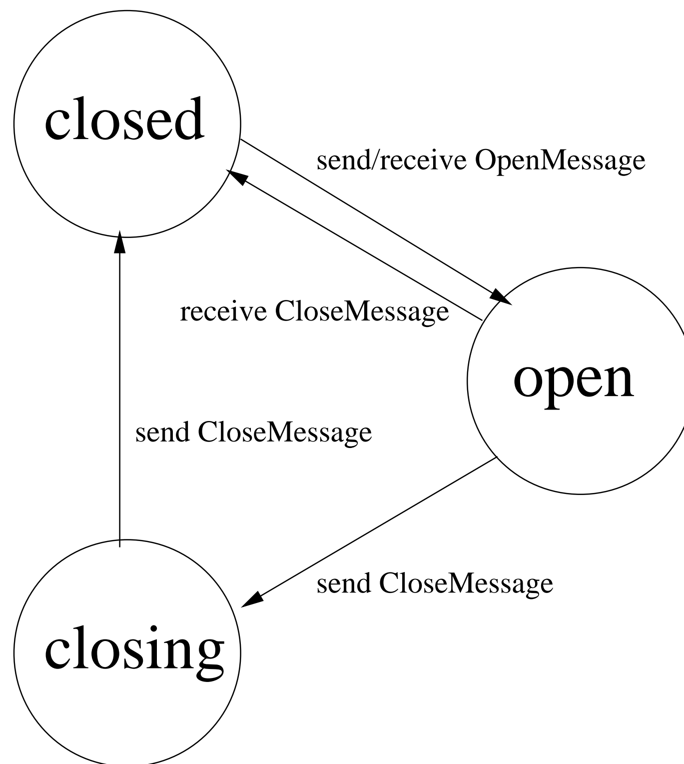
(a) $closing(id) \rightarrow closed(id)$
$CloseMessage(id)$を遠隔ホストから受信する。受信確認を送信する必要はない

5. $closed(id)$. $CloseMessage(id)$を遠隔ホストから受信した状態である

   (a) $closed(id) \rightarrow opening(id)$
   $OpenMessage(id)$を遠隔ホストへ再輸出する

   (b) $closed(id) \rightarrow opened(id)$
   $OpenMessage(id)$を遠隔ホストから再輸入する

遷移グラフを簡約化すると $\emptyset(id) = closed(id)$ である。また、$opener(id)$の ID 空間は分離してあるので、$opening(id)$と $opened(id)$を区別する必要はない。



5.3 誰が ID を再利用するか

1. $closee(id)$が id を再利用する場合、$CloseMessage(id)$の受信確認を送り出した直後に id を再利用した $OpenMessage(id)$を送信することができる。

   一方、$closer(id)$が id を再利用する場合、通信相手からの $CloseMessage(id)$の受信確認を受け取るまで id を再利用することはできない。

2. *openee*(*id*)が id を再利用する場合、id の利用権が交互に移動する。

3. *opener*(*id*)が id を再利用する場合、ID 空間の利用権を持っている側が常に id を再利用する。

   openee(*id*)が再利用する場合と異なり、ID 空間が動的に変化することはない。

   再利用以外の場合と整合しているので、この案を採用する。

closee(*id*)が id を再利用する場合、受信確認を待たずに再利用を始めることができる。しかし、次のようにクライアントからサーバへの典型的な要求ストリームを考えよう。

```
Request0 = [request0(Answer0) | Request1]
Request1 = [request1(Answer1) | Request2]
           :
```

このような通信パターンを続ける限り、サーバは決して変数を open せず、サーバの ID 空間は単調増加する。一方、クライアントは変数を open し続け、クライアントの ID 空間は単調減少し、やがて枯渇するであろう。変数の ID を資源と見なす観点からは、*opener*(*id*)が id を再利用する方式が適切である。

5.4 三者通信のプロトコル

ホスト A–B 間に通信路 K、ホスト B–C 間に通信路 L が開いている時、K と L の間に直接の関連がなければ、ホスト A–B 間、B–C 間の通信はそれぞれ独立な二者通信である。

しかし、ホスト B 内で通信路 K, L が直結されると、直結 K=L の実行をホスト A や C に知らせる必要がある場合がある。

1. ホスト B において通信路 K, L がいずれも in モードである時、in モードの変数同士を直結するのは、読み込みモードの変数に制約を課してある種の書き込みを行うことになるので、モード付きであるという前提に反する。

2. ホスト B において通信路 K, L がいずれも out モードである時、ホスト A や C に直結を報告する必要はない。K=L はホスト B からのマルチキャストであり、書き手 B が読み手 A, C に具体値を直接送信する責任を持つ。

3. ホスト B において通信路 K が in モード、通信路 L が out モードである時、ホスト B はホスト A へ間接輸出メッセージ “ *_ID = _ID1 @C*” を送信する。ここで *_ID* は A–B 間で K に割り当てられた識別子である。また、*_ID1* は B–C 間で L に割り当てられた識別子である。A–B 間で通信される

メッセージ内の変数名では、自明な “@AB” を省略していると考えて良い。“ _ID1 @C” では “@B” を省略している。このとき、ホスト B において通信路 L は間接輸出された ($exporting$) 状態へ遷移する。

(a) 間接輸出メッセージを受信したホスト A はホスト C へ、間接輸出確認メッセージ “ _ID1 @B= _ID2 ” を送信する。ここで _ID2 は、論理変数 K のホスト A–C 間における新たな通信路 K' の識別子である。ただし、ホスト A 内で論理変数 K に具体値 $Term$ が既に書き込まれていた場合、間接輸出確認メッセージ “ _ID1 @B=$Term$” を送信すれば良く、新たに K' を open する必要はない。

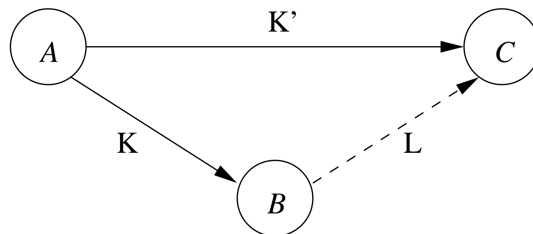
   DKLICP のソケット確立 (negotiation) 時に、クライアントが $Host$:$Port$ を自己申告するのは、新たな通信路 K' を確立する時に必要であることがあるためである。二者通信では、ソケットの確立時にクライアントが $Host$:$Port$ を自己申告する必要はない。

