第 I 編 参考文献

[1] エドワード・ファイゲンバウム, パメラ・マコーダック著, 木村繁訳：第五世代コンピュータ 日本の挑戦, TBSブリタニカ (1983)

[2] 渕 一博, 廣瀬 健：第５世代コンピュータの計画, 海鳴社 (1984)

[3] 廣瀬 健, 渕 一博：第５世代コンピュータの文化, 海鳴社 (1984)

[4] 渕 一博, 赤木 昭夫：第５世代コンピュータを創る — 渕 一博に聞く —, 日本放送出版協会 (1984)

[5] 上前 淳一郎：ジャパニーズ・ドリーム — 未知の森へ第五世代コンピュータ, 講談社 (1985)

[6] 松尾 博志：スーパー頭脳集団 電総研, コンピュータ・エージ社 (1987)

[7] 上前 淳一郎：めざすは新世代コンピュータ, 角川文庫 (1988)

[8] 今岡 和彦：我が志の第五世代コンピュータ — 渕一博とICOTの技術戦士たち, TBSブリタニカ (1989)

[9] *Proc. FGCS'84 : International Conference on Fifth Generation Computer Systems,* Tokyo (November 1984), also reprinted and published by Ohmsha

[10] *Proc. FGCS'88 : International Conference on Fifth Generation Computer Systems,* Tokyo (November 1988), also reprinted and published by Ohmsha

[11] *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* Tokyo (June 1992), also reprinted and published by Ohmsha

[12] Kurozumi, T. : Overview of the Ten Years of the FGCS Project, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* pp.9–19, Tokyo (June 1992)

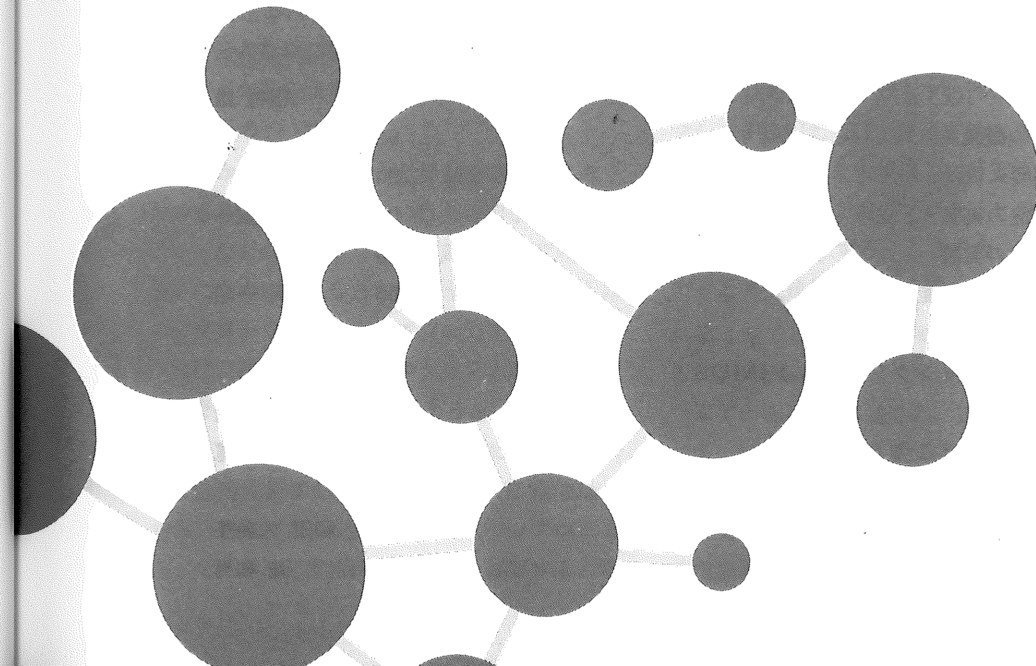
[13] Furukawa, K. : Summary of Basic Research Activities of the FGCS Project, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* pp.20–32 (1992)

[14] Uchida, S. : Summary of the Parallel Inference Machine and its Basic Software, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems,* pp.33–49 (1992)

第 II 編

技術編

—第五世代コンピュータの並列処理技術—



第 II 編　目次および執筆者

第 1 章　第五世代コンピュータの並列処理技術	55	瀧 和男
1.1　はじめに	55	
1.2　大目標とアプローチ	56	
1.3　どんな並列処理を目指すか	57	
第 2 章　KL1 言語	61	近山 隆
2.1　はじめに	61	
2.2　KL1 言語とは	61	
2.3　実行機構の概要	62	
2.4　KL1 で扱えるデータ	66	
2.5　プロセスとストリーム通信	69	
2.6　プロセス・ネットワーク	75	
2.7　プログラム動作の指定	80	
2.8　一階述語論理と KL1	82	
2.9　むすび	85	
第 3 章　ハードウェア	86	瀧 和男
3.1　はじめに	86	
3.2　マシンアーキテクチャの特徴	86	
3.3　五つの PIM モデルの概要	87	
3.4　要素プロセッサ	90	
3.5　ネットワーク	90	
3.6　キャッシュシステム	91	
第 4 章　言語処理系の実装	92	
4.1　はじめに	92	稲村 雄
4.2　言語処理系の概説	92	
4.3　処理方式概要	94	
4.4　基本言語機能の実装	99	近藤 誠一
4.5　拡張言語機能の実装	103	
4.6　効率向上のための機能と実現	106	六沢 一昭
4.7　ノード間処理の実現	112	
第 5 章　並列オペレーティングシステム PIMOS	125	
5.1　はじめに	125	近山 隆
5.2　既存の OS と違うところ	127	屋代 寛
5.3　特徴的な実装	129	
5.4　プログラム開発環境	137	和田 久美子
第 6 章　むすび	146	瀧 和男

第 1 章

第五世代コンピュータの並列処理技術
──何を目指すか──

1.1　はじめに

第 II 編では，いよいよ第五世代コンピュータの並列処理に関する技術面に踏み込んでゆきたい．

第五世代コンピュータの並列処理技術は，そもそも，近い将来の高度知識情報処理を実現するためになくてはならない超高速推論処理を実現するための技術として，研究開発がスタートした．それは，中核となる並列論理型言語と，それを高速実行するハードウェア，ソフトウェア技術として研究が進められ成長していった．同時にそこで生まれた技術は，論理型言語や推論処理に限って役立つものではなく，より広範囲な大規模並列処理に広く適用できる基礎技術であるということが，研究開発をとおしてしだいに明らかになってきた．しかも，そのなかには海外の研究では試されたことのない文字どおりの世界最先端技術がいくつも含まれているのである．

それらのなかには，研究者が世界初を意識して取り組んだものもあれば，意識しないうちに自然に生まれ成長していったものもある．論理型言語を研究の道具として用いたためにそうなったという面が多いように思われる．ソフトウェア，ハードウェアすべての研究の中核に論理型言語を据えたプロジェクト創始者たちの洞察の深さに敬服するばかりである．

第五世代コンピュータの並列処理技術には，ハードウェアから応用技術まで含まれるが，第 II 編ではそのなかから，言語，言語処理系，OS，ハードウェアを中心に，システム実現技術の全体像を描写することを試みる．

そのためには，ここで取り扱う並列処理が世の中の並列処理全般のなかでどのような位置を占めるのか，何を目指しどのような特徴をもっているのか，ほかとどう異なり，どう同じなのかを知っておく必要がある．第 1 章では主にこの辺りのことを説明し，第 II 編の導入部とする．

そして第 2 章以下では，第五世代コンピュータを構成する主要技術を順を追って見ていく．はじめは KL1 言語である．第五世代コンピュータの並列処理が，知識処理をはじめとするいくつもの分野で大きな成果を上げることができたのは，KL1 言語の言語仕様に負うところが大きい．第 2 章では KL1 言語の言語仕様と特徴，基本的なプログラミング技法について解説する．

第 3 章ではハードウェアについて述べる．並列推論マシン PIM は，KL1 言語を効率よく実行するために特別に設計された大規模並列計算機である．最も大きいモデルでは 512 台のプロセッサが協力して，一つの問題を並列計算する．ここでは，PIM のアーキテクチャと実現技術の概要を述べる．

第 4 章では KL1 言語処理系の実装について説明する．並列推論マシン上では OS を含むすべてのソフトウェアが KL1 で記述され，KL1 言語処理系で実行される．KL1 言語処理系には，高度な機能をもつ KL1 言語を大規模並列計算機上に効率よく実現するための技術上のエッセンスがぎっしり詰まっている．それらの主要技術について解説する．

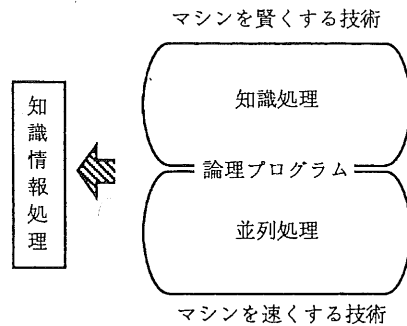


図 1.2.1 第五世代コンピュータ研究開発の枠組み

最後は，並列オペレーティングシステム PIMOS である．PIMOS は大規模並列計算機のためにまったくのゼロから作られた，おそらく世界で始めてのオペレーティングシステムであろう．ことごとく分散した資源管理方式や，プログラム開発環境に特徴をもつ．これらの方式について概要を説明する．

これらの解説を進めるにあたっては，何をやりたいためにどういう技術が作られ，それがどのようにすばらしいか，また，並列論理型言語を用いて研究開発を進めたためにどんな技術が生まれ，それによって何が可能になったかということも，技術そのものの解説に織り混ぜながら解き明かしてゆきたい．なお，並列処理技術のなかでも最もユーザ寄りの，並列プログラミングと応用については，第 III 編で詳述する．

1.2 大目標とアプローチ

a. 目標イメージ

第五世代コンピュータ (FGCS) プロジェクトは，21 世紀の高度知識情報処理に必要とされる，ソフトウェアとハードウェアの基礎技術を確立するために進められてきた．

高度知識情報処理の目指すものは，知識を活用することによる品質の高い情報処理と，プログラミングの煩わしさから利用者をできるだけ解放することである．究極の実現イメージの一例としてよく引き合いに出すのは，「人間が知的な生産活動をするときに，脇にいて助けてくれる賢い助手」である．たとえば，わたしが計算機の設計をするとき，設計方法の知識ベースや設計データベースを活用して設計作業を助けたり，新たな設計問題が発生したときには，その解決方法自体を可能な範囲で生成したりしてくれるような，スーパーワークステーションである．

b. 研究開発の枠組み

このような高度知識情報処理を実現するための技術要素をごく簡単に表現するならば，マシンを賢くするための技術としての知識処理と，知識処理が必要とする膨大な計算能力を提供する技術，すなわちマシンを速くする技術としての並列処理の二つとなる．ICOT では，プロジェクトを進めるにあたって，研究開発の枠組みを図 1.2.1 のように考えた．

知識処理の研究と並列処理の研究という，ともにたいそう難しいであろう二つの研究テーマを，遊離することなく並行して進めてゆくためには，それらのインタフェースになるとともに，理論的にしっかりした共通の研究基盤がぜひとも必要である．その役割を担うために選ばれたのが，**論理型言語**であった．この選択は，プロジェクトの創始者らの鋭い洞察に基づく次のような作業仮説によっている．すなわち，将来の情報処理にとって問題となりそうな数々の事項，たとえばプログラムの記述能力や知識の表現能力，ソフトウェアの生産性，並列処理の実行効率等々が，論理プログラムの枠組みに基づいてシステム全体を再構築することにより，劇的に改善されるというものである．

この作業仮説に基づき，システムのすべて，すなわちハードウェア，システムソフトウェア，中核となる言語，応用ソフトウェア，プログラム方法論まで，すべてを論理プログラムの枠組みのもとに，ゼロから構築する努力を開始した．これらに関する長い長い努力の道のりは，歴史編で述べたとおりである．

c. システム構成

こうして第五世代コンピュータの目標とするシステムイメージが，プロジェクトの中期までに固まった．本書の主題である並列処理に重点を置いてシステム構成を見た場合，図 1.2.2 のような階層構成となる．

最下層はハードウェアの層である．並列推論マシン PIM はここに入る．1,000 台の要素プロセッサをもち，論理型言語を効率よく実行する並列計算機が目標であった．その上の層が，核言語 KL1 の並列言語処理系の層である．KL1 は並列処理のために設計した論理型言語であり，ハードウェアとソフトウェアのインタフェースとなる．一種の高級言語マシンシステムであるため，核言語の層は OS よりも下にきている．その上に，並列オペレーティングシステム PIMOS の層がある．ここから上は，すべて KL1 で記述される．データベースを含む知識ベースシステムも，PIMOS とほぼ同層にある．応用ソフトウェアは，現在はまだほとんど KL1 をそのまま用いて記述しているが，プログラマの負担軽減のために，ユーザ言語や特定の問題領域向き言語が研究開発され始めている．応用ソフトウェアの層は実は大変に厚く，知識処理に関する諸々の応用ソフトウェアや研究ツールもここに含まれる．

d. アプローチ

ICOT では，これらすべての層にわたって，まったくのゼロに近いところから研究開発の積み重ねを行なってきた．論理型言語に基づく一貫したコンセプトですべてを作り，作ったなら使い込み，試してみるということを研究開発の基本ポリシーとしてきたのである．それは，先に述べた作業仮説を証明するための実践である．また，過去二十数年にわたる数多くの並列マシン失敗の歴史を振り返るとき，重大な失敗原因の一つが，並列のソフトウェアに関する研究環境を整備しきれなかったことによるとの分析があった．したがって今回のプロジェクトでは，ソフトウェアの研究開発環境作りに力を注ぐとともに，ハードウェアとソフトウェアの研究が密接な協調関係をもつこと，そしてお互いの研究成果を反映させながら次第に成長していけるような研究開発の仕掛けを作ることに大きな努力を払ってきたのである．

しかしながら，ほとんどゼロに近いところからハードウェアもソフトウェアも並行して研究開発を進めるというのは至難の技である．これを着実に進めるために採用したのが言語指向のアプローチである．まず最初に，並列プログラムの記述と並列実行の双方の効率的実現を目指して，核言語 KL1 の言語設計を行なった．これをインタフェースとして，図 1.2.2でいうならば下側の，並列推論マシン PIM のアーキテクチャおよびハードウェア設計と，KL1 言語の並列処理系の設計，そして上側の並列オペレーティングシステム PIMOS，さらに上の応用ソフトウェアの設計を並行して進めたのである．

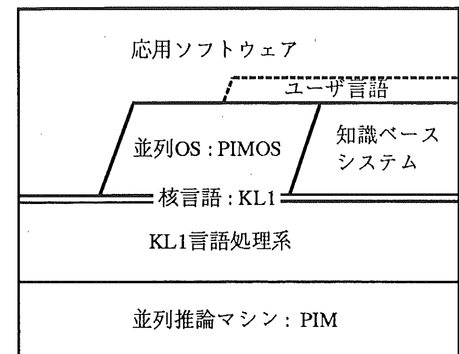


図 1.2.2 第五世代コンピュータ のシステム構成

1.3 どんな並列処理を目指すか

1.3.1 分散メモリ型大規模並列マシン上の汎用並列処理 —MIMD 対 SIMD—

ここでは，第五世代コンピュータの目指す並列処理と，第五世代コンピュータの並列処理研究ではカバーしていない領域とを対比の上で見ることにしよう．

並列推論マシン PIM は，分散メモリ構造をもつ MIMD 型並列マシンである[†1]．ネットワーク接続された計算ノード (PIM/p ではクラスタ，PIM/m やマルチ PSI では PE 1 個) の間ではメッセージ通信が行なわれる．各計算ノードは十分に高い計算性能と，大量のメモリをもっている．MIMD 型並列マシンが対象とすべき処理の特徴は，SIMD 型並列マシンと比べることにより鮮明となる．

1988 年に第五世代コンピュータ国際会議 (FGCS'88) を開いたとき，はじめて 64PE のマルチ PSI を出展した．このときある新聞は，「ICOT の作ったのは 64 台接続のマルチプロセッサでしかないが，世の中にはすでに 65,000 台もつながっているものもある」という記事を載せたことがある．この 65,000 台つないだのが SIMD マシンであった (よく知られているコネクションマシンの最初の 2 モデル)．SIMD マシンの特徴の一つは，図 1.3.1に示すように，小能力のプロセッサと小容量のメモリの対が，数多く高速のネットワークで接続された構成をとることである．これは，計算能力

†1 MIMD と SIMD は並列計算機の命令実行方式の分類である．MIMD: Multiple Instruction stream, Multiple Data stream; SIMD: Single Instruction stream, Multiple Data stream.

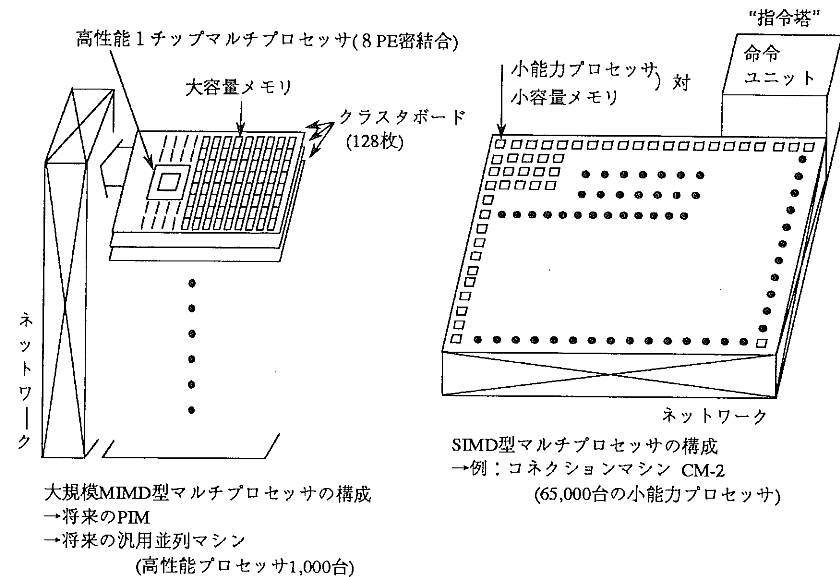


図 1.3.1 大規模 MIMD 型並列マシン対 SIMD

をもつメモリが空間内に均質に分布していると見ることもできる．もう一つの特徴は，すべてのプロセッサはただ一つの命令ユニットから同期して同じ命令を受け取り，足並みを揃えて動くことである．このようなマシンは，均質なデータに対し均質な処理を定められた手順で実行するような用途に向いている．たとえば，密なマトリクス計算とか画像処理がそれにあたる．命令が共通だから，個々のプロセッサがもつ計算の途中経過に従って，受け取った命令を実行するか無視するか程度の単純な選択は可能であるが，分布して配置されているデータのおのおのに依存して，処理アルゴリズムを大きく変えてゆくようなことは難しい．SIMD マシンが得意とする均質なデータ処理のなかの並列性のことを「データパラレリズム」と呼ぶことがある．SIMD マシンは，データパラレルなアルゴリズムが存在する問題に限って，非常に高い性能を発揮する．

一方 MIMD マシンというのは，一つひとつのプロセッサがもっと賢くて自立している．高性能のプロセッサと大容量のメモリの組が，ネットワークで接続された構成をとり，個々のプロセッサは独立した命令列を実行する．SIMD で，均質なデータに対して同期的な処理が行なわれるのとは対照的に，MIMD では，プロセッサが違えば異なる質のデータが置かれていて処理の質が違ったり，たとえ同質のデータに同質の処理が行なわれる場合でも，処理の進み方が異なっていて当たり前との立場をとる．同期が必要なときだけ，プログラムで同期をとる．すなわち，本質的に処理の不均質さが生じることを前提としたシステムである．したがって SIMD に比べて，不均質なデータに対する不均質な処理のある問題や，実行時の計算負荷が動的に変化する問題などにも適用しやすいシステムといえる．知識処理はその代表である．プロセッサによって格納されている知識の種類が違うために処理アルゴリズムが異なることもあり，入ってくるデータに応じて駆動される知識や推論のための計算量が動的に変化する．このように不均質な度合いの大きい問題を実行する場合ほど，同期をとる際に待たされる可能性が高くなる．あるプログラムが待ちに入ったときにプロセッサが遊ぶようでは性能は出ない．そこでプロセッサをいつも忙しく保つためには，たくさんの仕事をプロセッサに与えておく必要があり，必然的にプロセッサにつくメモリも大容量でなければならない．一般にメモリ容量は，プロセッサ性能に比例して大きい必要があるが，同時に対象とする処理の不均質さの度合いに応じて，大容量化が求められる．逆に，ネットワーク性能が高いと，同期の待ち時間の短縮や仕事の分配，再配置を高い頻度で行なえるようになり，プロセッサ当たりに必要なメモリ量は減少する傾向にある．

図 1.3.1のなかの左の絵は，将来の大規模 MIMD 型汎用並列マシンおよび将来の PIM のイメージを表わしたものである．8プロセッサを1チップに集積した LSI を用いて，ボード1枚でクラスタを構成する．クラスタごとに大量の共有メモリをもつ姿が描かれている．クラスタボード 128 枚をネットワーク接続し，1024PE からなるシステムを構成する．筐体1本ないし2本に実装する．

第五世代コンピュータが目指しているのは，このような MIMD マシンであり，大量の計算を発生する問題のうち，不均質なデータに対する不均質な処理や，動的処理の比率の高い問題を対象と考えている．知識処理はそのなかの主要な問題領域の一つであるが，そのほかにも，SIMD マシンには向かないが非常に高い計算能力を要する問題ならば，知識処理の要素が少なくてもよい処理対象と考えている．SIMD では効率よく扱えないような処理に対し，より広範囲に適用できるシステムを指向しているわけであり，それだけ難しい技術に挑戦しているともいえる．

1.3.2 並列処理でどのように速くしたいか

筆者らが対象とするのは，MIMD マシンを用いてプロセッサ台数に比べてはるかに大きい問題を解かせるような場合である．このとき，N 台のプロセッサを用いて最大でも N 倍の高速化が図れればよい (N の値はいまのところ 1,000 程度)．問題の性質によっては，定数分の N 倍だったり，N の平方根倍でも当たり前と考えている．

一方，これとは異なる考え方として，問題サイズと同程度あるいはさらに多数のプロセッサを使用して，問題サイズ P に対する計算時間のオーダを下げてしまおうというアプローチがある．逐次実行で P の2乗かかる計算が，並列処理で $P\log P$ に短縮できるというような議論がそれにあたる．実際にコネクションマシンでは，65,000 台のプロセッサ数を問題サイズと同程度と考えてよい場合に限り，この議論があてはまる．確かにオーダが下がるのはありがたく見えるが，そのようなアルゴリズムは限られており，P の大きさも限定される．

N 台のプロセッサで N 倍しか速くならないのかといわれたら，大きな問題を MIMD で解かせるときに，それ以上の何を望めるのかと問い返すことにしよう．しかしながら，世の中にはもっと過激な人もいて，並列処理を使っても速くならないと主張することがある．よくよく尋ねてみると，十分なプロセッサを用いても，並列処理によって計算時間のオーダを下げるようなアルゴリズムが存在しない，だから並列処理では速くならないとの議論展開である．単に，よい並列アルゴリズムがないといっているのだと思うことにしよう．

ところが，N 台のプロセッサをもってきて，N 倍よりも速くなる場合がある．スーパーリニア・スピードアップと呼ばれる．たとえば，スケジューリングを調節することにより探索の枝刈り効率が改善され，計算量が削減できるような問題があったとする．N 倍よりも十分速くなるのは次のような場合である．1台のプロセッサで，よくない擬似並列のスケジューリングをしていたために，枝刈り効率が悪く無駄な計算をしていたことを知らなかったとする．これを N 台プロセッサで実並列実行したら，たまたま非常によいスケジューリング状態が得られ，枝刈り効率が飛躍的に高まって，総計算量が著しく減少したとする．この場合は，N 台並列実行による高速化のほかに，総計算量減少による時間短縮が加わるため，後者の分だけ N 倍よりも高速化したように見えるのである．このような場合は，実は1台プロセッサでの実行時間も，スケジューリングを改善することにより短縮可能なことがおわかりであろう．

1.3.3 システムの特徴のまとめ

ここでは，並列処理から見た第五世代コンピュータシステムの全体イメージと主な特徴を簡単にまとめ，以下の各章への導入部とするとともに，あとから読み返すときのためのコンパクトなサマリーとする．

A. 並列論理型言語

論理型言語を核言語に使用することが，研究開発上の最初の選択であった．これは先に述べたとおり，プロジェクトの作業仮説に基づくものであり，論理型言語が知識処理と並列処理の両方に適するという，プロジェクト創始者たちの洞察によっていた．

核言語 KL1 は，並列処理のために設計された論理型言語であり，ソフトウェアとハードウェアの唯一のインタフェースである．KL1 は，並列プログラミングに適した汎用言語であり，特に並列の記号処理に適しているが，ある種の論理型言語がもつような特定の推論 (reasoning) メカニズムや知識表現機能を実現しよ

うとするものではない．KL1は，後に示す対象問題領域のプログラムを効率よく実現するための数多くの優れた機能を有する．

B. 分散メモリ型 MIMD マシン

並列推論マシン PIM のハードウェア構成を大まかにとらえるならば，分散メモリ構造をもつ MIMD 型並列計算機であり，数百の計算ノードが高速のネットワークで接続されている．プロセッサ台数の拡張性と実現の容易さを重視している．計算ノードは，1台のプロセッサか密結合のクラスタであり，大容量のメモリを備える．プロセッサは記号処理に適した機能をもつ．

C. 対象問題領域

大規模な知識処理および記号処理が，第一義的応用問題領域であるが，より広範囲の領域として，**動的で均質さの小さい大量計算問題**(非データ並列の大量計算問題)も対象とする．

問題の性質を説明するならば，たとえば発見的知識を用いる探索問題を解く場合，各計算ノードにおける計算負荷は，探索木の拡大や枝刈りの発生により劇的に変動する．また知識処理システムが，多くの性質の異なるオブジェクトからできている場合，それぞれのオブジェクトはしばしば非同期的に振る舞い，均質でない計算負荷を発生する．これらの問題の計算負荷は，実行する前から正確に予測することは困難な場合が多く，よい負荷バランスを得ることが難しい．

D. 言語実装

分散メモリ構造のハードウェア上に，一つの KL1 システムが実装されており，計算ノードごとに別々の KL1 システムが載っているのではない．論理変数やプログラムコードについて大域名前空間を実現している．すなわち，変数や述語の名前はシステム全体にわたって一括管理されている．プロセス間の通信や同期は，たとえ別々の計算ノード間であっても暗黙のうちに行なわれる．したがってプログラマは，煩わしい通信や同期を気にすることなくプログラムが記述できるだけでなく，通信・同期のバグも入りにくい．また小粒度の並行プロセスを効率よく実現していることも特徴である．

これらの実装方式は，KL1 言語のもつ上述の問題領域向き機能を効率よく実現することを目指したものである．

E. 並列 OS も新規開発

PIMOS は，並列推論マシンのためのオペレーティングシステムである．並列ソフトウェアの研究開発を行なう実験マシンのための実用 OS ともいえる．PIMOS は実験マシンによく見掛けるバックエンド OS ではなく，それ自体で完結したスタンドアローン OS である．また1個の OS が並列マシン上に分布して存在するように作られており，要素プロセッサごとに独立の OS を置くような分散 OS ではない．PIMOS はすべて KL1 で書かれており，アーキテクチャの細部からの独立性は高い．

PIMOS を実現するにあたっては，資源管理の考え方がその骨格をなすものとして重要な位置をしめている．PIMOS では，あらゆる資源を木構造を用いて分散管理し，ボトルネックの発生を防ぐとともに，OS の発生する通信量の削減にも努めている．

F. 負荷バランスの基本方針

計算ノード間の負荷割つけおよび負荷バランス制御は，プログラム制御を基本とする．プログラムで制御できないような自動負荷バランス機構は設けない．このことは，性質のことなる多くの問題に対して，良好な負荷バランスを実現する方式をこれからまだまだ研究開発してゆくことが必要であり，そのための道具として使えるシステムを目指すことを意味している．そこで言語システムは，ソフトウェアによる負荷分散実験に必要な，機能と処理効率を実現する．

第2章 KL1言語

2.1 はじめに

KL1 は並列論理型言語 Guarded Horn Clauses[1] に基づいて設計された，並列処理の記述に適したプログラム言語である．第五世代コンピュータプロジェクトがその目標としてきた並列推論システムの中核となる並列推論マシンの共通核言語として，オペレーティングシステム [2] から種々の応用プログラムの記述にまで，広く用いられてきた [3]．この章では,KL1 の言語仕様とプログラミング技法について解説する．

2.2 KL1 言語とは

KL1 という言語の特徴を一言でいうならば

記号処理を並列に行なうための言語

ということになる．

数値処理に比べて記号処理では，複雑に絡み合うデータを取り扱う必要があることが多い．このため，こうしたデータをどのようなデータ構造で表現するかが重要になる．また，そうしたデータ構造をどのようにメモリ上 (あるいはディスク装置上) で管理するかも問題である．こうした管理にプログラマが多くの労力を割かずに済むように，言語システムで基本的な機能を用意して，もっと本質的なプログラミングに集中できるようにしよう，というのが記号処理言語の考え方である．具体的には，一意性のあるものに名前をつけると自動的に一意な表現を与えてくれる**記号アトム**の機構や，標準的なデータ構造についてそのためのメモリ割りつけや解放の労力を減らしてくれる**自動メモリ管理機構**などが，代表的な特徴である．この点，KL1 は代表的な記号処理言語である Lisp 一族と同様の機能を提供している．

並列処理のためには，全体の処理を複数の部分処理に分割して，必要なところでは部分処理間の同期を取りながら計算を進める必要がある．このために，逐次処理言語に並列実行を指定する機構と同期のための機構を追加し，並列処理にも使えるようにした言語 (あるいは OS の機能まで含めたシステム) は数多い．

こうしたアプローチには二つの大きな問題点がある．一つは，もともとの言語の設計の原則は逐次処理のままなので，並列に実行できる部分をいちいち指定しなければならないことである．このため，同じプログラムをプロセッサ台数の大きく異なるハードウェアで共通に使い，しかも効率よく動かすようにすることが難しく，ハードウェアごとにプログラム全体をかなり書き直す必要が生じる．まして，どのような並列処理が適当なのかよくわからない問題について，実験を積み重ねながら並列処理の仕方を模索していくような場合，そのたびにプログラムを書き直してデバッグし直すことになり，多大な労力が必要になる．もう一つは同期処理の面倒さである．同期の必要性を意識しながらのプログラミングは非常に厄介であるし，同期にバグが入ると，そのバグがどのように表面化するかに再現性がない (実行するたびに違う現象が起きる) ことが多いので，デバッグは非常に困難になる．

KL1 は逐次処理に並列実行のための機構を付加するという方法で設計した言語ではない．最初からすべてが並行動作することを前提とした言語である．こうすると非常に頻繁な同期が必要になるのだが，**デー**

タフロー同期機構を導入して同期を自動化することによって，プログラマによる同期の誤りが入り込む可能性を排除している．物理的な並列実行の指定は別途**プラグマ**と呼ばれる記述によって行なう．プラグマはプログラムの正当性[†1]を変えないように設計してあるので，並列処理の仕方を変えるたびにデバッグし直す必要がない．

こうした KL1 の特徴のおかげで，並列処理ソフトウェアの研究開発の労力は非常に軽減される．同じプログラムのプラグマ部分を変更するだけでさまざまな並列実行の仕方を指定できるので，いったんバグを取ってしまえば並列実行指定を変えるたびにバグに悩まされる必要はない．このため第五世代コンピュータプロジェクトでは，並列処理の最も困難な課題である負荷分散方式の研究などを，数多くの実験を行ないながら実証的に進めることができたわけである．

2.3 実行機構の概要

まず KL1 のプログラム言語としての機構の概要を説明することにしよう．

2.3.1 プログラムの形式と基本実行機構

最初に KL1 のプログラムの形式と，基本的な実行機構について，簡単な例を用いて解説する．

A. 簡単なプログラム

最も簡単な KL1 プログラムの例として，入力の 0, 1 を反転して出力するインバータを考えてみる．プログラムは以下のように書ける．

```
:- module inverter.
:- public not/2.

not(In, Out):- In = 0 | Out = 1.
not(In, Out):- In = 1 | Out = 0.
```

最初の2行はプログラムのモジュール化のためのおまじないで，ここでは気にしなくてよい．あとの2行がプログラム本体である．それぞれ

- もし not の第1引数 In が 0 だったら，第2引数 Out は 1 にする
- もし not の第1引数 In が 1 だったら，第2引数 Out は 0 にする

と読む．この二つの行をそれぞれ**節** (clause) と呼び，KL1 のプログラムの基本単位である．

それぞれの節の冒頭部分にある “`not(In, Out)`” は，この節が述語 not に関する定義であること，引数は二つで，この節のなかではそれぞれ In, Out と呼ぶことを宣言している．この部分を節の**ヘッド** (head) と呼ぶ．**述語**という名前は KL1 の論理型言語としての生い立ちからくるのだが，ここでは単に手続き，あるいはサブルーチンのことだと思ってさしつかえない．

ヘッドに続く “`:-`” のあと，縦棒 (“|”) の前までを**ガード** (guard) と呼ぶ．ガードはその節を選んでよいかどうかの条件を指定する条件部である．

縦棒のあと，フルストップ (“.”) までを**ボディ** (body) と呼ぶ．ボディはその節を選んだときに何をするかを指定する部分である．ボディにはいろいろなものが書けるが，この例では出力用の引数の値を決める操作である “`Out = 1`” などを行なっている．

B. プログラムの実行

前掲のインバータのプログラムを会話的に実行すると，以下のようになる．

```
?- inverter:not(1, X).
X = 0
yes.
```

会話的な実行にはプロンプト記号 (“?-”) のあとに呼び出したい述語名と引数並びを書き，フルストップで終わる．述語名 not の前の “`inverter:`” は述語を定義しているモジュールの指定である．述語名と引数の並びを合わせて**ゴール** (goal) と呼ぶ．ゴールは KL1 プログラムの実行の単位である．

ここでは引数に 1 と X を与えた．整数 1 は普通の整数値 1 を表わす．KL1 では大文字で始まる名前は変数を表わす．ここでは，ほかにどこにも出てこないまっさらの変数 X を第2引数にわたしたわけである．

プログラムの実行は以下のようになる．

†1 正確には停止性を除いた部分正当性．

1) 第1引数の In は会話的な呼出しのときに与えた引数 1 に対応する．これは，二つあった節のうち，前のほうの節のガードに書いた選択条件 “`In = 0`” は満たさないが，あとの節の条件 “`In = 1`” は満たしている．そこで，このあとのほうの節を選ぶ．
2) 節が選ばれたので，そのボディを実行する．ボディでは第2引数 Out (ここでは会話的に呼び出したときに書いた X に対応づけられている) の値を 0 と決めている (“`Out = 0`”)．
3) ほかにすることはないので，これで実行を終わる．

この実行の結果，引数に与えた X の値が 0 に決まったので，それが次の行に表示されている．その後の “yes” は実行がつつがなく終了したことを示すものである[†1]．

KL1 の変数は，C のような手続き型言語でいう「変数」とは大きく異なる．手続き型言語では「変数」は値の格納場所であって，計算の進行に伴って格納されている値は 0 になったり 1 になったりと変化する．KL1 の変数は数学でいう変数により近く，値は決まっていないか決まっているかのどちらかで，いったん値を決めたらあとで変わることはない．

最初に与える第1引数を 0 に変えれば，当然

```
?- inverter:not(0, Y).
Y = 1
yes.
```

のように結果は 1 になる．

C. ユニフィケーション

前掲のインバータのプログラムのなかの “`In = 0`” や “`Out = 1`” のような，等号 (“=”) の両辺に値を書いた形のゴールを**ユニフィケーション** (unification) と呼ぶ．等号を用いているように，これは両辺の値が等しいという意味だが，縦棒の左のガードにある “=” は条件指定，右のボディにある “=” は値の決定と，同じ記号でも出てくる場所で意味が違う．C 言語ならそれぞれ “==” と “=” にあたるようなものである．

†1 後に述べるが，プログラムの実行はつつがなく終了するとは限らない．

a. ガード・ユニフィケーション

節の選択条件になるガードでのユニフィケーションは，両辺が等しいことが条件の一部であることを示すものである．これは単に等しいかどうかを試すだけなので受動的なユニフィケーション (passive unification) とも呼ばれる．前掲の例の “`In = 0`” では In が 0 かどうかを調べているわけである．

b. ボディ・ユニフィケーション

実行を指定するボディでのユニフィケーションは，両辺を等しいものにするという積極的な操作で，能動的なユニフィケーション (active unification) とも呼ばれる．前掲の例の “`Out = 1`” では，まだ値の決まっていなかった Out の値を 1 に決めてしまうことによって，両辺を積極的に等しいものにしてしまっているわけである．

D. 節の形式と基本実行機構

プログラムを定義する節は，一般には以下のような形をしている．

述語名 (引数, ...) :- ガード | ボディ.

それぞれの部分の役割は以下のとおりである．

述語名: 節で定義 (の一部) を与える述語 (手続き) の名前．

引数: 述語の引数と節のなかで使う変数名の対応をつけるための仮引数の並び．KL1 では変数名の有効範囲は一つの節のなか (および，一つの会話的な実行指示のなか) だけで，同じ名前でも別の節にあればまったく別のものとみなされる．同じ述語の再帰的な呼出しについても，そのたびごとに同じ名前の変数が別の変数を意味する．これは C の auto 変数と同様である．

ガード: 節の適用条件を指定する部分．カンマで区切ってゴールをいくつでも書け，すべてを満足した場合にだけ適用条件を満たしたものとする．ガードにはユニフィケーションのほかに，大小比較など，言語で定義するある決まった種類の述語の呼出しだけが書ける．

ボディ: 節を選んだときに実行すべき部分．やはりカンマで区切ってゴールをいくつでも書け，その

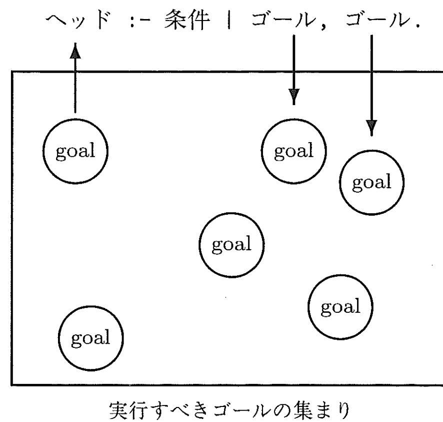


図 **2.3.1** KL1 の実行機構

節が選ばれたらすべてを実行する．ボディにはユニフィケーションのほかに，他の述語，あるいは自分自身を呼び出すゴールが書ける．

KL1 で呼び出せる述語には，あらかじめ言語で定義する**組込み述語** (built-in predicates) と，プログラム中に定義する**ユーザ定義述語** (user-defined predicates) の 2 種類がある．

組込み述語の多くは種々のデータについての基本操作や基本的な値の検査のためのもので，それらがどのように動作するかは KL1 言語の仕様の一部として定められている．特殊な組込み述語として，引数のない述語 "true" がある．これはガードに現われれば常に真 (つまり無条件)，ボディに現われれば何もしないこと (no operation) を意味する．

ユーザ定義述語のゴールの実行では，その述語を定義する節がいくつかあるとすると，まず各節の適用条件であるガードの真偽を試す．次にガードが真であるとわかった節を一つだけ選び[†1]，そのボディのゴール (ユニフィケーションを含む) をすべて今後実行すべきゴールとする．これを繰り返すのが KL1 プログラムの実行過程である．

E. 並列実行，通信，同期の機構

節は，ある条件 (ガード) を満たすゴールを，ボディのゴールに書き換える書換えルールであると見ることもできる．図 2.3.1 に示すように，書き換えるべきゴールの集合[†2]から一つのゴールを取り出し，この書換えを施した結果をゴールの集合に戻すのが実行の 1 ステップになる．書換えを繰り返すことによって，最終的にはすべてのゴールが何もしないゴール "true" にまで書き換えられると実行は終了する[†3]．この過程は，最初に与えられたゴールを "true" にまで書き換えていく**簡約化** (reduction) であるとも考えられる．

この書換えはかならずしも逐次的に一つずつ行なわなくともよい．複数のゴールについていっぺんに行なえば，実行は並列になる．

KL1 では複数のゴールが同じ**変数を共有**することがある．あるゴールが変数の値を決めると，それを共有するほかのゴールがその値を使って，どの節を使って簡約化するかを決めることもできる．これが KL1 の通信機構である．

前掲のインバータのプログラムについて，以下のようなゴール群を実行することを考えてみよう．

```
?- inverter:not(1, X),
   inverter:not(X, Y).
```

このように実行すると，X で示す変数は二つのゴールで共有することになる．

KL1 では，複数のゴールがあるときにはどの順番に実行するかは特に決めておらず，処理系の都合で決めてよいことになっている[†4]．

仮に，まず上に書いたゴール "not(1, X)" から先に実行するものとしてみよう．第 1 引数は 1 だから，第 2 引数 X の値は 0 に決まる．この X はもう一つのゴール "not(X, Y)" の第 1 引数にもなっている．だからこのゴールの第 1 引数は 0 になったわけである．そこで，このゴールを実行すると Y は 1 に決まる．したがって，このゴール列の実行の結果，X は 0 に，Y は 1 になる．

実行の順序が変わったらどうなるだろう．最初に下のほうのゴールを実行しようとしたとする．すると，第 1 引数 X の値はまだ決まっていないから，述語 not の二つの節のガードに書いてある適用条件 In = 0 と In = 1 は，どちらも In の値がまだ決まってい

†1 複数あれば，どれが選ばれるかはわからない．

†2 同じものがあってもよいので，正確にはマルチ集合．

†3 KL1 ではいつまで待っても実行が終らないが有用なプログラムというものも考えられる．

†4 後述の優先度指定機構はあるが，絶対的なものではない．

ないことになる．これではどの節を使ってよいのか，条件を試すことができない．このような場合，当面の間このゴールの実行は見合わせる．これを実行の**中断** (suspension) と呼ぶ．

下のゴールがすぐ実行できないとなると，ほかには上のゴールしかすることはない．そこで上のