

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)	8
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	22
5.4	三者通信のプロトコル	22
第 6 章	dklicio の設計	26
6.1	分散論理変数 distributed	26
6.2	ID manager (Name Server)	27
6.3	変数表オブジェクト variable_table	28
第 7 章	dklicio の実装	32
7.1	分散論理変数オブジェクト	32
7.1.1	fusesusp	32
7.1.2	bindhook	37
7.2	変数表オブジェクト	38
7.3	変数ネームサーバ	39
7.4	モジュール dklicio	40
第 8 章	DKLIC 処理系の評価と考察	42
8.1	既存の試作からの改良点	42
8.2	KLIC 並列実装との比較	42
8.3	中継通信の排除	43
8.4	Oz との比較	43
8.5	通信例外処理	43
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	44
8.7	線形で非 mobile なストリームの ID 即時再利用	44
8.8	strictly linear ID allocation	45
8.9	今後の課題	45
第 9 章	まとめ	47
	参考文献	49
付 録 A	補足	52
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	52
A.2	DKLIC-XMLP: inter-operability	53
付 録 B	DKLIC 処理系の仕様と実装	54
B.1	主な開発環境	54
B.2	dklicio	55
B.3	変数ネームサーバ	63
B.4	変数表オブジェクト	66

B.5	fusesusp	77
付 録 C	DKLIC 処理系の応用例	84
C.1	Remote Predicate Call	84
C.2	検索サーバ	85

目次

5.1	state transition of distributed variables	21
5.2	reconnection of channels	24
7.1	channel and its modes	33
7.2	fusesusp (in and out)	34
7.3	fusion of out-out to same remote host	35
7.4	fusion of in-out to same remote host	35
7.5	fusion of out-out to different remote hosts	36
7.6	fusion of in-out to different remote hosts	36
7.7	fusesusps and bindhooks	38
7.8	dklicio and variable_table	41

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すべて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ー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ー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ースは 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ー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入力・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。

これらの機構は基本的に逐次処理を行う手続き型言語に基づいているため、非同期通信と並行処理を記述するのは非常に難しい。[32] は、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。

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やサーバへのアクセス権を、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な応用されている。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、それらを利用する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、並行処理や非同期メッセージ通信の記述は難しい。

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を構成する次のような操作群は、他の操作に割り込まれることなく不可分に実行されなければならない。

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ありふれた事故である。

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。

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

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 $\wedge/1$ 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ユーザ定義サーバへの通信路 `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` と名付けられたユーザ定義サーバへの通信路 `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ユーザ定義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ユーザ定義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、ユーザ定義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` サーバを通じてユーザ定義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。形式文法の階層については [30] を参考にした。

1. 正則文法 $lex : String \rightarrow Token^*$. KL1 言語の字句解析と同一である
2. 文脈自由文法 $parse : Token^* \rightarrow Term^*$. KL1 言語の構文解析と同一である
3. 文脈依存文法 $unmarshal =_{\text{def}} marshal^{-1} : (String \rightarrow)Term^*$. 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、廃棄メッセージ “ $_ID = Removed$ ” 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、廃棄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。 $_ID$ の割当て権がホスト 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、廃棄メッセージを受信すると、通信路を閉じて分散論理変数の識別子 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、識別子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、通信路を開いて識別子 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

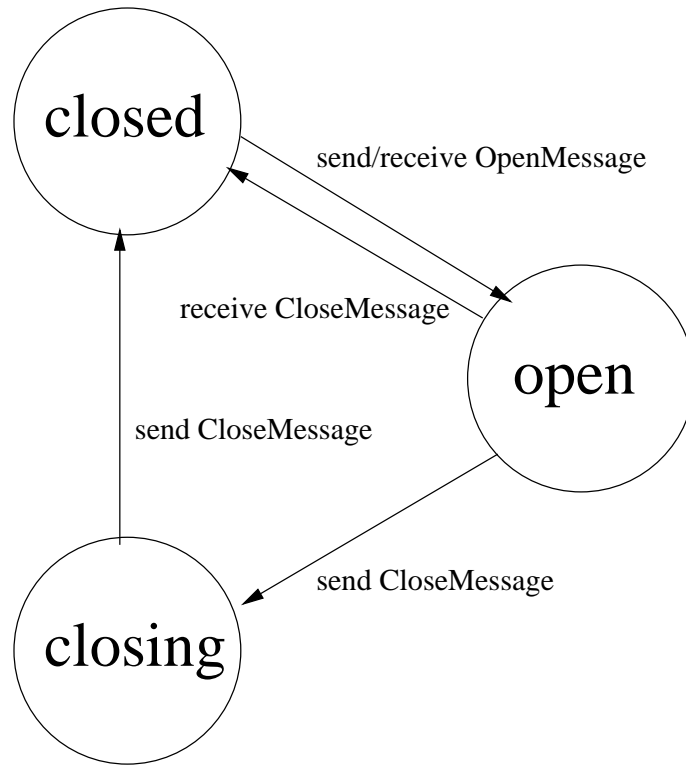


図 5.1: state transition of distributed variables

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、 id が重複して発行されることはなく、 $opening(id)$ 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、*OpenMessage(id)*を送信して *id* を再利用することができる。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* を含む 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]
      :
```

ここに現れる変数 *Request0*, *Request1*, *Request2*, *Answer0*, *Answer1* は全て、クライアントが *open* するものである。

このように典型的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。

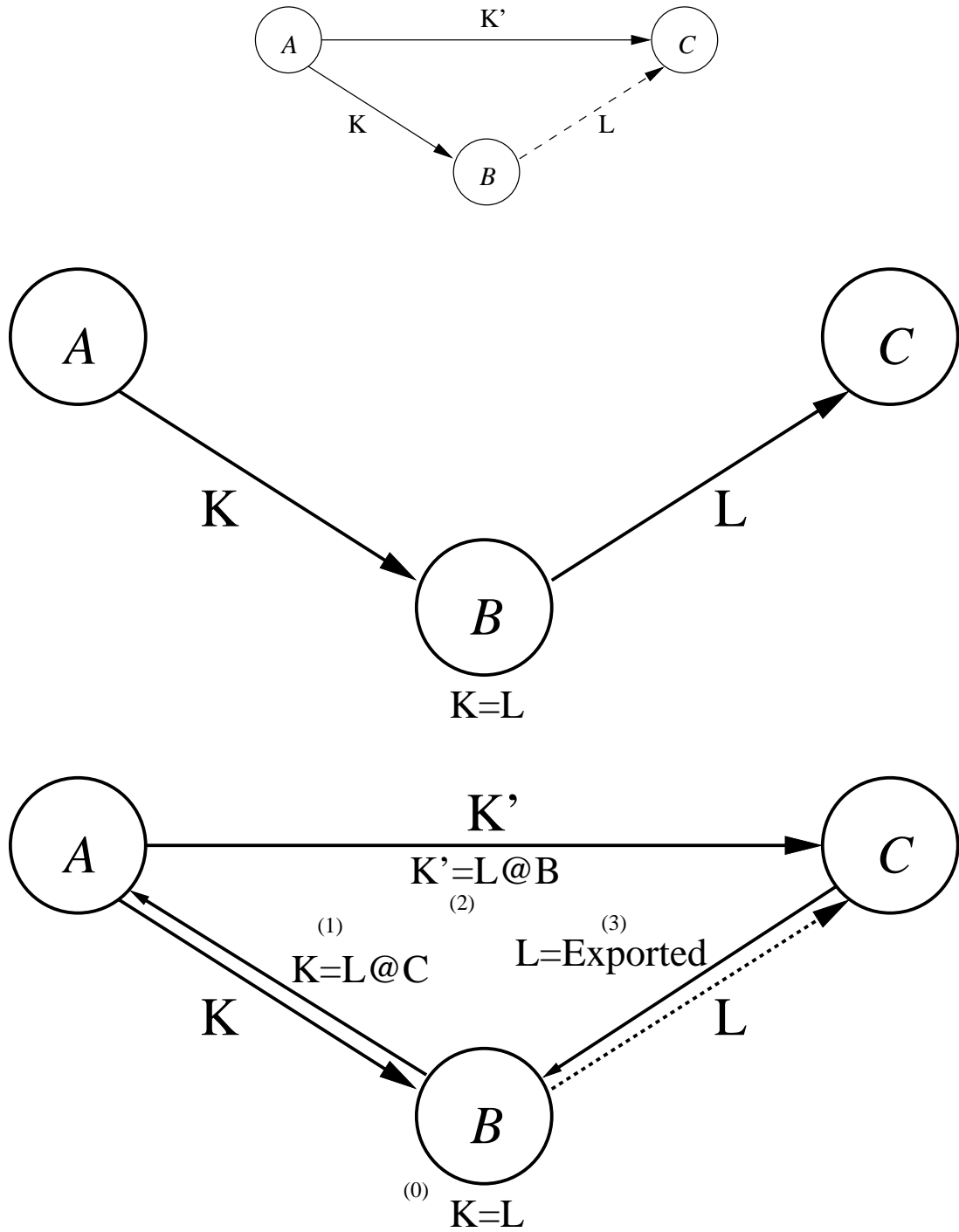
- (a) 間接輸出メッセージを受信したホスト A はホスト C へ、間接輸出確認メッセージ “ $_ID1 @B = _ID2$ ” を送信する。ここで $_ID2$ は、論理変数 K のホスト $A-C$ 間における新たな通信路 K' の識別子である。ただし、ホスト A 内で論理変数 K に具体値 *Term* が既に関与していた場合、間接輸出確認メッセージ “ $_ID1 @B = Term$ ” を送信すれば良く、新たに K' を open する必要はない。

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。



☒ 5.2: reconnection of channels

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ず、また、中継を終えたはずのホストへ情報が通信されることは保安上問題があるからである。

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（プッシュ型であるので）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(\Leftrightarrow 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。

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

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、

$fuse : Dist \times Constraints \times Term \rightarrow$

$Constraints \times \{bound/2, fused/2\}$

$fuse((id, var), constraints, term) =$

$(constraints \cup \{var = term\},$

$message(id, term))$

where

$message = bound \quad \text{if } term \in Bound$

$message = fused \quad \text{if } term \notin Bound$

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。

$$Table =_{\text{def}} 2^{Dist} \ni \{(id, var)\}$$

$$\forall i, j; (id_i, var_i), (id_j, var_j) \in table \Rightarrow$$

$$(id_i = id_j \Leftrightarrow var_i = var_j) \quad (6.1)$$

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)。

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

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트리 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。

$$\begin{aligned}
& add : State \times Var \rightarrow ID \times State \\
& add(table, ns, var) = \\
& \quad (id, table \cup \{id\}, ns_1) \\
& \quad \text{where } get(ns) = (id, ns_1)
\end{aligned}$$

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。

$$\begin{aligned}
& remove : State \times ID \rightarrow State \\
& remove(table, ns, id) = \\
& \quad (table \setminus \{(id, var)\}, release(ns, id))
\end{aligned}$$

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。

$$\begin{aligned}
& getID : State \times Var \rightarrow ID \\
& getID(table, ns, var) = id \\
& \quad \text{where } (id, var) \in table
\end{aligned}$$

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。
 $fused : State \times ID \times Var \rightarrow State$
 $fused(table, ns, id, var) = (table, ns) \text{ if } id < getID(var)$
 $= remove(table, ns, id) \text{ if } id > getID(var)$
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 空間を縮めると言う。

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 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 [5]。

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 generate メソッドが呼び出され、具体値や中断構造を生成する処理が行われる。その後、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 処理系は中断構造のフック群を単にマージし、それらのフック群には直結を報告しない。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。

7.1.1 fusesusp

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。

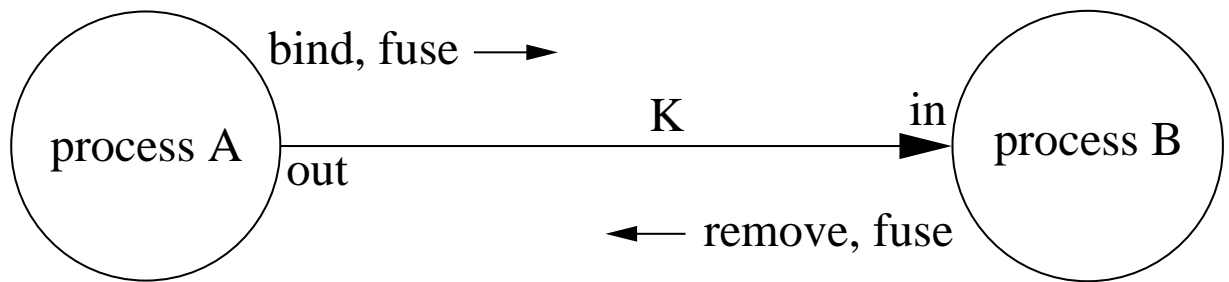


図 7.1: channel and its modes

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、廃棄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 $\text{bound}(\text{id}, \text{Term})$ 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。

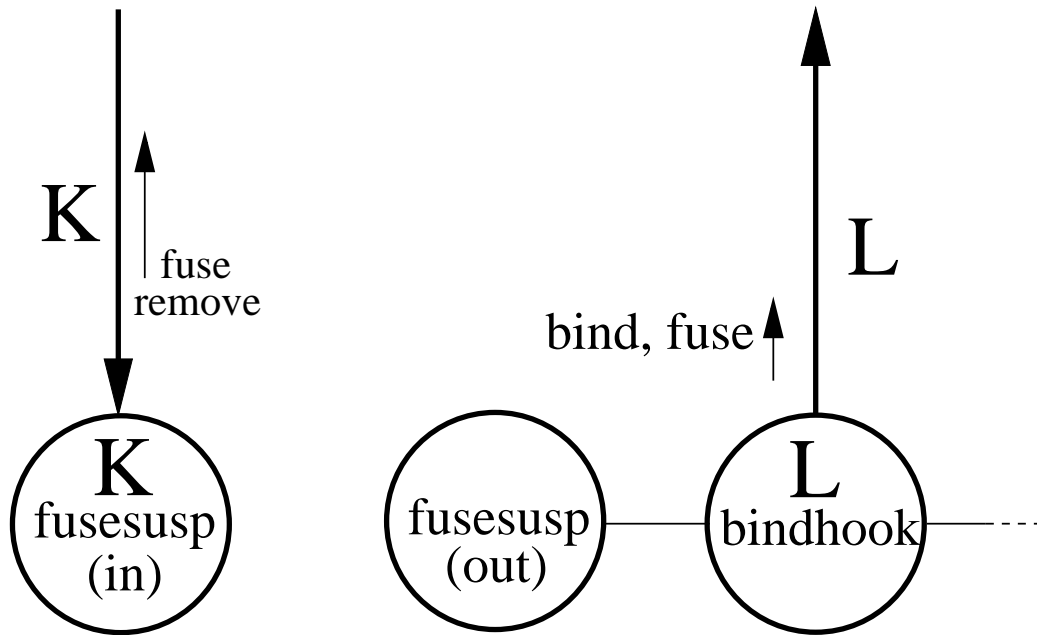


図 7.2: fusesusp (in and out)

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。

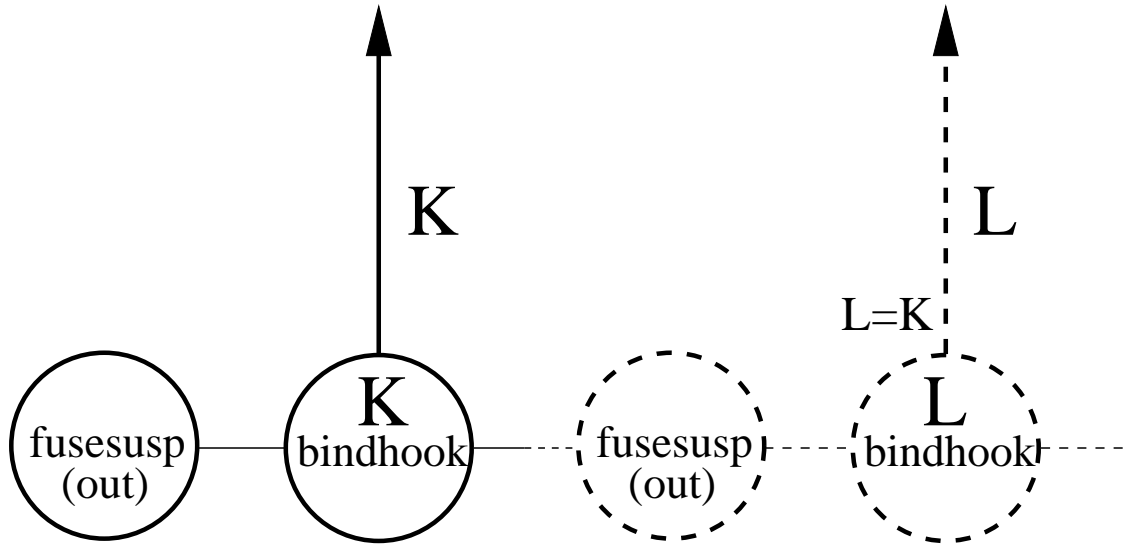
2. fusesusp(out) は out モードの分散論理変数に対応付けられたローカルホスト内の論理変数を表す。fusesusp(out) のメンバは次のような意味を持つ。

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) 同士の区別は不要であるので、全ての fusesusp(out) の識別子は out とする。

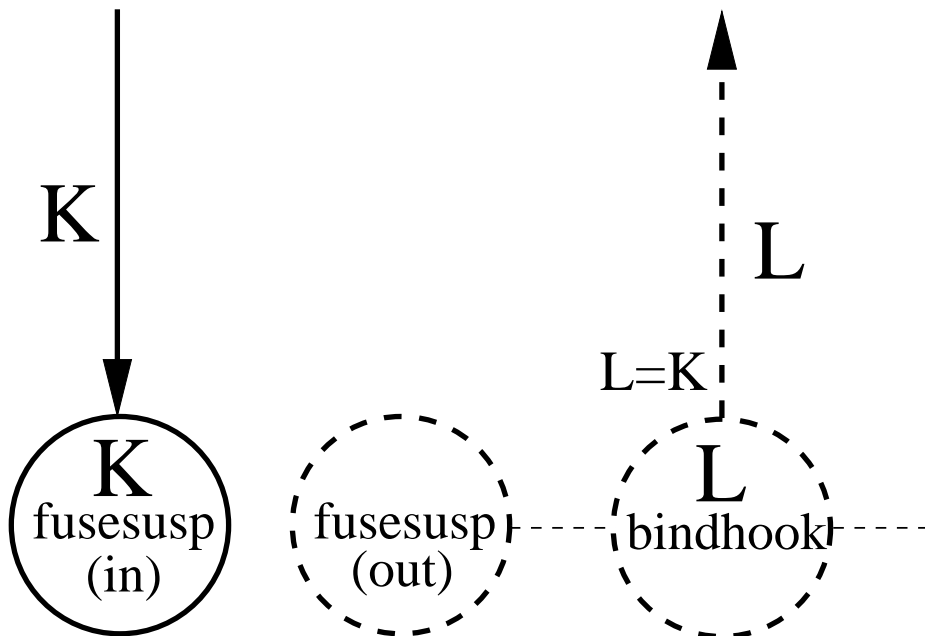
repo. この fusesusp の具体化を待つ bindhook への報告ストリームである。

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。

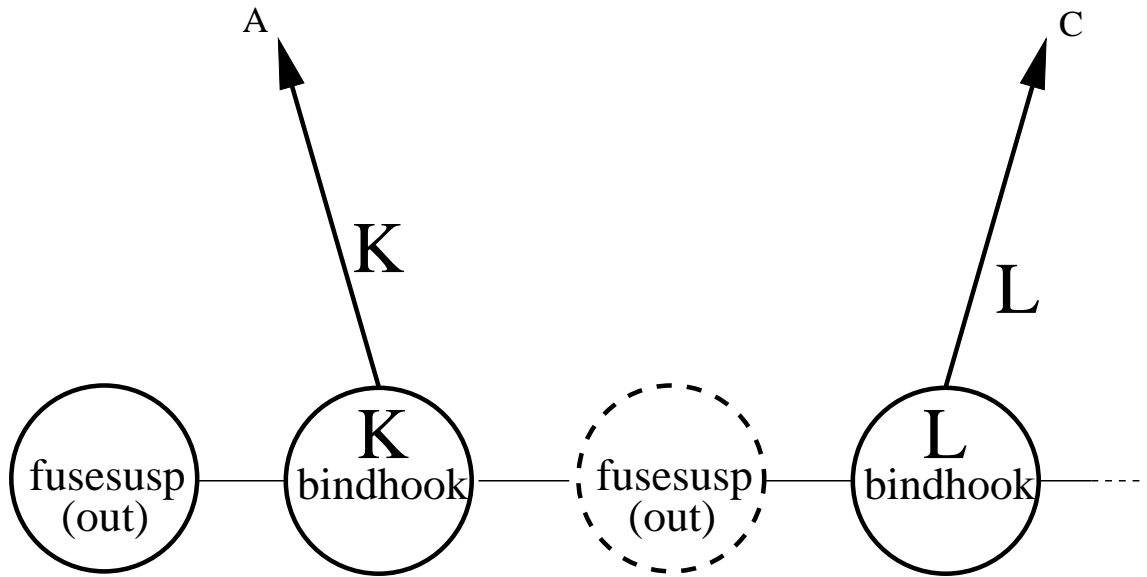
他の fusesusp(out) と直結されると、直結相手の repo のフック bindhook:ID@Host 毎に自分の repo に fused(out, ID@Host) を報告する。その後、双方の repo を直結する。すると KLIC 処理系が双方の repo をマージする。



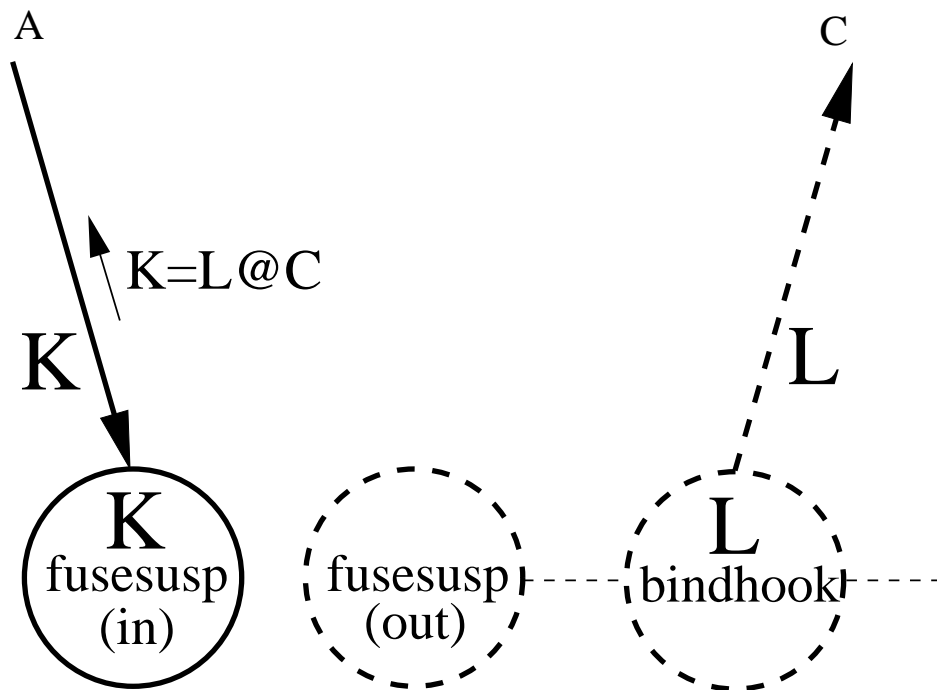
☒ 7.3: fusion of out-out to same remote host



☒ 7.4: fusion of in-out to same remote host



☒ 7.5: fusion of out-out to different remote hosts



☒ 7.6: fusion of in-out to different remote hosts

7.1.2 bindhook

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、廃棄報告を受ける。

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。

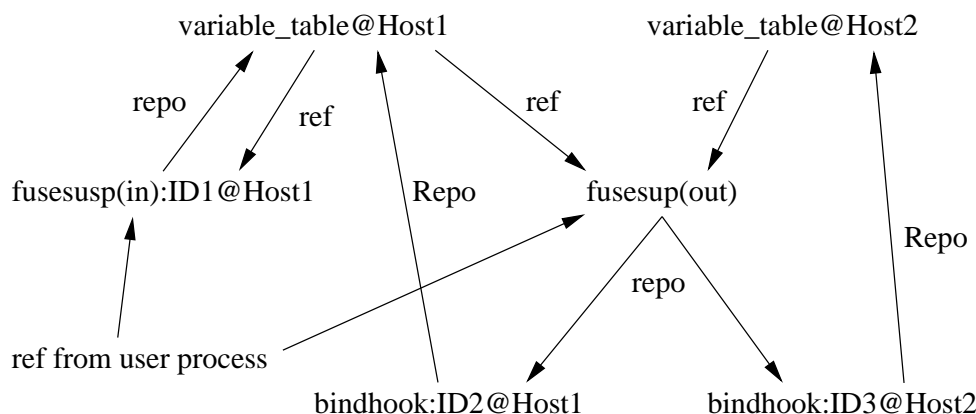


図 7.7: fusesusps and bindhooks

- (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。

7.2 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ため、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なければ、(`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。

7.3 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

7.4 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。

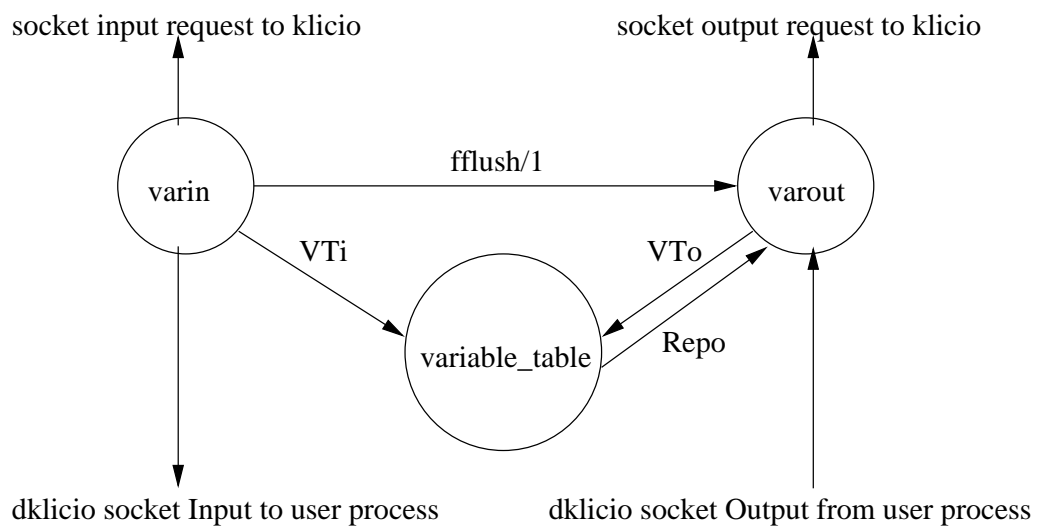
varin から vaout へは fflush 要求ストリームが通じている。

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。



☒ 7.8: dklicio and variable_table

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对（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 “ $_i - _j = \text{Removed}$ ” 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] Kazunori Ueda, Resource-Passing Concurrent Programming. In *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software*, Kobayashi, N. and Pierce, B. (Eds.), Lecture Notes in Computer Science 2215, Springer, 2001, pp. 95–126.
- [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.
- [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>

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).
```