(b) 間接輸出確認メッセージを受信したホスト C はホスト B へ間接輸出済みメッセージ “ _ID1 =Exported” を送信する。

(c) 間接輸出済みメッセージを受信したホスト B は $exporting$ 状態の通信路 L を閉じる ($closed(id)$)。$exporting$ 状態の通信路 L は、間接輸出済みメッセージ以外の$CloseMessage(id)$によって $closed(id)$状態へ遷移してはならない。なぜなら、A から C へ間接輸出確認メッセージが届く前に _ID1 が他の変数に再利用されてはならないからである。

(d) $exporting$ 状態の通信路 L に対して、間接輸出メッセージ “ _ID1 = _ID3 @D” を受信したら、ホスト B はホスト C へ間接輸出不許可メッセージ “ _ID1 =Exporting” を送信する。間接輸出不許可メッセージを受信したホスト C は、ホスト A からの間接輸出確認メッセージを待ち、間接輸出メッセージ “ _ID2 = _ID3 @D” を送信し直す。

これらのメッセージ通信を終えると通信路 K=L は、ホスト A からのマルチキャストである K=K' へと接続し直されている。



本プロトコルは、プッシュ型の通信を前提として設計した。すなわち、通信路の直結によって通信の中継を終了し、通信路が接続し直されることを目的とした。

マルチキャストが行われる場合、できる限り書き手から読み手へ直接、中継なしに通信が行われるべきである。なぜなら、分散環境においては通信遅延が無視できないからである。

従って、中継ノードにおいて入力と出力の直結が行なわれた場合、その直結は、まず書き手側に報告されるべきである。本プロトコルはこの規則に従っている。

本プロトコルの正当性の検証は今後の課題である。

以上で、分散論理変数を含む KL1 項を送受信するプロトコルについて解説した。次章では、DKLIC 処理系の最下位サーバ dklicio の設計を行う。

第6章　dklicio の設計

前章では、dklicio の通信プロトコル DKLICP を解説した。本章では、DKLIC 処理系の最下位サーバである dklicio の設計について述べる。

dklicio は次の三つの階層から成る。

1. 分散論理変数 distributed
2. 変数表 variable_table(⇔ ID manager)
3. 宣言型変数入出力ソケット io

6.1 分散論理変数 distributed

分散論理変数は具体化、直結、廃棄されたことを検出する必要がある。

変数表のエントリとなる分散論理変数は、次のようなものである。

$Dist =_{\mathrm{def}} ID \times Var \ni (id, var)$

概念的には次のようなクラス distributed を作ることになる。

```
d(ID, X, Report) :- wait(X) |
  Report = [bound(ID, X)].
alternatively.
d(ID, X, Report) :- fused(X) |
  Report = [fused(ID, X) | Report1],
  d(ID, X, Report1).
d(ID, X, Report) :- garbage(X) |
  Report = [removed(ID)].
```

すなわち、具体化、直結、廃棄されたことを検出し、その旨を変数表に報告する必要がある。

KLIC 処理系は次のメソッドによってクラス distributed を操作する。

1. コンストラクタ　$+ID$, ?Report
2. イベント fuse ?Any
   ローカルホスト内で distributed が単一化されたことを示すイベントである。

ローカルホスト内の制約記憶（単一化の履歴）を $constraints$ とすると、

$$\text{fuse} : Dist \times Constraints \times Term \rightarrow$$
$$\quad Constraints \times \{bound/2, fused/2\}$$
$$\text{fuse}((id, var), constraints, term) =$$
$$\quad (constraints \cup \{var = term\},$$
$$\quad message(id, term))$$
$$\quad \text{where}$$
$$\quad message = bound \quad \text{if } term \in Bound$$
$$\quad message = fused \quad \text{if } term \notin Bound$$

KLIC 処理系で未定義変数同士の直結を検出するには、その未定義変数がジェネレータオブジェクトとして実装されている必要がある。従って、distributed 同士の直結を検出するために、distributed をジェネレータオブジェクトとして実装する。

3. イベント gc ($-$Address)
ローカルホスト内で KLIC 処理系のコピー GC が行われ、ユーザプロセスからこのオブジェクトへの参照が生存しているので、このオブジェクトを新領域へコピーする必要があることを示すイベントである。

   変数表からこのオブジェクトへの参照は「生存していない」ので、それ以外の参照が生存していない場合、このオブジェクトは旧領域へ取り残され、廃棄される。

6.2 ID manager (Name Server)

Name Server は概念的に様々なレベルで現れる。例えば、

1. DNS はホスト名を IP アドレスに対応付ける。
2. ソケットはポート番号をサーバに対応付ける。
3. ID manager は ID を分散論理変数に対応付ける。
4. java.rmi.Naming (rmiregistry) はインタフェイスをサーバオブジェクトに対応付ける。
5. ORB はサービスを分散オブジェクトに対応付ける。

DNS, ORB は実体に、異なる名前空間における別名を付ける。ユーザは実体の本名を知る必要がなく、別名によって実体を特定することができる。

ソケット, ID manager, Java RMI は実体に広域名を付けることによって、遠隔ユーザが実体にアクセスすることを可能にする。

ここで作る Name Server は次のようなものである。
$NS : ID \rightarrow \{free, used\}$
この NS に対しては次の二つの操作を行う。

$$
\begin{aligned}
get &: NS \rightarrow ID \times NS \\
release &: NS \times ID \rightarrow NS
\end{aligned}
$$

$get(ns) = (id, ns \setminus \{(id, free)\} \cup \{(id, used)\})$
$release(ns, id) = ns \setminus \{(id, used)\} \cup \{(id, free)\}$
NS の機能は変数表（配列）に含めることが可能であるが、新しい ID を $O(1)$ で取得するために未使用 ID をリストで管理する。リストの扱いは C 言語より KL1 言語の方が得意としているので、NS は KL1 言語によって記述し、C 言語によって記述する変数表オブジェクトとは別プロセスとする。また、ジェネリックオブジェクトよりも KL1 プロセスの方が、他の KL1 プロセスからのアクセスが容易であるという理由もある。

```
ns(NS, Free) :- NS = [get(ID) | NS1] |
  Free = [ID | Free1],
  ns(NS1, Free1).
ns(NS, Free) :- NS = [release(ID) | NS1] |
  Free1 = [ID | Free],
  ns(NS1, Free1).
```

概念的には上のような NS を作るが、ID 空間が実際には有限であることを考慮する必要がある。
例えば、未使用 ID がない時に get されたら変数表を広げる。変数表が縮んだら未使用 ID を再計算する。
変数表（配列）のインデクスを表す整数を ID として使用する。

6.3 変数表オブジェクト variable_table

変数表オブジェクトは次のようなものである。
$Table =_{\mathrm{def}} 2^{Dist} \ni \{(id, var)\}$

$$
\begin{aligned}
&\forall i, j; (id_i, var_i), (id_j, var_j) \in table \Rightarrow \\
&\qquad (id_i = id_j \Leftrightarrow var_i = var_j)
\end{aligned}
\tag{6.1}
$$

論理変数同士の同一性を判定することが可能であることを前提とする。同一の論理変数が同一の変数表において複数の異なるエントリを持つことはない (6.1)。

$$
\begin{aligned}
&State = Table \times NS \\
&\forall id, ns, table; \\
&\quad (ns(id) = used \Leftrightarrow (id, var) \in table) \wedge \\
&\quad (ns(id) = free \Leftrightarrow (id, var) \notin table)
\end{aligned} \tag{6.2}
$$

変数表オブジェクトは NS と連動する。変数表オブジェクトは NS の内部に隠蔽し、他のプロセスから直接操作されないようにする (6.2)。

NS は次のメソッドによって変数表オブジェクトを操作する。

1. コンストラクタ +Length, ?Report

2. イベント gc (−Length)
ローカルホスト内で GC が行われていることを示すイベント。変数表オブジェクトは自分自身を新領域へコピーするが、エントリ distributed をコピーしない。GC 終了直後に廃品として回収された distributed の後始末をする。

3. io からのメッセージ add $-id$, ?Var, (−Length)
io が変数表オブジェクトに登録されていない論理変数を輸出したい時、または io が遠隔ホストから $OpenMessage(id)$を受け取った時に発するメッセージである。
$add : State \times Var \rightarrow ID \times State$
$add(table, ns, var) =$
$\quad (id, table \cup \{id\}, ns_1)$
$\quad$ where $get(ns) = (id, ns_1)$

4. io からのメッセージ remove $+ID$
io が遠隔ホストから‘$_ID$=Removed’を受け取った時に発するメッセージである。
$remove : State \times ID \rightarrow State$
$remove(table, ns, id) =$
$\quad (table \setminus \{(id, var)\}, release(ns, id))$

5. io からのメッセージ getID $-ID$, ?Var
io が変数表オブジェクトに登録済みの分散論理変数を遠隔ホストに送信したい時、ローカルホスト内の論理変数に対応する ID を要求するメッセージである。
$getID : State \times Var \rightarrow ID$
$getID(table, ns, var) = id$
$\quad$ where $(id, var) \in table$

6. io からのメッセージ get $+ID$, ?Var
io が遠隔ホストから‘$_ID$’を受け取った時に、ID に対応するローカルホスト内の論理変数を要求するメッセージである。
$get : State \times ID \rightarrow Var \times State$
$get(table, ns, id) = (var, table, ns)$
  where $(id, var) \in table$

7. io からのメッセージ set $+ID$, +Bound
io が遠隔ホストから‘$_ID{=}Bound$’を受け取った時に発するメッセージである。
$set : State \times Constraints \times ID \times Bound \rightarrow$
  $State \times Constraints$
$set(table, ns, constraints, id, bound) =$
  $(table \setminus \{(id, var)\}, release(ns, id),$
  $constraints \cup \{var = bound\})$

8. io からのメッセージ fuse $+ID$, $+ID1$
io が遠隔ホストから‘$_ID{=}_ID1$’を受け取った時に発するメッセージである。
fuse : $State \times ID \times ID \rightarrow State$
fuse$(table, ns, id, id_1) =$
  $(table \setminus \{(id, var), (id_1, var_1)\} \cup$
  $\{(id_1, var)\}, release(ns, id))$

9. distributed からのメッセージ bound $+ID$, +Bound
distributed が具体化された時に発するメッセージである。
bound : $State \times ID \times Bound \rightarrow State$
bound$(table, ns, id, bound) =$
  $remove(table, ns, id)$

10. distributed からのメッセージ fused $+ID$, +Var
distributed が他の distributed と単一化された時に発するメッセージである。
$fused : State \times ID \times Var \rightarrow State$
$fused(table, ns, id, var)$
$= (table, ns)$ if $id < getID(var)$
$= remove(table, ns, id)$ if $id > getID(var)$

11. distributed からのメッセージ removed ID
distributed が KLIC の GC フェイズ中に新領域へコピーされなかった時、廃棄される直前に発するメッセージである。

```
vt(VT, T, NS) :- VT = [add(ID,X) | VT1] |
  NS = [get(ID) | NS1],
  T1 = [(ID,X) | T],
  vt(VT1, T1, NS1).
vt(VT, T, NS) :- VT = [remove(ID) | VT1] |
  remove(T, ID, T1),
  NS = [release(ID) | NS1],
  vt(VT1, T1, NS1).
vt(VT, T, NS) :- VT = [get(ID,X) | VT1] |
  get(T, ID, X, T1),
  vt(VT1, T1, NS).
vt(VT, T, NS) :- VT = [set(ID,Bound) | VT1] |
  set(T, ID, Bound, T1),
  vt(VT1, T1, NS).
vt(VT, T, NS) :- VT = [fuse(ID,ID1) | VT1] |
  fuse(T, ID, ID1, T1),
  vt([remove(ID) | VT1], T1, NS).
vt(VT, T, NS) :- VT = [bound(ID,X) | VT1] |
  remote(set(ID,X)),
  vt([remove(ID)|VT1], T, NS).
vt(VT, T, NS) :- VT = [fused(ID,ID1) | VT1] |
  remote(fuse(ID,ID1)),
  vt([remove((ID)) | VT1], T, NS).
```

概念的には上のような変数表オブジェクトを作りたい。

NS に ID 空間が枯渇したと言われた場合、自分自身を広げる。自分自身を縮めた場合、NS に ID 空間を縮めろと言う。

以上で、dklicio の設計を終える。次章では、dklicio の実装を行う。

第7章　dklicio の実装

前章では、DKLIC 処理系の最下位サーバである dklicio の設計を行った。本章では、dklicio の実装について解説する。

7.1　主な開発環境

本研究で主に開発を行なった環境は次のようなものである。

1. Intel Pentium III 800EB
2. PC–AT 互換機
3. Debian GNU/Linux 2.2
4. KLIC–3.003 with patch [6]

7.2　分散論理変数オブジェクト

分散論理変数は、ローカルホスト内の参照がなくなったことを検出できる必要がある。また、未定義変数同士の直結を検出する必要がある。

KLIC 処理系で未定義変数を表現するオブジェクトは純粋未定義変数と中断原因変数がある。中断原因変数は中断ゴールやコンシューマオブジェクトがフックされた中断構造かジェネレータオブジェクトに分類される。

1. 純粋未定義変数. まだ一度も具体値を読み込まれたことがない未定義変数である。KLIC 処理系においては自分自身を指すポインタとして表現される
2. 中断原因変数 (susp). この変数の具体値を読み込んで中断したオブジェクトがフック (hook) されている変数である
   (a) 中断構造. 一つの中断構造には複数の中断ゴールやコンシューマオブジェクトをフックすることができる
   (b) ジェネレータオブジェクト. ジェネレータオブジェクトは常に自分自身のみをフックしている。中断構造と直結されると、具体値や中断構

造を生成する処理が行われた後、KLIC 処理系がその結果を直結相手である中断構造に書き込む

このうち、他の未定義変数との直結を検出できるのはジェネレータオブジェクトのみである。中断構造が他の未定義変数と直結されると、KLIC 処理系は中断原因変数のフック群をマージし、直結を報告しない。従って、分散論理変数はジェネレータオブジェクトとして実装する必要がある。

分散論理変数を表現するジェネレータオブジェクト fusesusp は次の三つのメンバを持つ。

1. id. この fusesusp の識別子である。識別子の内容は fusesusp のモードによって異なる。

2. repo. この fusesusp が具体化、直結、廃棄されたら、その旨を識別子と共に報告する。報告先は fusesusp のモードによって異なる。

3. susp. この fusesusp の具体値を読み込んで中断した中断ゴールやコンシューマオブジェクトをフックしている中断原因変数である。

fusesusp はジェネレータオブジェクトであるので、具体化、直結、削除されたことを検出することができる。

1. この fusesusp に具体値 *Term* が書き込まれると、fusesusp は susp に *Term* を書き込み、repo に具体化 bound(id, *Term*) を報告する。

2. (a) fusesusp 以外の変数 X と直結されると、fusesusp は susp と X を直結する。すると KLIC 処理系が susp と X のフックをマージする。

   (b) 他の fusesusp と直結されると、互いの susp を直結する。すると KLIC 処理系が双方のフックをマージする。

fusesusp は in/out いずれかのモードを持つ。

1. fusesusp(in) は in モードの分散論理変数を表す。fusesusp(in) のメンバは次のような意味を持つ。

   id. *ID@Host* という形の大域識別子とするが、C 言語によって記述する fusesusp に不要な知識を持たせないため、fusesusp は id の内部形式について何ら仮定を置かないようにする。

   repo. 変数表への報告ストリームである。

   fusesusp(in) は他の fusesusp(in) と直結されることはない。なぜなら、DKLIC 処理系はモード付きのアプリケーションを前提としているからである。

   fusesusp(out) と直結されると、fusesusp(in) は fusesusp(out) の repo のフック bindhook:ID@Host 毎に repo に fused(id, ID@Host) を報告する。その

後、fusesusp(out) の repo に id を書き込む。これは fusesusp(out) が id と直結されたことの報告であるが、ジェネリックオブジェクト内でファンクタを構築するのが面倒であるので C 言語による記述部分を減らすために簡略化した。この「直結報告」を受け取るのは fusesusp(out) の bindhook であるので、bindhook がこのメッセージを正しく解釈する限り問題は起きない。

2. fusesusp(out) は out モードの分散論理変数に対応付けられたローカルホスト内の論理変数を表す。fusesusp(out) のメンバは次のような意味を持つ。

   id. id は他の fusesusp と直結された時に利用するものであるが、fusesusp(out) 同士の直結においては識別は不要であるので、全ての fusesusp(out) の識別子は out とする。

   repo. この fusesusp の具体化を待つ bindhook への報告ストリームである。

   fusesusp(out) は fusesusp(in) と直結されると、自分と直結相手を逆転して前述の in–out 直結処理を行なう。

   他の fusesusp(out) と直結されると、直結相手の repo のフック bindhook:ID@Host 毎に自分の repo に fused(out, ID@Host) を報告する。その後、双方の repo を直結する。すると KLIC 処理系が双方の repo をマージする。

bindhook は fusesusp(out) の repo にフックされる中断ゴールである。fusesusp(out) に具体値が書き込まれると、bindhook は各々の担当する遠隔ホストへ具体化を報告する。すなわち、bindhook は各々 out モードの分散論理変数を表す。

一つの論理変数の読み手が複数の遠隔ホストに分散している場合、その論理変数 fusesusp(out) の具体化は複数の遠隔ホストにマルチキャストする必要がある。このため、一つの fusesusp(out) に複数の bindhook をフックする。

bindhook は次の三つのメンバを持つ。

1. ID@Host. この bindhook の識別子である

2. X. この bindhook がフックしている fusesusp(out) の repo である

3. Repo. この bindhook の具体化を待つ変数表への報告ストリームである

bindhook はフックしている fusesusp(out) から具体化、直結、削除報告を受ける。

1. X に bound(out, *Term*) が報告されたら、bindhook は Repo に bound(ID, *Term*) を報告して終了する。

2. X に ID1@Host1 が書き込まれたら、この bindhook がフックしている fusesusp(out) が fusesusp(in):ID1@Host1 と直結されたことを意味している。

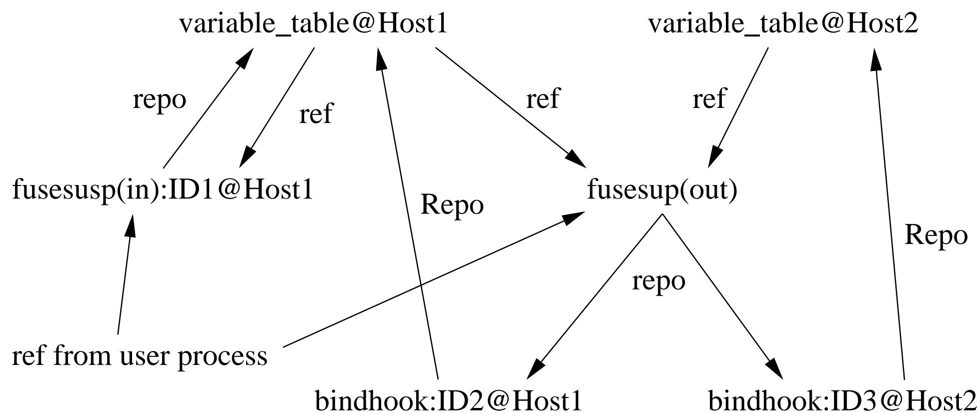
(a) Host と Host1 が同じならば、ID=ID1 の具体値を書き手である遠隔ホストに報告する必要はないので、bindhook は直結 fused(ID, ID1) を報告して終了する。

(b) Host と Host1 が異なる場合、ID@Host が間接輸出されたことを示しているので、bindhook は変数表に間接輸出メッセージ exporting(ID) を報告して終了する。

3. X に fused(out, ID1@Host1) が報告されたら、この bindhook:ID@Host をフックしている fusesusp(out) が bindhook:ID1@Host1 をフックしている他の fusesusp(out) と直結されたことを示す。

   (a) Host と Host1 が同じならば、同じ遠隔ホストに ID=ID1 の具体値を二度報告する必要はないので、bindhook:ID@Host は直結 fused(ID, ID1) を報告して終了する。つまり、一つの fusesusp(out) に bindhook:ID@Host と bindhook:ID1@Host がフックされないようにする。

   (b) Host と Host1 が異なる場合、ID@Host=ID1@Host1 はローカルホストからのマルチキャストであるので、bindhook:ID@Host は何もする必要はない。



7.3 変数表オブジェクト

dklicio は、遠隔ホストに接続されたソケット毎に一つの変数表をローカルホスト内に生成する。ソケットを接続したクライアントホスト側にあるか接続されたサーバホスト側にあるかによって、変数表は Side = client / server のいずれかの属性を持つ。

クライアントホストは open する分散変数に偶数から成る ID を割り当てる。サーバホストは奇数に基づいて ID を生成する。このことによって、クライアントホスト側・サーバホスト側で割り当てられる ID 空間は分離されている。

opener(*id*)がどちら側であるかによって*closer*(*id*)がどちら側であるかが決定される訳ではないので、たとえ変数表を輸出表と輸入表に分割してもそれらを独立に扱うことはできない。また、*closer*(*id*)によって変数表を分割したとしてもやはり *opener*(*id*)の ID 空間は分離しなければならない。いずれにせよ、変数表を独立な部分表に分割することはできない。

KLIC の GC フェイズで変数表から分散変数への参照をたどって生存性を判断してはならない。このため、変数表は KLIC の GC フェイズにおけるコピー処理を KLIC 処理系に任せず自分で行うデータオブジェクトとして実装する。

変数表オブジェクトのドライバである変数ネームサーバは、次のメソッドによって変数表オブジェクトを操作する。

1. state. 現在の ID の状態を取得するメソッドである
2. set_state. ID の状態を変更した新たな値を返すメソッドである
3. id. ID を取得するメソッドである
4. var. ID の値を取得するメソッドである
5. add. エントリを追加した新たな表を返すメソッドである
6. bind. ID に具体値を書き込んで新たな表を返すメソッドである
7. fuse. ID と他の ID1 を直結して新たな表を返すメソッドである
8. remove. エントリ ID を削除した新たな表を返すメソッドである

KLIC の GC フェイズが起き、KLIC 処理系から変数表オブジェクトの gc メソッドが呼び出されると、変数表は自分自身を新領域へコピーするが、エントリのコピーは行わない。ただし、具体化・直結によって *closing*(*id*)状態へ遷移したエントリは、たとえローカルホスト内で廃棄されても *closed*(*id*)状態へ遷移するまでは遠隔ホストで廃棄されていない可能性があるので、コピーする必要がある。

KLIC の GC フェイズの終了直後に変数表の GC フェイズが始まるようにするため、変数表の GC 処理を行う gc_variable_tables 関数を KLIC 処理系の register_after_gc_hook によって事前に登録しておく。

gc_variable_tables 関数は、ローカルホスト内にある全ての変数表のエントリを正しく書き直す。具体的には、エントリに登録された変数が新領域へコピーされているか否かを検査する。新領域へコピーされていれば、（*closing*(*id*)状態でない）エントリから変数へのポインタは旧領域を指したままになっているので、新領域のコピーを直接指すように書き換える。コピーされていなければ、その分散論理変数は廃棄されたものである。

KLIC の GC フェイズ（の後処理）から呼び出されるため、gc_variable_tables 関数はインスタンスメソッドでなくクラスメソッドでなければならない。gc_variable_tables

関数がローカルホスト内にある全ての変数表を扱うため、変数表はヒープ上のインスタンス変数でなく静的なクラス変数を通してリスト管理しておく必要がある。

7.4 変数ネームサーバ

変数ネームサーバは変数表オブジェクトのドライバであり、第6章で述べたネームサーバに相当する。

C 言語による記述部分を小さく単純にするため、変数表オブジェクトは非常に単純なデータオブジェクトとした。しかし、変数表は変化する状態を持つプロセスである。従って、変数表は不変の値としてではなくプロセスとしてのインタフェイスを持つべきである。変数ネームサーバは、値としてのインタフェイスしか持たない変数表オブジェクトに対してプロセスとしてのインタフェイスを decorate する。

変数ネームサーバは vt, repo の二つの副プロセスから成る。

vt 副プロセスは次のような通信路を持つ。

1. クライアント repo からの close 要求ストリーム
2. クライアント dklicio:varin からの var, bind, fuse, remove 要求ストリーム VTi
3. クライアント dklicio:varout からの id 要求ストリーム VTo
4. 変数表に登録された fusesusp(in) / bindhook からの報告ストリームから成るベクタ

repo 副プロセスは次のような通信路を持つ。

1. 変数表に登録された fusesusp(in) / bindhook からの報告ストリームを一本にマージしたストリーム Repo
2. vt への close 要求ストリーム

7.5 モジュール dklicio

モジュール dklicio は varin / varout という二つの副プロセスから成る。

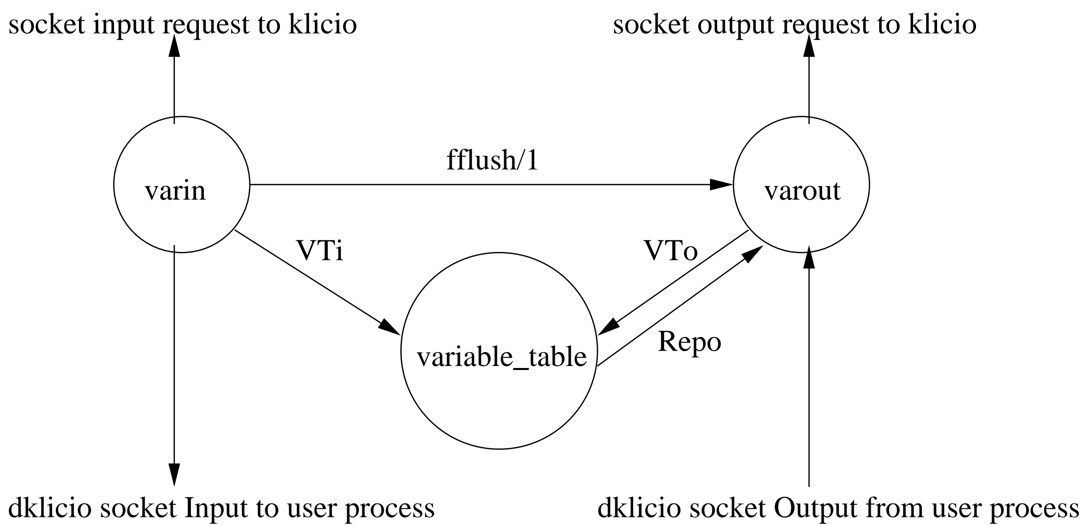
varin / varout はそれぞれ unwrap / wrap 部分処理を利用している。

varin から vaout へは fflush 要求ストリームが通じている。

fflush/1 メッセージによって KLIC 処理系がバッファの書き出しを行うとすると、fflush/1 メッセージを適度に抑制することでメッセージ通信のバッファリングを行うことになる。

しかし、入力からの読み込み直前に fflush を行うという規約は通信遅延を無視できる標準入出力に対するものであって、分散環境においてこの規約が適切であるか否かは実際の通信状態を定量化して考察する必要がある。

恐らくこの規約は最適ではないが、項を出力する度に fflush を行う既存の試作もやはり最適ではないであろう。また、fflush/1 メッセージによって実際にバッファリングが行われるか否かは下位ネットワークにも依存するであろう。



以上で、DKLIC 処理系の実装を終える。次章では、DKLIC 処理系について評価と考察を行う。

第8章　DKLIC 処理系の評価と考察

前章では、DKLIC 処理系を実装した。本章では、DKLIC 処理系について評価と考察を行う。

8.1　既存の試作からの改良点

DKLIC 処理系の既存の試作 [14], [15], [16] では未定義変数同士の直結を検出することができなかった。本研究では未定義変数同士の直結を検出することによって、二重送信の防止、中継の排除を行うことを可能にした。

また、既存の試作では分散論理変数の ID の再利用が行われていなかった。本研究では ID を再利用して良いタイミングと、ホストの ID 利用・再利用権を考察した。また、ID を正しく再利用するために必要な受信確認メッセージを導入した新たな通信プロトコル DKLICP を設計した。そして、DKLICP に従って通信を行う DKLIC 処理系を実装した。

8.2　KLIC 並列実装との比較

KLIC 処理系では、ホスト毎に輸出表と輸入表が生成される。ローカルホストから輸出された変数は輸出表に登録され、遠隔ホストから輸入された変数は輸入表に登録される。

DKLIC 処理系の変数表と同様、輸入表に対しては KLIC の GC フェイズ終了直後に輸入表の GC フェイズが起動され、輸入変数の廃棄が輸入元のホストに報告される。しかし、輸出表に対してはこのような GC 処理は行われない。

輸出される変数の ID は、その変数を輸出したホストと輸出表におけるインデクスから成る。従って、$opener(id)$による ID 空間の分離は自然になされている。KLIC 処理系の並列実装において全てのホストは順序付けられており、その順序数によってホストを特定することができるが、DKLIC 処理系の前提とする分散環境においては文字列あるいは IP アドレスによってホストを特定する。KLIC の ID 割り当て方式をそのまま拡張すると、変数識別子は KLIC の整数の範囲を越えてしまうであろう。

DKLIC 処理系はホスト間接続の対毎に変数表を一対（それぞれソケットの両端に）生成する。DKLIC 処理系はまだ例外処理を実装していないが、将来例外

処理を実装した時、問題の起きたソケットと一対の変数表のみを処理し、その他の健全なソケットと変数表はそのままにしておくということが可能である。一方、KLIC のように各ホストが輸出表と輸入表の組をただ一つしか持たない場合、その中から問題の起きた接続に対応する変数群を選び出さなければならないであろう。

8.3 中継通信の排除

プッシュ型通信における中継を排除するための通信プロトコルは、文献 [2] にも見られる。ただし、このプロトコルは通信路が単一代入であることを前提としていないため、DKLICP より効率が悪い。また、多くの変数が線形であることを暗黙に仮定している。

ホスト A から B へ通信路 K、ホスト B から C へ通信路 L が通じている時、ホスト B 内で K=L が実行されたが、K=L を参照するプロセスがホスト B 内に残っている場合、文献 [2] の方式ではホスト A 内での K の具体化メッセージはホスト B を中継してホスト C へ届く。

これは論理的に正しいが、分散環境の遅延時間が無視できないほど大きいことを考慮すると望ましくない。DKLICP はホスト B 内で K=L への参照が残っているか否かに関わらず、ホスト A–C 間の通信をホスト B が中継するような状況を排除するように設計した。

8.4 Oz との比較

並行制約言語 Oz は静的解析を目指していない。分散環境においては、素性の知れないデータやモバイルコードに対して静的解析・検証を行えることは非常に重要であろう。

Oz は多分木の完全な単一化を目指しているのに対し、DKLIC は分散変数を単に通信路と見なしている。DKLICP では out モード同士の分散変数の直結に対して何の処理も行わないが、モードに頼らない Oz はこのような通信の最適化を行うことができない。

8.5 通信例外処理

通信中に遠隔ホスト、通信路、通信相手のプロセスが落ちた場合、読み込みモードの分散論理変数に強制的に [] を書き込むことによって、落ちた通信相手からのメッセージを待ち続けるデッドロックが起きないことを保証することができるであろう。

8.6 ストリームへ連続した ID を割り当てて返事をバッファする

現在 DKLIC 処理系は分散変数への ID の割り当てに関して、その変数の型を何ら仮定していない。

しかし、ストリームへ連続した ID を割り当てれば、個々のコンスに対して受信確認を送信せず、一定時間毎に _i から _j までのコンスに対する受信確認メッセージ“ _i - _j =Removed” を送信することによって通信メッセージをバッファリングすることが可能であろう。これは TCP パケットの受信確認が「指定した番号までのパケット群」に対する受信確認であるのに良く似た方式である。

また、上記の方式以外にも dklicio の優先度を下げることによって、ストリームがある程度書き込まれてから dklicio プロセスによる具体化メッセージが送信されるようにして自然にバッファリングを行うことも可能である。こちらの方式では、そもそも ID の割り当て頻度が低くなるという望ましい性質がある。

ID の割り当てと通信メッセージ量の定量評価は今後の課題である。

8.7 線形で非 mobile なストリームの ID 即時再利用

ストリームへ連続した ID を割り当てる前節の方式をさらに押し進めると、“ _ID =[T|ID]” のようにストリームの cdr にそのストリームの ID を即時再利用する方式にたどり着く。この方式では受信確認は不要である。

この場合、ストリームの ID は変数でなくストリームの識別子であることになる。従って、このストリームは単一代入でなくなる。

```
append(X, Y, Z) :- X = [] |
  Z = Y.
append(X, Y, Z) :- X = [A | X1] |
  Z = [A | Z1],
  append(X1, Y, Z1).
```

append/3 述語においてストリーム X, Y, Z が単一代入でない場合、Z = Y によって Z と Y のどの時点 (continuation) を直結するのか判断できない。

また、ストリームが間接輸出される場合も、ストリームのどの時点が間接輸出されているのか判断することはできない。どのような場合に、ストリームの cdr にストリームの ID を即時再利用して良いのか検証することは今後の課題であろう。

8.8 strictly linear ID allocation

KL1 言語の静的解析の体系には、型・モード体系の他に参照数体系がある。参照数がちょうど二つであるような変数は、書き手一人に対して読み手一人であるため、破壊的代入が可能である。この性質を線形性と呼ぶ。

線形性の概念を具体値にも押し進めた strict linearity は、プログラムが引数に渡された資源のみを利用して実行を行うことを示す性質である [8]。strictly linear なプログラムは利用する資源が静的に判明するため、GC を必要としない。

strict linearity の概念を変数への ID 割り当てにも適用すると、次のような stack/2 サーバ述語は下記の strict linear な stack/3 サーバ述語に書き換えることができる。

```
stack(S, D) :- S = [] |
  true.
stack(S, D) :- S = [push(X) | S1] |
  stack(S1, [X|D]).
stack(S, D) :- S = [pop(X) | S1], D = [Y|D1] |
  X = Y,
  stack(S1, D1).

stack(S, ^(R), D) :- S = [](*) |
  R = [](D).
stack(S, ^(R), D) :- S = [push([X|*]) | S1] |
  R = [push(*) | R1],
  stack(S1, R1, [X|D]).
stack(S, ^(R), D) :- S = [pop(^(X)) | S1], D = [Y|D1] |
  X = Y,
  R = [pop([*|*]) | R1],
  stack(S1, ^(R1), D1).
```

全てのサーバが strict linear であると仮定すると、DKLIC 処理系は受信確認を省略することができる。しかし、これはサーバプログラマが受信確認の送信を明示することを意味し、受信確認の責任が DKLIC 処理系からサーバプログラマへ移っただけであると言える。

8.9 今後の課題

現在 DKLIC 処理系は未定義変数の輸出入とその具体化、直結、廃棄のみを管理している。しかし、記号処理言語 KL1 において記号の効率的な輸出入、送受信に対する要望もあり得るであろう。

また、既に具体化された項の送受信をバッファリングすることは通信量の削減のために必要である。このようなサーバはユーザ定義サーバとして実装し、DKLICP の上位プロトコルを構築することが可能であろう。

DKLIC 処理系は扱うプログラムがモード付きであることを前提としているが、KLIC 翻訳系はモード解析を行わない。DKLIC プログラマは KLIC に加えて Kima を利用することによってモード付きプログラムを記述する必要があるであろう。しかし、プログラマが Kima を強制されるのは少々面倒である。今後 KLIC あるいはそれに代わる翻訳系は Kima のような静的解析器を含むことが望ましい。

以上で、DKLIC 処理系の評価と考察を終える。次章では、本論文についてまとめる。

第9章　まとめ

本章では、前章までの議論をまとめる。

本研究では、分散プログラミングに不可欠な処理の記述を手続き型言語は不得意としていることを踏まえ、動的プロセス生成、自動同期、非同期メッセージ通信を宣言的に記述する KL1 言語に基づいて宣言型ソケット通信機能を提供する DKLIC 処理系を実装した。このことによって、複雑な分散プロセスや通信プロトコルを直観的に記述することを可能とした。

直結・廃棄されたことを検出する分散論理変数オブジェクト fusesusp を実装することによって、既存の DKLIC 処理系では不可能であった二重送信や通信の中継の排除を行い、分散論理変数の ID を早期に再利用することを可能とした。

また、KL1 言語の高度な静的解析技術を分散プログラミングに適用し、素性の知れない通信データやモバイルコードを検証することを可能とした。加えて、分散論理変数の状態遷移を明確に示した。分散論理変数の open（通信路の確立）という概念は既存の静的解析の体系に明確には現れていなかったものである。

モード付けによって静的に排他制御された分散 KL1 プログラムを前提として効率良く分散論理変数を通信するプロトコル DKLICP を設計し、DKLICP に従って通信を行う DKLIC 処理系を実装した。

例外処理、より効率の良い上位の通信プロトコルの設計、検索サーバを含む上位サーバの実装は今後の課題である。

以上で、本論文を終える。

謝辞

本研究の様々なアイデアは、上田研究室言語班のみなさんとの議論の中で生まれてきたものです。

　その中でも、特に上田和紀先生と加藤紀夫さんは言語班の一員として議論の相手となってくれたばかりでなく、論文の書き方について丁寧に指導して下さいました。深くお礼申し上げます。

参考文献

[1] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[2] Hiroshi Nakashima and Yu Inamura, An Efficient Message Transfer Mechanism Bypassing Transit Processors. In *Proc. Joint Symposium on Parallel Processing*, pp. 123–130, June 1992.

[3] 瀧和男 編, 第五世代コンピュータの並列処理. bit 別冊, 共立出版, 1993.

[4] Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.

[5] 関田大吾, Inside KLIC Version 1.0. KLIC Task Group / AITEC / JIPDEC, 1998. `http://www.klic.org/software/klic/inside/master.html`

[6] `http://www.klic.org/`
`http://www.ueda.info.waseda.ac.jp/%7Etakagi/kl1/klic-3.003/fix.html`
`http://www.ueda.info.waseda.ac.jp/%7Etakagi/kl1/klic-3.003/rewriting.html`

[7] Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

[8] Ueda, K., STRICT LINEARITY

[9] 網代育大, 上田和紀, Kima: 並行論理プログラム自動修正系. コンピュータソフトウェア別冊 ソフトウェア発展, Vol. 18, No. 0 (2001), pp. 122–137.

[10] 加藤紀夫, 上田和紀, 並行論理プログラムにおける逐次実行部分の抽出方法論. In *Proc. JSSST Workshop on Programming and Programming Languages (PPL2001)*, 2001.

[11] 倉持聡, KLIJava: KL1 から Java へのコンパイラおよび実行環境. 早稲田大学理工学研究科情報学修士論文, 1999.
http://www.ueda.info.waseda.ac.jp/%7Esatoshi/klijava/

[12] 五十嵐宏, 分散 KL1 言語処理系の設計と実装. 早稲田大学理工学部情報学科卒業論文, 1999.

[13] 五十嵐宏, メタインタプリタに基づく分散並行論理型言語処理系. 早稲田大学大学院理工学研究科修士論文, 2001.

[14] 高木祐介, KL1 による分散 KL1 言語処理系の実装. 早稲田大学理工学部情報学科卒業論文, 2000. http://www.ueda.info.waseda.ac.jp/%7Etakagi/

[15] 高木祐介, 上田和紀, dklic: KL1 による分散 KL1 言語処理系の実装. In *Proc. JSSST Workshop on Systems for Programming and Applications (SPA2001)*, 2001. http://www.dcl.info.waseda.ac.jp/SPA2001/

[16] 高山啓, 並行論理型言語 KL1 による分散 KL1 言語処理系の実装. 2000 年度早稲田大学理工学部情報学科卒業論文, 2001.

[17] 松村量, 並行論理型言語 KL1 の分散拡張のための遠隔ノード管理の実装. 2000 年度早稲田大学理工学部情報学科卒業論文, 2001.

[18] 金木佑介, 並行論理型言語 KL1 における例外処理の実装. 2001 年度早稲田大学理工学部情報学科卒業論文, 2002.

[19] 粉川友宏, DKLIC 処理系における分散資源の位置管理. 2001 年度早稲田大学理工学部情報学科卒業論文, 2002.

[20] P. Deransart, A. Ed-Dbali, L. Cervoni, Prolog: The Standard Reference Manual. Springer-Verlag, 1996.

[21] Ehud Shapiro, A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[22] Vijay A. Saraswat, Ken Kahn, and Jacob Levy, Janus: A Step towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pp. 431–446, 1990.

[23] Diaz, M., Rubio, B. and Troya, J. M., DRL: A Distributed Real-Time Logic Language. In *Comput. Lang.*, Vol. 23, No. 2–4 (1997), pp. 87–120.

[24] Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M. and Scheidhauer, R., Mobile Objects in Distributed Oz. In *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 5, September 1997, pp. 804–851.

[25] Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. In *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.

[26] Haridi, S., Van Roy, P., Brand, P., Mehl, M., Scheidhauer, R. and Smolka, G., Efficient Logic Variables for Distributed Computing. In *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, May 1999, pp. 569–626.

[27] J. L. ピーターソン, A. シルバーシャッツ 共著, 宇津宮孝一, 福田晃 共訳, オペレーティングシステムの概念. 培風館, 1987.

[28] 山口和紀 監修, The UNIX Super Text. 技術評論社, 1992.

[29] サミュエル P. ハービソン, ガイ L. スティール 共著, 斎藤信男 監訳, 新・詳説 C 言語. ソフトバンク, 1994.

[30] R.C. ホルト 他, 湯浅太一 監訳, 松田元彦, 三好博之, 竹岡尚三 訳, プログラミング言語 Turing — 言語仕様の基礎理論と設計技法. 啓学出版, 1990.
DKLICP 言語の形式文法を階級分類する参考にした

[31] James Gosling, Bill Joy, Guy Steele 著, 村上雅章 訳, The Java 言語仕様. アジソン・ウェスレイ, 1997.

[32] Samuel C. Kendall, Jim Waldo, Ann Wollrath, Geoff Wyant, A Note on Distributed Computing. Sun Microsystems Research Technical Report TR-94-29, 1994. `http://www.sun.com/research/techrep/1994/abstract-29.html`
分散系には、逐次系にない通信遅延、メモリモデル、並行性、部分的な故障があるため、分散オブジェクトを逐次オブジェクトと同様に扱うことはできないと主張している

付 録A　補足

A.1　モードなしのプログラムを等価なモード付きプログラムへ書き換える

本節では、例示によって非形式的に書き換えを説明する。モードなしのプログラムは、次のようにモード付きのプログラムに書き換えることができる。

結果の検査を行うべきところで、あるべき結果を積極的に書き込んでしまうプログラムはモード付きでない。このようなプログラムをモード付きに書き換えるには、結果を検査する新たな述語を加える。

```
p(Arg1, Arg2) :- true |
  Result = normal,
  proc(Arg1, Result),
  post(Arg2).

p(Arg1, Arg2) :- true |
  proc(Arg1, Result),
  p1(Arg2, Result).

p1(Arg2, Result) :- Result = normal |
  post(Arg2).
p1(Arg2, Result) :- Result = abnormal |
  process_exception(Arg2).
```

木の探索において、各部分木に対して子プロセスを生成し、早い者勝ちで結果を書き込むようなプロセスはモード付きでない。このように並行に書き込まれる結果は KLIC 処理系が提供する並行マージサーバ merge を利用して、結果の並行書き込みを一本のストリームへ順序付けるべきである。

銀行口座の残高値そのものを早い者勝ちで書き込むのは明らかに誤りである。ロック、セマフォ、モニタのような書き込み権の取得を競争するように書き直すべきであろう。

A.2 DKLIC-XMLP: inter-operability

本節では、他言語において KL1 項の字句解析、構文解析を容易に行うために DKLICP を XML 言語に対応させる例を挙げる。

例えば、次の KL1 項は下記の XML として表すことができる。

```
f([X], {a, "s"})

<functor arity="2">
 <name>f</name>
 <arg number="1">
  <cons>
    <car><variable>X</variable></car>
    <cdr><atom>[]</atom></cdr>
  </cons>
 </arg>
 <arg number="2">
  <vector length="2">
    <elem number="0"><atom>a</atom></elem>
    <elem number="1"><string>s</string></elem>
  </vector>
 </arg>
</functor>
```

付 録B　DKLIC 処理系の仕様と実装

```
dklicio
  bind inet(Port) ^Result=normal(^ServerSocket)
  connect inet(Host,Port) ^Result=normal(In,^Out)

ServerSocket
  accept ^Result=normal(In,^Out)

naming
  register Name Host Port
  lookup Name ^Result=normal(Host,Port)

exec
  go Agent
```

B.1　dklicio

B.2　変数ネームサーバ

B.3　変数表オブジェクト

B.4　fusesusp

付 録C　DKLIC 処理系の応用例

C.1 Remote Predicate Call

C.2 Chat Server

C.3 検索サーバ

C.4 Mobile Agent