上のプログラムは、手続き `proc` が正常終了することを前提として `Result = normal` をハードコーディングしてしまっている。しかし、手続きの正常終了を前提として結果の検査を怠るのは良いプログラミングスタイルとは言えない。たとえば `proc` が `Result = normal` を返すことが保証されていたとしても、手続き `post` が `proc` の後に実行されるべき場合、`proc` の終了を検査することは重要である。

上のプログラムは下のように書き換え、結果の検査を行うのが良いプログラミングスタイルである。

```
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。

KL1 項も XML データも共に木構造であるので、それらの対応付けは容易である。

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

例えば、上の KL1 項は下記の XML データとして表すことができる。

```
<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>
```

DKLICP におけるメッセージ (パケット) は全て KL1 項によって構成されるので、これらを XML データに置き換えれば DKLICP を XML 対応とすることができる。

付録B DKLIC 処理系の仕様と実装

B.1 主な開発環境

プログラムの開発・実行確認は次の環境で行った。

1. Intel Pentium III 800EB
2. PC-AT 互換機
3. Debian GNU/Linux 2.2
4. KLIC-3.003-2002-01-25 [6]

次節以降に `dklicio` のソースプログラムを挙げるが、現在残っている問題点は次の通りである。

1. DKLICP に完全に対応していない。
 - (a) ソケット接続時にクライアントがホストを申告しない。単純な実装としてホストをプログラマに申告させることも考えられるが、`hostname` などを利用して自動化することが望ましいであろう。
 - (b) 三者通信のプロトコルを完全には実装していない。一般にソケットを利用するサーバはホストとポート番号によって特定される。従って、ホストを申告する際にポート番号も共に申告すべきであろう。前述のようにホストは自動的に特定することが可能であり、すべきであろうが、ポート番号はプログラマに申告させるべきであろうか。プログラマにとって使いやすい API を考察し、実装する必要がある。
2. 変数表オブジェクトの `gc_variable_tables` 関数を実装していない。2002-01-25 時点における KLIC-C API は悲惨なものであり、このアナクロなバージョンで開発・試験を行うことは困難である。もっと容易に開発を行える KLIC-C API を実装するため、現在 KLIC の洗練を進めている。

例えば、変数表オブジェクトの次の関数

```
SET_NEWOBJ(q* g_allocp, GD_OBJ_TYPE* GD_SELF, GD_OBJ_TYPE* newself,
    long g_method_functor, q* GD_ARGV, int len, int size )
```

の引数のうち `g_method_functor` や `GD_ARGV` は使われていないように見えるが、これは KLIC-C API が中途半端な隠蔽を行っているためである。これらの引数が必要であることに気付くためには、これらの引数なしでコンパイルしてみて、ほとんど意味不明なエラーメッセージを見せられ、それを読み解かなければならなかった。また、ほとんどドキュメントのない API を誤って使用した結果、セグメンテーションフォールトが起きることもしばしばであった。

B.2 dklicio

```
1  /*****
2  * dklicio: DKLIC io streams
3  *
4  * 2002-01-18 TAKAGI Yusuke
5  *
6  * Module: dklicio (cf. klicio)
7  *
8  * Predicates:
9  *   connect +inet(+Host,+Port) -normal(In, Out)
10 *   bind +inet(+Port) -normal(ServSock)
11 *
12 * Message on ServSock:
13 *   accept -normal(In, Out)
14 *
15 * Message on In:
16 *   -Term
17 *
18 * Message on Out:
19 *   +Term
20 *
21 * Messages from variable_table:
22 *   bound   +ID +Term
23 *   fused   +ID +ID1
24 *   removed +ID ?Var
25 */
26
27 :- module dklicio.
28
29 :- public connect/2, bind/2.
30
31 connect(Spec, Res) :- Spec=inet(Host, Port), 0 =< Port, Port < 65536 |
32   klicio:klicio([connect2(Spec, R)]),
```

```

33     vario(Host, client, R, Res).
34 otherwise.
35 connect(Spec, Res) :- true |
36     Res = abnormal.
37
38 bind(Spec, Res) :- Spec=inet(Port), 0 =< Port, Port < 65536 |
39     klicio:klicio([bind(Spec, R)]),
40     bind(R, Res).
41 bind(R, Res) :- R = normal(SS) |
42     Res = normal(ServSock),
43     accept(ServSock, SS).
44 otherwise.
45 bind(R, Res) :- true |
46     Res = abnormal.
47
48 accept(ServSock, SS) :- ServSock=[] |
49     SS = [].
50 accept(ServSock, SS) :- ServSock=[accept(Res)|ServSock1] |
51     SS = [accept2(R) | SS1],
52     vario("localhost", server, R, Res),
53     accept(ServSock1, SS1).
54 accept(ServSock, SS) :- ServSock=[Mes|ServSock1], Mes \= accept(x) |
55     accept(ServSock1, SS).
56
57 vario(Host, Side, R, Res) :- R = normal(I, O) |
58     Res = normal(In, Out),
59     variable_table:new(Host, Side, VTi, VTo, Repo),
60     varin(In, I, IO, VTi) @lower_priority,
61     varout(Out, O, IO, VTo, Repo).
62 otherwise.
63 vario(Host, Side, R, Res) :- true |
64     Res = abnormal.
65
66
67 check(X, X1, Res, Res1) :- X=X1 |
68     Res = Res1.
69 otherwise.
70 check(X, X1, Res, Res1) :- wait(X) |
71     Res = abnormal.
72
73 wait(X, Res, Res1) :- wait(X) |
74     Res = Res1.
75
76
77 /*****
78 * varin server > unwrap
79 *
80 * client: remote host

```

```

81 * ^In : input terms
82 * ^I  : socket input (file_io) server
83 * ^IO : fflush stream to varout
84 * ^VTi: variable_table server
85 */
86
87 varin(In, I, IO, VTi) :- true |
88     IO = [fflush(F) | IO1],
89     check(F, 0, I, [getwt(R) | I1]),
90     varin(R, In, I1, IO1, VTi).
91
92 varin(R, In, I, IO, VTi) :- R=normal(end_of_file) |
93     In = [],
94     I = [],
95     O = [],
96     VTi = [].
97 varin(R, In, I, IO, VTi) :- R=normal(functor(variable(S)=WT)),
98     string(S,Len,8), Len > 1 |
99     name_id(S, ID),
100    varin(ID, WT, In, I, IO, VTi).
101 otherwise.
102 varin(R, In, I, IO, VTi) :- R=normal(WT), functor(WT,_,1) |
103     unwrap(WT, T, VTi, VTi1),
104     wait(T, In, [T | In1]),
105     varin(In1, I, IO, VTi1).
106
107 varin(ID, WT, In, I, IO, VTi) :- ID >= 0, WT=variable("Removed") |
108     VTi = [remove(ID) | VTi1],
109     varin(In, I, IO, VTi1).
110 varin(ID, WT, In, I, IO, VTi) :- ID >= 0, WT=variable(S),
111     string(S,Len,8), Len > 1, string_less_than(S, "Removed") |
112     name_id(S, ID1),
113     VTi = [fuse(ID, ID1) | VTi1],
114     varin(In, I, IO, VTi1).
115 varin(ID, WT, In, I, IO, VTi) :- ID >= 0, WT \= variable(x) |
116     unwrap(WT, T, VTi, VTi1),
117     wait(T, VTi1, [bind(ID, T) | VTi2]),
118     varin(In, I, IO, VTi2).
119
120
121 /*****
122 * varout server > wrap
123 *
124 * client: local host
125 * Out : output terms
126 * ^O   : socket output (file_io) server
127 * IO   : fflush stream from varin
128 * ^VTo : variable_table server

```

```

129 * Repo: bound, fused, removed stream from variable_table
130 */
131
132 varout(Out, O, IO, VTo, Repo) :- IO=[fflush(F)|IO1] |
133   O = [fflush(F) | O1],
134   varout(Out, O1, IO1, VTo, Repo).
135 varout(Out, O, IO, VTo, Repo) :- Out=[], Repo=[] |
136   O = [fflush(F)],
137   check(F, O, VTo, []).
138 varout(Out, O, IO, VTo, Repo) :- Out=[T|Out1], wait(T) |
139   wrap(T, WT, VTo, VTo1),
140   wait(WT, O, [putwt(WT) | O1]),
141   varout(Out1, O1, IO, VTo1, Repo).
142 varout(Out, O, IO, VTo, Repo) :- Repo=[bound(ID,X)|Repo1], wait(X) |
143   wrap(X, WX, VTo, VTo1),
144   varout(ID, putwt(WX), Out, O, IO, VTo1, Repo1).
145 varout(Out, O, IO, VTo, Repo) :- Repo=[fused(ID,ID1)|Repo1], ID1 >= 0 |
146   id_name(ID1, S1),
147   varout(ID, fwrite(S1), Out, O, IO, VTo, Repo1).
148 varout(Out, O, IO, VTo, Repo) :- Repo=[removed(ID)|Repo1] |
149   varout(ID, fwrite("Removed"), Out, O, IO, VTo, Repo1).
150
151 varout(ID, Put, Out, O, IO, VTo, Repo) :- ID >= 0 |
152   id_name(ID, S),
153   O = [fwrite(S), putc("#="), Put, fwrite(".\n") | O1],
154   varout(Out, O1, IO, VTo, Repo).
155
156
157 /*****
158 * wrap < varout server
159 *
160 * T : term to wrap
161 * ^Res: wrapped T
162 * VTo: variable_table server
163 */
164
165 wrap(Term, Res, VTo, VTo_) :- atom(Term) |
166   Res = atom(Term),
167   VTo = VTo_.
168 wrap(Term, Res, VTo, VTo_) :- integer(Term) |
169   Res = integer(Term),
170   VTo = VTo_.
171 wrap(Term, Res, VTo, VTo_) :- float(Term) |
172   Res = floating_point(Term),
173   VTo = VTo_.
174 wrap(Term, Res, VTo, VTo_) :- Term=[T1|T2] |
175   wrap(T1, T1R, VTo, VTo1),
176   wrap(T2, T2R, VTo1, VTo_),

```

```

177     wait(T1R, R, [T1R | T2R]),
178     wait(T2R, Res, R).
179 wrap(Term, Res, VTo, VTo_) :- functor(Term,_,Ari), Term \= ^(x) |
180     wrap_functor(Term, WF, 1, Ari, VTo, VTo_),
181     wait(WF, Res, functor(WF)).
182 wrap(Term, Res, VTo, VTo_) :- vector(Term,Len) |
183     wrap_vector(Term, WV, 0, Len, VTo, VTo_),
184     wait(WV, Res, vector(WV)).
185 wrap(Term, Res, VTo, VTo_) :- string(Term,_,8) |
186     Res = string(Term),
187     VTo = VTo_.
188 wrap(Term, Res, VTo, VTo_) :- Term = ^(X) | % variable:unbound(X,{0,_,X})
189     VTo = [id(ID,X,in) | VTo_],
190     id_name(ID, S),
191     wait(S, Res, functor(^(variable(S)))).
192 alternatively.
193 wrap(Term, Res, VTo, VTo_) :- true | % variable:unbound(Term,{0,_,Term})
194     VTo = [id(ID,Term,out) | VTo_],
195     id_name(ID, S),
196     wait(S, Res, variable(S)).
197
198 wrap_functor(F, WF, I, Ari, VTo, VTo_) :- I > Ari |
199     WF = F,
200     VTo = VTo_.
201 wrap_functor(F, WF, I, Ari, VTo, VTo_) :- I =< Ari |
202     wrap(X, W, VTo, VTo1),
203     setarg(I, F, X, W, F1),
204     I1 := I + 1,
205     wrap_functor(F1, WF, I1, Ari, VTo1, VTo_).
206
207 wrap_vector(V, WV, I, Len, VTo, VTo_) :- I:=Len |
208     WV = V,
209     VTo = VTo_.
210 wrap_vector(V, WV, I, Len, VTo, VTo_) :- I < Len |
211     wrap(X, W, VTo, VTo1),
212     set_vector_element(V, I, X, W, V1),
213     I1 := I + 1,
214     wrap_vector(V1, WV, I1, Len, VTo1, VTo_).
215
216
217 /*****
218 * unwrap < varin server
219 *
220 * W : wrapped term to unwrap
221 * ^Res: unwrapped W
222 * VTi: variable_table server
223 */
224

```



```

225 unwrap(W, Res, VTi, VTi_) :- W=variable(S),
226     string(S,Len,8), Len > 1, string_element(S,0,#"-") |
227     name_id(S, ID),
228     wait(ID, VTi, [var(ID,Var,in) | VTi_]).
229 unwrap(W, Res, VTi, VTi_) :- W=functor(^(variable(S))),
230     string(S,Len,8), Len > 1, string_element(S,0,#"-") |
231     name_id(S, ID),
232     wait(ID, VTi, [var(ID,Var,out) | VTi_]).
233 unwrap(W, Res, VTi, VTi_) :- W=atom(A), atom(A) |
234     Res = A,
235     VTi = VTi_.
236 unwrap(W, Res, VTi, VTi_) :- W=integer(Z), integer(Z) |
237     Res = Z,
238     VTi = VTi_.
239 unwrap(W, Res, VTi, VTi_) :- W=floating_point(F), float(F) |
240     Res = F,
241     VTi = VTi_.
242 unwrap(W, Res, VTi, VTi_) :- W=list([W1|W2]) |
243     unwrap(W1, W1R, VTi, VTi1),
244     unwrap(W2, W2R, VTi1, VTi_),
245     wait(W1R, R, [W1R | W2R]),
246     wait(W2R, Res, R).
247 unwrap(W, Res, VTi, VTi_) :- W=functor(WF), functor(WF,_,Ari) |
248     unwrap_functor(WF, Res, 1, Ari, VTi, VTi_).
249 unwrap(W, Res, VTi, VTi_) :- W=vector(WV), vector(WV,Len) |
250     unwrap_vector(WV, Res, 0, Len, VTi, VTi_).
251 unwrap(W, Res, VTi, VTi_) :- W=string(S), string(S) |
252     Res = S,
253     VTi = VTi_.
254
255 unwrap_functor(WF, F, I, Ari, VTi, VTi_) :- I > Ari |
256     F = WF,
257     VTi = VTi_.
258 unwrap_functor(WF, F, I, Ari, VTi, VTi_) :- I =< Ari |
259     unwrap(W, X, VTi, VTi1),
260     setarg(I, WF, W, X, WF1),
261     I1 := I + 1,
262     unwrap_functor(WF1, F, I1, Ari, VTi1, VTi_).
263
264 unwrap_vector(WV, V, I, Len, VTi, VTi_) :- I:=Len |
265     V = WV,
266     VTi = VTi_.
267 unwrap_vector(WV, V, I, Len, VTi, VTi_) :- I < Len |
268     unwrap(W, X, VTi, VTi1),
269     set_vector_element(WV, I, W, X, WV1),
270     I1 := I + 1,
271     unwrap_vector(WV1, V, I1, Len, VTi1, VTi_).
272

```

```

273
274 id_name(ID, S) :- ID==0 |
275     S = "_0".
276 id_name(ID, S) :- ID > 0 |
277     id_name(ID, S, []).
278
279 id_name(N, S, L) :- N==0 |
280     new_string(S, ["_" | L], 8).
281 id_name(N, S, L) :- N > 0 |
282     D := N mod 16,
283     N1 := N / 16,
284     convert(D, C),
285     id_name(N1, S, [C | L]).
286
287 name_id(S, ID) :- string(S,Len,8), Len>1, string_element(S,0,#"_") |
288     name_id(S, 1, Len, ID, 0).
289 otherwise.
290 name_id(S, ID) :- true |
291     ID = -1.
292
293 name_id(S, I, Len, ID, N) :- I==Len |
294     ID = N.
295 name_id(S, I, Len, ID, N) :- I < Len |
296     string_element(S, I, C),
297     convert(C, D),
298     I1 := I + 1,
299     N1 := N*16 + D,
300     name_id(S, I1, Len, ID, N1).
301
302 % depending on ASCII character code
303
304 convert(I, N) :- true |
305     V = {
306     #0", #1", #2", #3", #4", #5", #6", #7",
307     #8", #9", #A", #B", #C", #D", #E", #F",
308     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
309     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
310     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0,
311     0, 16'A, 16'B, 16'C, 16'D, 16'E, 16'F, 0, 0, 0, 0, 0, 0, 0, 0,
312     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
313     0, 16'a, 16'b, 16'c, 16'd, 16'e, 16'f, 0, 0, 0, 0, 0, 0, 0, 0,
314     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
315     convert(V, I, N).
316
317 convert(V, I, N) :- vector(V, Len), 0 =< I, I < Len |
318     vector_element(V, I, N).
319
320 % end dklicio

```

ドライバを利用したコンパイルと実行結果は次の通りである。

```
% klic dklicio-test.kl1 dklicio.kl1 vart3.kl1 vart3data.c fusesusp.c
% ./a.out
server_socket
server
variable__table 0x805f96c newed side=1, free=1, used=-1, len=100
variable__table 0x8060850 newed side=0, free=0, used=-1, len=100
client
variable__table copied to 0x80785d8
variable__table copied to 0x80787b0
variable__table copied to 0x805f5cc
variable__table copied to 0x805f7a4
```

次にドライバ dklicio-test.kl1 を挙げる。

```
1  /*****
2  * dklicio driver
3  *
4  * 2002-01-21 TAKAGI Yusuke
5  */
6
7  :- module main.
8
9  :- public main/0.
10
11 main :- true |
12     _ = [closed],
13     Port = 65432,
14     dklicio:bind(inet(Port), Bind),
15     server(Bind),
16     main(Bind, Port),
17     true.
18
19 main(Bind, Port) :- wait(Bind) |
20     dklicio:connect(inet("localhost", Port), Connect),
21     client(Connect).
22
23 server(normal(SS)) :- true |
24     builtin:print(server_socket),
25     SS = [accept(R)],
26     server(R).
27 server(normal(I,0)) :- true |
28     builtin:print(server),
29     server(I, 0).
30
31 server(I, 0) :- I = [X|I1] |
```

```

32     builtin:print(server(X)),
33     server(I1).
34
35 client(normal(I,0)) :- true |
36     builtin:print(client),
37     O = [client, 2, "server", Message].

```

B.3 変数ネームサーバ

```

1  /*****
2  * vart: Variables Import/Export Table (cf. runtime/atomt.kl1)
3  *
4  * 2002-01-18 TAKAGI Yusuke
5  *
6  * Predicate:
7  *   new +Host +Side ?VTi ?VTo ^Repo
8  *     creates a new variable_table, for client or server Side versus Host.
9  *
10 * Message on VTo:
11 *   id -ID ?Var +Mode gets ID associated to Var of Mode.
12 *
13 * Messages on VTi:
14 *   var +ID ?Var +Mode gets Var of Mode identified by ID.
15 *
16 *   bind +ID +Term binds (ID,_) to Term, and removes (ID,_) .
17 *   fuse +ID +ID1 fuses (ID,_) with (ID1,_) , and removes (ID,_) .
18 *   remove +ID removes the entry (ID,_) .
19 *
20 * Repo from variable_table:
21 *   bound ID Term entry (ID,_) has been bound to Term.
22 *   fused ID ID1 entry (ID,_) has been fused with (ID1,_) .
23 *   removed ID entry (ID,_) has been removed.
24 */
25
26 :- module variable_table.
27
28 :- public new/5.
29
30 new(Host, Side, VTi, VTo, Repo) :- true |
31     generic:new(variable_table, T, Side),
32     vt(C, VTi, VTo, Ds, T, Host),
33     generic:new(merge, Ds, D),
34     repo(D, C, Repo),
35     true.
36
37
38 /*****

```

```

39 * vt sever
40 *
41 * C : close stream from repo
42 * VTi: stream from dklicio:varin
43 * VTo: id stream from dklicio:varout
44 * ^Ds : vector of merged Repo streams from fusesusp/bindhook
45 * T : table data object
46 */
47
48 vt(C, VTi, VTo, Ds, T, Host) :- C=[], VTi=[], VTo=[] |
49   Ds = [].
50 vt(C, VTi, VTo, Ds, T, Host) :- C=[close(ID)|C1], ID >= 0 |
51   generic:set_state(T, ID, closing, T1),
52   vt(C1, VTi, VTo, Ds, T1, Host).
53 vt(C, VTi, VTo, Ds, T, Host) :- VTo=[id(ID,X,in)|VTo1] |
54   generic:id(T, ID, X, Found),
55   vt(C, VTi, VTo1, Ds, T, Host, ID, X, Found, in, ^).
56 vt(C, VTi, VTo, Ds, T, Host) :- VTo=[id(ID,X,out)|VTo1] |
57   generic:id(T, ID, X, Found),
58   vt(C, VTi, VTo1, Ds, T, Host, ID, X, Found, out).
59 vt(C, VTi, VTo, Ds, T, Host) :- VTi=[var(ID,X,in)|VTi1], ID >= 0 |
60   generic:var(T, ID, X, Found),
61   vt(C, VTi1, VTo, Ds, T, Host, ID, X, Found, in).
62 vt(C, VTi, VTo, Ds, T, Host) :- VTi=[var(ID,X,out)|VTi1], ID >= 0 |
63   generic:var(T, ID, X, Found),
64   vt(C, VTi1, VTo, Ds, T, Host, ID, X, Found, out, ^).
65 vt(C, VTi, VTo, Ds, T, Host) :- VTi=[remove(ID)|VTi1], ID >= 0 |
66   generic:remove(T, ID, State, T1),
67   close(C, VTi1, VTo, Ds, T1, Host, ID, State).
68 vt(C, VTi, VTo, Ds, T, Host) :- VTi=[fuse(ID,ID1)|VTi1], ID >= 0, ID1 >= 0 |
69   generic:fuse(T, ID, ID1, State, T1),
70   close(C, VTi1, VTo, Ds, T, Host, ID, State).
71 vt(C, VTi, VTo, Ds, T, Host) :- VTi=[bind(ID,X)|VTi1], ID >= 0, wait(X) |
72   generic:bind(T, ID, X, State, T1),
73   close(C, VTi1, VTo, Ds, T1, Host, ID, State).
74
75 close(C, VTi, VTo, Ds, T, Host, ID, State) :- State = closing |
76   vt(C, VTi, VTo, Ds, T, Host).
77 close(C, VTi, VTo, Ds, T, Host, ID, State) :- State \= closing |
78   generic:set_state(T, ID, closing, T1),
79   vt(C, VTi, VTo, Ds, T1, Host, [removed(ID)]).
80
81 vt(C, VTi, VTo, Ds, T, Host, D) :- true |
82   Ds = {D, Ds1},
83   vt(C, VTi, VTo, Ds1, T, Host).
84
85 vt(C, VTi, VTo, Ds, T, Host, ID, X, Found, Mode) :- Found=true |
86   vt(C, VTi, VTo, Ds, T, Host).

```

```

87 vt(C, VTi, VTo, Ds, T, Host, ID, X, Found, Mode) :- Found=false |
88     addnew(C, VTi, VTo, Ds, T, Host, ID, X, D, Mode).
89
90 vt(C, VTi, VTo, Ds, T, Host, ID, X, Found, Mode, ^) :- Found=true |
91     new(C, VTi, VTo, Ds, T, Host, ID, X, Mode).
92 vt(C, VTi, VTo, Ds, T, Host, ID, X, Found, Mode, ^) :- Found=false |
93     addnew(C, VTi, VTo, Ds, T, Host, ID, X, D, Mode).
94
95 addnew(C, VTi, VTo, Ds, T, Host, ID, X, D, Mode) :- true |
96     generic:add(T, ID, X, open, T1),
97     new(C, VTi, VTo, Ds, T1, Host, ID, X, Mode).
98
99 new(C, VTi, VTo, Ds, T, Host, ID, X, Mode) :- Mode=in |
100     in(Host, ID, X, D),
101     vt(C, VTi, VTo, Ds, T, Host, D).
102 new(C, VTi, VTo, Ds, T, Host, ID, X, Mode) :- Mode=out |
103     out(Host, ID, X, D),
104     vt(C, VTi, VTo, Ds, T, Host, D).
105
106 in(Host, ID, X, D) :- ID >= 0 |
107     generic:new(fusesusp, X, ID@Host, D).
108
109 out(Host, ID, X, D) :- ID >= 0 |
110     generic:new(fusesusp, X, out, 0),
111     bindhook(ID@Host, 0, D).
112
113
114 /******
115 */
116
117 repo(D, C, Repo) :- D=[] |
118     C = [],
119     Repo = [].
120 repo(D, C, Repo) :- D=[Mes|D1], Mes=bound(ID,X), wait(X) |
121     repo(D1, C, Repo, ID, Mes).
122 repo(D, C, Repo) :- D=[Mes|D1], Mes=fused(ID,ID1), ID1 >= 0 |
123     repo(D1, C, Repo, ID, Mes).
124 repo(D, C, Repo) :- D=[Mes|D1], Mes=removed(ID) |
125     repo(D1, C, Repo, ID, Mes).
126
127 % repo(D, C, Repo) :-
128 %     D=[Mes|D1], Mes=fused(ID@Host,ID1@Host1), Host\=Host1 |
129 %     Repo = [Mes | Repo1],
130 %     repo(D, C, Repo1).
131
132 repo(D, C, Repo, ID, Mes) :- ID >= 0, functor(Mes,_,_) |
133     C = [close(ID) | C1],
134     Repo = [Mes | Repo1],

```

```

135     repo(D, C1, Repo1).
136
137
138  /*****
139  * bindhook server < out-mode fusesusp server
140  *
141  * ID  : associated to ID of variable_table versus remote Host
142  * X   : report stream from out-mode fusesusp
143  * ^Repo: report stream to variable_table
144  */
145
146  bindhook(ID@Host, X, Repo) :- X=[] |
147     Repo = [].
148  bindhook(ID@Host, X, Repo) :- X=(ID1@Host1), Host\=Host1 |
149     Repo = [exporting(ID)].
150  bindhook(ID@Host, X, Repo) :- X=(ID1@Host1), Host=Host1 |
151     Repo = [fused(ID, ID1)].
152  bindhook(ID@Host, X, Repo) :- X=[fused(out, ID1@Host1)|X1], Host\=Host1 |
153     bindhook(ID@Host, X1, Repo).
154  bindhook(ID@Host, X, Repo) :- X=[fused(out, ID1@Host1)|X1], Host=Host1 |
155     Repo = [fused(ID, ID1)].
156  bindhook(ID@Host, X, Repo) :- X=[bound(out, T)] |
157     builtin:print(bindhook:bound(ID@Host, T)),
158     Repo = [bound(ID, T)].
159
160
161  % end variable_table

```

B.4 変数表オブジェクト

```

1  /** vart: Distributed Variable Import/Export Table
2  *
3  * 2002-01-18 TAKAGI Yusuke
4  *
5  * variable_table: class
6  *
7  * new ?T +Side creates a new variable_table T for client or server Side.
8  *
9  * gc moves into new area.
10 *
11 * state +ID -State gets State of the entry (ID,_).
12 * set_state +ID +State -T sets State of the entry (ID,_).
13 *
14 * id -ID ?Var -Found gets ID associated to Var, or Found=false.
15 * var +ID ?Var -Found gets Var identified by ID, or Found=false.
16 *
17 * add +ID ?Var +State -T adds a new entry (ID,Var).

```

```

18  *
19  * remove +ID      -State -T  removes the entry (ID,_).
20  * fuse   +ID +ID1 -State -T  fuses (ID,_) with (ID1,_), and removes (ID,_).
21  * bind   +ID +Term -State -T  binds (ID,_) to Term, and removes (ID,_).
22  */
23
24  #include <stdio.h>
25  #include <string.h>
26  #include <klic/gdobject.h>
27  #include <klic/gd_macro.h>
28  #include "atom.h"
29  #include "funct.h"
30
31  #define GD_CLASS_NAME()  variable__table
32  #define GD_OBJ_TYPE     struct variable_table
33  #define GD_OBJ_SIZE(obj) (G_SIZE_IN_Q(GD_OBJ_TYPE))
34  #define STR  GD_CLASS_NAME_STRING " "
35
36  static const int SIZE = 100;
37  static const q CLOSED = makesym(atom_closed);
38
39  struct entry {
40      q var;
41      q state;
42  };
43
44  enum side { CLIENT=0, SERVER=1 };
45
46  GD_OBJ_TYPE {
47      struct data_object_method_table* method_table;
48      int side;
49      int free;
50      int used;
51      int len;
52      unsigned : 0; /* aligns tab */
53      struct entry tab[0]; /* tab[0..len-1] */
54  };
55
56  static int
57  tabsize(int len)
58  {
59      return len * sizeof(struct entry);
60  }
61
62  static void
63  print(GD_OBJ_TYPE* self)
64  {
65      debug_printf(STR "%p ", self);

```



```

66 }
67
68 GDDEF_METHOD(state_2)
69 {
70     G_STD_DECL;
71     int id;
72
73     print(GD_SELF);
74     GD_RETURN;
75 }
76
77 GDDEF_METHOD(set__state_3)
78 {
79     G_STD_DECL;
80     int id;
81
82     print(GD_SELF);
83     debug_printf("set_state #%k, %k = %p\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[2]);
84
85     id = intval(GD_ARGV[0]);
86     if(id < 0 || GD_SELF->len <= id)
87         GD_FAIL(STR "non-existent entry\n");
88     if(GD_SELF->tab[id].state==CLOSED)
89         GD_FAIL(STR "closed entry\n");
90
91     GD_SELF->tab[id].state = GD_ARGV[1];
92     GD_UNIFY(GD_ARGV[2], makefunctor(GD_SELF));
93     GD_RETURN;
94 }
95
96 static q
97 DEREf(q var)
98 {
99     while(isref(var)){
100         q tmp = derefone(var);
101         if(var==tmp || (isref(tmp) && var==derefone(tmp)))
102             return var;
103         var = tmp;
104     }
105     GD_FAIL(STR "variable is bound\n");
106 }
107
108 GDDEF_METHOD(id_3)
109 {
110     G_STD_DECL;
111     int id;
112     q var;
113

```

```

114     print(GD_SELF);
115     debug_printf("id (%k,%k) found %k\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[2]);
116
117     var = Deref(GD_ARGV[1]);
118     for(id=0; id < GD_SELF->len; id++){
119         if(GD_SELF->tab[id].state==CLOSED)
120             continue;
121
122         GD_SELF->tab[id].var = Deref(GD_SELF->tab[id].var);
123         if(var==GD_SELF->tab[id].var){
124             GD_UNIFY(GD_ARGV[0], makeint(id));
125             GD_UNIFY(GD_ARGV[2], makesym(atom_true));
126             GD_RETURN;
127         }
128     }
129     if(id==GD_SELF->len){
130         GD_UNIFY(GD_ARGV[0], makeint(GD_SELF->free));
131         GD_UNIFY(GD_ARGV[2], makesym(atom_false));
132     }
133     debug_printf("id (%k,%k) found %k\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[2]);
134     GD_RETURN;
135 }
136
137 GDDEF_METHOD(var_3)
138 {
139     G_STD_DECL;
140     int id;
141
142     print(GD_SELF);
143     debug_printf("var (%k,%k) found %k\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[2]);
144
145     id = intval(GD_ARGV[0]);
146     if(id < GD_SELF->len && GD_SELF->tab[id].state != CLOSED){
147         GD_UNIFY(GD_ARGV[1], GD_SELF->tab[id].var);
148         GD_UNIFY(GD_ARGV[2], makesym(atom_true));
149     }else{
150         GD_UNIFY(GD_ARGV[2], makesym(atom_false));
151     }
152     GD_RETURN;
153 }
154
155 static void
156 init(GD_OBJ_TYPE* self, int id)
157 {
158     self->tab[id].var = makeref(NULL);
159     self->tab[id].state = CLOSED;
160 }
161

```

```

162 static q*
163 SET_NEWOBJ(q* g_allocp, GD_OBJ_TYPE* GD_SELF, GD_OBJ_TYPE* newself,
164 long g_method_functor, q* GD_ARGV, int len, int size )
165 {
166     declare_globals;
167     GDSET_NEWOBJ(newself);
168     g_allocp += tabsize(len);
169
170     newself->len = len;
171     memcpy(newself->tab, GD_SELF->tab, size);
172     return g_allocp;
173 }
174
175 GDDEF_METHOD(add_4)
176 {
177     G_STD_DECL;
178     GD_OBJ_TYPE* newself;
179     int id;
180
181     print(GD_SELF);
182     debug_printf("add (%k,%k) %k = %p\n",
183         GD_ARGV[0], GD_ARGV[1], GD_ARGV[2], GD_ARGV[3] );
184
185     id = intval(GD_ARGV[0]);
186     if(id < GD_SELF->len){
187         newself = GD_SELF;
188     }else{
189         int len = GD_SELF->len * 2;
190         int id;
191
192         g_allocp = SET_NEWOBJ(g_allocp, GD_SELF, newself,
193             g_method_functor, GD_ARGV, len, tabsize(GD_SELF->len) );
194         for(id=GD_SELF->len; id < len; id++)
195             init(newself, id);
196         newself->side = GD_SELF->side;
197     }
198     newself->used = (id > GD_SELF->used ? id : GD_SELF->used);
199     newself->tab[id].var = GD_ARGV[1];
200     newself->tab[id].state = GD_ARGV[2];
201     debug_printf(" + used=%d, tab[%d]=%k, %k\n",
202         newself->used, id,
203         newself->tab[id].var, newself->tab[id].state);
204
205     if(id==newself->free){
206         int i;
207         for(i=id+2; i < newself->len; i+=2)
208             if(newself->tab[i].state==CLOSED)
209                 break;

```

```

210     newself->free = i;
211 }
212 GD_UNIFY(GD_ARGV[3], makefunctor(newself));
213
214 print(GD_SELF);
215 debug_printf("+ (%k,%k) = %p\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[3]);
216 GD_RETURN;
217 }
218
219 static q*
220 CLOSE(q* g_allopc, GD_OBJ_TYPE* self, long method, int argc, q* argv)
221 {
222     GD_OBJ_TYPE* newself = self;
223     int id = intval(argv[0]);
224     int len;
225
226     if(id < 0 || self->len <= id)
227         GD_FAIL(STR "closed a non-existent entry\n");
228     GD_UNIFY(argv[argc-2], makesym(self->tab[id].state));
229     init(self, id);
230
231     if(id < self->free)
232         self->free = id;
233     if(id==self->used){
234         for( ; id >= 0; id--)
235             if(self->tab[id].state != CLOSED)
236                 break;
237         self->used = id;
238     }
239
240     len = self->len / 2;
241     if(SIZE < self->used && self->used < len)
242         g_allopc = SET_NEWOBJ(g_allopc, self, newself,
243             method, argv, len, tabsize(len) );
244
245     GD_UNIFY(argv[argc-1], makefunctor(newself));
246     return g_allopc;
247 }
248
249 GDDEF_METHOD(remove_3)
250 {
251     G_STD_DECL;
252
253     print(GD_SELF);
254     debug_printf("- #%k (%k) = %p\n", GD_ARGV[0], GD_ARGV[1], GD_ARGV[2]);
255
256     g_allopc = CLOSE(g_allopc, GD_SELF, g_method_functor, 3, GD_ARGV);
257     GD_RETURN;

```

```

258 }
259
260 GDDEF_METHOD(fuse_4)
261 {
262     G_STD_DECL;
263     int id, id1;
264
265     print(GD_SELF);
266     debug_printf("fuse #%k (%k) with #%k\n", GD_ARGV[0], GD_ARGV[2], GD_ARGV[1]);
267
268     id = intval(GD_ARGV[0]);
269     id1 = intval(GD_ARGV[1]);
270     if(id1 < 0 || GD_SELF->len <= id1)
271         GD_FAIL(STR "fused with non-existent entry\n");
272     GD_UNIFY(GD_SELF->tab[id].var, GD_SELF->tab[id1].var);
273     g_allocp = CLOSE(g_allocp, GD_SELF, g_method_functor, 4, GD_ARGV);
274     GD_RETURN;
275 }
276
277 GDDEF_METHOD(bind_4)
278 {
279     G_STD_DECL;
280     int id;
281
282     print(GD_SELF);
283     debug_printf("bind #%k (%k) to %k\n", GD_ARGV[0], GD_ARGV[2], GD_ARGV[1]);
284
285     id = intval(GD_ARGV[0]);
286     debug_printf(" bind tab[%d]=%k, %k to %k\n",
287         id, GD_SELF->tab[id].var, GD_SELF->tab[id].state, GD_ARGV[1] );
288     GD_UNIFY(GD_SELF->tab[id].var, GD_ARGV[1]);
289     g_allocp = CLOSE(g_allocp, GD_SELF, g_method_functor, 4, GD_ARGV);
290     GD_RETURN;
291 }
292
293 GDDEF_GENERIC()
294 {
295     G_STD_DECL;
296     GD_SWITCH_ON_METHOD{
297         //GD_METHOD_CASE(state_2);
298         GD_METHOD_CASE(set__state_3);
299         GD_METHOD_CASE(id_3);
300         GD_METHOD_CASE(var_3);
301         GD_METHOD_CASE(add_4);
302         GD_METHOD_CASE(remove_3);
303         GD_METHOD_CASE(fuse_4);
304         GD_METHOD_CASE(bind_4);
305         GD_METHOD_CASE_DEFAULT;

```

```

306     }
307     GD_RETURN;
308 }
309
310 GDDEF_PRINT()
311 {
312     G_STD_DECL;
313     print(GD_SELF);
314     GD_RETURN_FROM_PRINT;
315 }
316
317 GDDEF_GC()
318 {
319     G_STD_DECL;
320     GD_OBJ_TYPE* newself;
321     int len;
322
323     GDSET_NEWOBJ_IN_NEWGEN(newself);
324     newself->side = GD_SELF->side;
325     newself->free = GD_SELF->free;
326     newself->used = GD_SELF->used;
327     len = newself->len = GD_SELF->len;
328     memcpy(newself->tab, GD_SELF->tab, tabsize(len));
329
330     debug_printf(STR "copied to %p\n", newself);
331     GD_RETURN_FROM_GC(newself);
332 }
333
334 #if HERE_IS_PSEUDO_CODE
335 struct list {
336     GD_OBJ_TYPE* table;
337     struct list* next;
338 };
339
340 /* All 'variable_table's in localhost
341  * Each GDDEF_NEW updates this */
342 static struct list* tables;
343
344 /* Following two functions need 'register_after_gc_hook'ed
345  * when GDDEF_NEW is called first time */
346
347 static void remove_variable_table()
348 {
349     GD_OBJ_TYPE* table;
350     for(table=tables->table; table != NULL; table=table->next){
351         if(removed(table)) update(tables);
352     }
353 }

```

```

354
355 static void gc_variable_tables()
356 {
357     GD_OBJ_TYPE* table;
358     for(table=tables->table; table != NULL; table=table->next){
359         for(int i=0; i < table->len; i++){
360             if(copied(table->tab[i])) table->tab[i] = pointer_to_new_area_directly;
361             else if(closing(table->tab[i])) copy(table->tab[i]);
362             else table->tab[i] = NULL;
363         }
364     }
365 }
366 #endif /* PSEUDO_CODE */
367
368 #define GDUSE_MY_GENERIC
369 #define GDUSE_MY_PRINT
370 #define GDUSE_MY_GC
371
372 #include <klic/gd_methhtab.h>
373
374 GDDEF_NEW()
375 {
376     GD_STD_DECL_FOR_NEW;
377     GD_OBJ_TYPE* newself;
378     int len = SIZE;
379     int id;
380
381     if(GD_ARGC != 1)
382         GD_ERROR_IN_NEW(STR "new ?VariableTable +Side");
383
384     GDSET_NEWOBJ_FOR_NEW(newself, GD_OBJ_SIZE(newself) + tabsize(len));
385
386     newself->side = (GD_ARGV[0]==makesym(atom_client) ? CLIENT : SERVER);
387     newself->free = newself->side;
388     newself->used = -1;
389     newself->len = len;
390     for(id=0; id < len; id++)
391         init(newself, id);
392
393     print(newself);
394     debug_printf("newed side=%d, free=%d, used=%d, len=%d\n",
395         newself->side, newself->free, newself->used, newself->len );
396     GD_RETURN_FROM_NEW(newself);
397 }

```

次のコマンドでコンパイルと実行確認を行った。

```
% klic vart3data-test.kl1 vart3data.c
```

```

% ./a.out
variable__table 0x804e4d8 newed side=0, free=0, used=-1, len=100
variable__table 0x804e4d8 id (0,_32B) found _32C
id (0,_32B) found false
variable__table 0x804e4d8 add (0,_32B) open = 0x804f17c
+ used=0, tab[0]=_32B, open
variable__table 0x804e4d8 + (0,_32B) = 0x804f17c
variable__table 0x804e4d8 id (2,_330) found _338
id (2,_330) found false
variable__table 0x804e4d8 add (2,_330) open = 0x804f1ac
+ used=2, tab[2]=_330, open
variable__table 0x804e4d8 + (2,_330) = 0x804f1ac
variable__table 0x804e4d8 id (4,_333) found _33A
id (4,_333) found false
variable__table 0x804e4d8 add (4,_333) open = 0x804f1b4
+ used=4, tab[4]=_333, open
variable__table 0x804e4d8 + (4,_333) = 0x804f1b4
vt(=(0,_32B),=(2,_330),=(4,_333))
variable__table 0x804e4d8 set_state #4, closing = 0x804f1c0
variable__table 0x804e4d8 var (0,_350) found _351
var(0,_32B,true)
vt(=(0,_32B),=(2,_330),=(4,_333))
variable__table 0x804e4d8 fuse #2 (_362) with #0
variable__table 0x804e4d8 fuse(2,0,(null),)
vt(=(0,_32B),=(2,_32B),=(4,_333))
variable__table 0x804e4d8 bind #0 (_374) to [{i,3,2},"sdgl"]
bind tab[0]=_32B, open to [{i,3,2},"sdgl"]
variable__table 0x804e4d8 bind(0, [{i,3.200000},"sdgl"],(null),)
vt(=(0, [{i,3,2},"sdgl"]),=(2, [{i,3,2},"sdgl"]),=(4,_333))
variable__table 0x804e4d8 - #4 (_385) = 0x804f2e0
variable__table 0x804e4d8 remove(4,(null),)
vt(=(0, [{i,3,2},"sdgl"]),=(2, [{i,3,2},"sdgl"]),=(4,_333))
gc(922,27)
vt(=(0, [{i,3,2},"sdgl"]),=(2, [{i,3,2},"sdgl"]),=(4,_11))

```

次にドライバ vart3data-test.kl1 を挙げる。

```

1  /*****
2  * vart3data driver
3  *
4  * 2002-01-21 TAKAGI Yusuke

```



```

5  */
6
7  :- module main.
8
9  :- public main/0.
10
11 main :- true |
12     _ = [client, server, closed],
13     generic:new(variable_table, T, client),
14     main(T, 0, 2, 4).
15
16 main(T, ID1, ID2, ID3) :- wait(T) |
17     VT = (ID1=X1),
18     generic:generic(T, id(ID1, X1, F1)),
19     generic:generic(T, add(ID1, X1, open, T1)),
20     add2(T1, ID1, ID2, ID3, VT).
21
22 add2(T, ID1, ID2, ID3, VT) :- wait(T) |
23     VT1 = vt(VT, ID2=X2, ID3=X3),
24     generic:generic(T, id(ID2, X2, F2)),
25     generic:generic(T, add(ID2, X2, open, T2)),
26     generic:generic(T2, id(ID3, X3, F3)),
27     generic:generic(T2, add(ID3, X3, open, T3)),
28     builtin:print(VT1),
29     generic:generic(T3, set_state(ID3, closing, T4)),
30     main(T4, ID1, ID2, ID3, VT1).
31
32 main(T, ID1, ID2, ID3, VT) :- wait(T), Var=var(ID1, X, Found) |
33     generic:generic(T, Var),
34     % generic:generic(T, var(ID1, X, Found)),
35     % generic:var(T, ID1, X, Found),
36     builtin:print(Var),
37     builtin:print(VT),
38     fuse(T, ID1, ID2, ID3, VT).
39
40 fuse(T, ID1, ID2, ID3, VT) :- wait(T) |
41     Fuse = fuse(ID2, ID1, Closing, T5),
42     % generic:generic(T, Fuse),
43     generic:fuse(T, ID2, ID1, Closing, T5),
44     builtin:print(Fuse),
45     builtin:print(VT),
46     bind(T5, ID1, ID2, ID3, VT).
47
48 bind(T, ID1, ID2, ID3, VT) :- wait(T) |
49     Bind = bind(ID1, [{i,3.2},"sdgl"], Closing1, T6),
50     % generic:generic(T, Bind),
51     generic:bind(T, ID1, [{i,3,2},"sdgl"], Closing1, T6),
52     builtin:print(Bind),

```

```

53     builtin:print(VT),
54     remove(T6, ID1, ID2, ID3, VT).
55
56 remove(T, ID1, ID2, ID3, VT) :- wait(T) |
57     Remove = remove(ID3, Closing2, T7),
58     % generic:generic(T, Remove),
59     generic:remove(T, ID3, Closing2, T7),
60     builtin:print(Remove),
61     builtin:print(VT),
62
63     system_control:gc(B,A),
64     builtin:print(gc(B,A)),
65     builtin:print(VT).

```

B.5 fusesusp

```

1  /*****
2  * fusesusp: Generator Object senses unification
3  *
4  * 2002-01-16 TAKAGI Yusuke
5  *
6  * new ?X +ID ^Repo: creates in-mode X
7  * new ?X +out ^Repo: creates out-mode X
8  *
9  * bind: reports bound(ID, Val).
10 * unify: in=in mode error.
11 *     in=out reports fused.
12 *     out=out merges both hooks.
13 *     others merges hooks.
14 * remove: reports removed(ID).
15 */
16
17 #include <stdio.h>
18 #include <klic/gcobject.h>
19 #include <klic/susp.h>
20 #include <klic/gg_macro.h>
21 #include "atom.h"
22 #include "funct.h"
23
24 #define GG_CLASS_NAME() fusesusp
25 #define GG_OBJ_TYPE struct fusesusp
26 #define BINDHOOK predicate_variable__table_xbindhook_3
27
28 extern const struct predicate BINDHOOK;
29 static const struct predicate* bindhook = &BINDHOOK;
30
31 struct fusesusp {

```

```

32     struct generator_object_method_table* method_table;
33     q id;
34     q repo;
35     q susp;
36 };
37
38 static void
39 PRINT(GG_OBJ_TYPE* self)
40 {
41     debug_printf(GG_CLASS_NAME_STRING " %k ", self->id);
42 }
43
44 static q*
45 REPO(q* g_allocp, GG_OBJ_TYPE* self, unsigned long pf, q arg)
46 {
47     declare_globals;
48     struct cons* repo;
49     struct functor* mes;
50     int i;
51
52     GG_TRY_TO_ALLOC(mes, (struct functor*), 3, gc_request);
53     mes->functor = makesym(pf);
54     mes->args[0] = self->id;
55     mes->args[1] = arg;
56
57     GG_TRY_TO_ALLOC(repo, (struct cons*), 2, gc_request);
58     repo->car = makefunctor(mes);
59     GGSET_VAR(repo->cdr);
60
61     GG_UNIFY(self->repo, makecons(repo));
62     self->repo = repo->cdr;
63     return g_allocp;
64
65     gc_request:
66     PRINT(self);
67     fatal("failed to allocate Repo");
68 }
69
70 static q*
71 unify_hooks(q* g_allocp, GG_OBJ_TYPE* self, q susp)
72 {
73     struct susprecp* susprecp = suspp(susp);
74     struct hook* top = susprecp->u.first_hook.next;
75     struct hook* hookp;
76     for(hookp = top->next; hookp != top; hookp = hookp->next){
77         union goal_or_consumer u = hookp->u;
78         if( u.l != OL && !is_consumer_hook(u) &&
79             isint(u.g->next) && u.g->pred==bindhook ){

```

```

80     g_allocp = REPO(g_allocp, self, functor_fused_2, u.g->args[0]);
81     }
82     }
83     return g_allocp;
84 }
85
86 static q*
87 unify_out(q* g_allocp, GG_OBJ_TYPE* self, GG_OBJ_TYPE* other)
88 {
89     if(derefone(other->repo) != other->repo)
90         g_allocp = unify_hooks(g_allocp, self, other->repo);
91
92     return g_allocp;
93 }
94
95 static q*
96 unify_susp(q* g_allocp, GG_OBJ_TYPE* self, q GG_TERM)
97 {
98     GG_OBJ_TYPE* other = (GG_OBJ_TYPE*)
99     untag_generator_susp(generator_suspp(derefone(GG_TERM))->u.o);
100    if(self->method_table != other->method_table){
101        g_allocp = unify_hooks(g_allocp, self, GG_TERM);
102        debug_printf("fused with other susp %k\n", GG_TERM);
103        GG_UNIFY(self->susp, GG_TERM);
104    }else{
105        const q out = makesym(atom_out);
106        if(self->id != out){
107            if(other->id != out) fatal("two in-mode fusesusps fused\n");
108            debug_printf("in fused with %k\n", other->id);
109            GG_UNIFY(self->id, other->repo);
110            g_allocp = unify_out(g_allocp, self, other);
111        }else if(other->id != out){
112            debug_printf("fused with in %k\n", other->id);
113            GG_UNIFY(self->repo, other->id);
114            g_allocp = unify_out(g_allocp, other, self);
115        }else{
116            debug_printf("fused with %k\n", other->id);
117            g_allocp = unify_out(g_allocp, self, other);
118            GG_UNIFY(self->repo, other->repo);
119        }
120    }
121    return g_allocp;
122 }
123
124 GGDEF_UNIFY()
125 {
126     G_STD_DECL;
127     GG_OBJ_TYPE* self;

```

```

128
129     self = (GG_OBJ_TYPE*)
130         untag_generator_susp(generator_suspp(derefone(GG_SELF))->u.o);
131     PRINT(self);
132     GG_SWITCH_ON_TERM(list0, atomic0, functor0, data0, susp0);
133
134     list0:
135     atomic0:
136     functor0:
137     data0:
138         debug_printf("bound to %k\n", GG_TERM);
139         g_allocp = REPO(g_allocp, self, functor_bound_2, GG_TERM);
140         GG_UNIFY(self->repo, NILATOM);
141         GG_UNIFY(self->susp, GG_TERM);
142
143         derefone(GG_SELF) = GG_TERM;
144         self->method_table = NULL;
145         GG_TERMINATE;
146
147     susp0:
148         g_allocp = unify_susp(g_allocp, self, GG_TERM);
149         GG_TERMINATE;
150 }
151
152 GGDEF_GENERATE()
153 {
154     G_STD_DECL;
155     PRINT(GG_SELF);
156     debug_printf("generate\n");
157     return GG_SELF->susp;
158 }
159
160 GGDEF_GC()
161 {
162     G_STD_DECL;
163     GG_OBJ_TYPE* newself;
164
165     PRINT(GG_SELF);
166     debug_printf("copied\n");
167
168     GGSET_NEWOBJ_IN_NEWGEN(newself);
169     G_COPY_KL1_TERM_TO_NEWGEN(GG_SELF->id , newself->id);
170     G_COPY_KL1_TERM_TO_NEWGEN(GG_SELF->repo, newself->repo);
171     G_COPY_KL1_TERM_TO_NEWGEN(GG_SELF->susp, newself->susp);
172
173     GG_RETURN_FROM_GC(newself);
174 }
175

```

```

176 GGDEF_DEALLOCATE()
177 {
178     G_STD_DECL;
179
180     PRINT(GG_SELF);
181     debug_printf("removed\n");
182
183     g_allocp = REPO(g_allocp, GG_SELF, functor_removed_1, NULL);
184     GG_UNIFY(GG_SELF->repo, NILATOM);
185
186     GG_SELF->method_table = (void*) NILATOM;
187     GG_RETURN;
188 }
189
190 GGDEF_PRINT()
191 {
192     G_STD_DECL;
193     PRINT(GG_SELF);
194     debug_printf("print\n");
195     GG_RETURN_FROM_PRINT;
196 }
197
198 #define GGUUSE_MY_UNIFY
199 #define GGUUSE_MY_GENERATE
200 #define GGUUSE_MY_GC
201 #define GGUUSE_MY_DEALLOCATE
202 #define GGUUSE_MY_PRINT
203
204 #include <klic/gg_methstab.h>
205
206 GGDEF_NEW()
207 {
208     GG_STD_DECL_FOR_NEW;
209     GG_OBJ_TYPE* newself;
210     q var;
211
212     if(GG_ARGC != 2)
213         fatal(GG_CLASS_NAME_STRING ":new ?X +ID ^Repo");
214
215     GGSET_NEWOBJ_FOR_NEW(newself, (GG_OBJ_TYPE*));
216     newself->id = GG_ARGV[0];
217     newself->repo = GG_ARGV[1];
218     GGSET_VAR(newself->susp);
219     register_for_deallocation(newself, G_GENERATOR);
220
221     var = GG_MAKE_HOOK_VAR((struct generator_object*) newself);
222     GG_RETURN_FROM_NEW(var);
223 }

```

```
224
225 /* end fusesusp */
```

次のコマンドでコンパイルと実行確認を行った。

```
% klic fusesusp-test.kl1 fusesusp.c vart3.kl1 vart3data.c
% ./a.out
xyz(fusesusp(@(3,local),_63FD,_14),\
  fusesusp(@(4,remote),_63FA,_19),fusesusp(out,_63F7,_48))
fusesusp @(3,local) copied
fusesusp @(4,remote) copied
fusesusp out copied
gc(99,80)
xyz(fusesusp(@(3,local),_D,_14),\
  fusesusp(@(4,remote),_1A,_21),fusesusp(out,_27,_30))
fusesusp out bound to 24
fusesusp @(3,local) bound to 33
:(bindhook,bound(@(5,host),24))
:(bindhook,bound(@(6,node),24))
xyz(fusesusp(@(3,local),33,[bound(@(3,local),33)]),\
  fusesusp(@(4,remote),_1A,_21),fusesusp(out,24,[bound(out,24)]))
```

次にドライバ fusesusp-test.kl1 を挙げる。

```
1 /** test fusesusp
2 *
3 * 2002-01-18 TAKAGI Yusuke
4 */
5
6 :- module main.
7
8 :- public main/0.
9
10 main :- true |
11   _ = [out, bound(_,_), fused(_,_), removed(_)],
12   generic:new(fusesusp, X, 3@local, XR),
13   generic:new(fusesusp, Y, 4@remote, YR),
14   generic:new(fusesusp, Z, out, ZR),
15   variable_table:bindhook(5@host, ZR, R5),
16   variable_table:bindhook(6@node, ZR, R6),
17   %Z = 32,
18   %X = Z,
19   %Z = X,
20   %Y = X,
21   D = xyz(fusesusp(3@local, X, XR),
```

```

22     fusesusp(4@remote, Y, YR), fusesusp(out, Z, ZR)),
23     builtin:print(D),
24     gc(D).
25
26 gc(D) :- true |
27     system_control:gc(B, A),
28     builtin:print(gc(B, A)),
29     builtin:print(D),
30     %remove(D).
31     bind(D).
32
33 bind(D) :- D = xyz(fusesusp(_,X,_), fusesusp(_,Y,_), fusesusp(_,Z,ZR)) |
34     Z = 24,
35     %Z = X,
36     %Y = X,
37     X = 33,
38     builtin:print(D).
39
40 remove(D) :- D = xyz(X, Y, Z) |
41     remove(_X, _Y, _Z).
42
43 remove(X, Y, Z) :- true |
44     system_control:gc(B, A),
45     builtin:print(gc(B, A)),
46     builtin:print(xyz(X, Y, Z)).
47     klicio:klicio([stdout(normal([putt(X), nl, putt(Y), nl, putt(Z)])))]).

```


付録C DKLIC 処理系の応用例

C.1 Remote Predicate Call

dklicio を利用して遠隔述語呼び出しを行うクライアント/サーバ系は次のようになる。

まず、RPC クライアントを挙げる。

```
:- module rpc_client.  
  
new(Host, Port, RPC) :- true |  
    dklicio:connect(inet(Host, Port), R),  
    new1(RPC, R).  
new1(RPC, R) :- R = normal(Sock) |  
    RPC = Sock.
```

この RPC クライアントを利用するユーザプロセスは、ストリーム RPC に遠隔呼び出ししたいゴールを次のように書き込めば良い。

```
RPC = [append(X, Y, ^(Z)), chat(^(I), 0) | RPC1],
```

結果をもらうための in モードの変数を $\wedge/1$ の引数としていることに注意せよ。この表記は、Janus [22] から借りたものである。

手続き型言語と異なり、これらの分散変数は単純な整数や文字列だけでなく変数を含む複雑な KL1 項としての値を取ることができる。

次に RPC サーバを挙げる。

```
:- module rpc_server.  
  
new(Port) :- true |  
    dklicio:bind(inet(Port), R),  
    new1(R).  
new1(R) :- R = normal(ServSock) |  
    accept(ServSock).  
  
accept(ServSock) :- true |
```

```

ServSock = [accept(R) | ServSock1],
accept1(ServSock1, R).
accept1(ServSock,R) :- R=normal(Socket) |
    rpcd(Socket),
    accept(ServSock).

```

```

rpcd(Socket) :- Socket = [M:Call|Socket1] |
    functor(Call, Name, Arity),
    generic:new(module, Modu, M),
    generic:new(predicate,
        Pred, Modu, Name, Arity),
    args_vector(Call, Arity, V, []),
    apply(Pred, V).

```

```

args_vector(F,Arity,V,L) :- Arity==0 |
    new_vector(V, L).
args_vector(F,Arity,V,L) :- Arity > 0 |
    arg(Arity, F, Arg),
    L1 = [Arg | L],
    Arity1 := Arity - 1,
    args_vector(F, Arity1, V, L1).

```

C.2 検索サーバ

DKLIC におけるネームサーバの例を次に挙げる。

```

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ユーザ定義サーバへの通信路 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 と名付けられたユーザ定義サーバへの通信路 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ユーザ定義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ユーザ定義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、ユーザ定義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 サーバを通じてユーザ定義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。

dklicio を用いてホスト間通信を行っていれば name サーバがユーザ定義サーバの仲介検索を終えた後、name サーバの中継、従ってホスト B の中継は行われない。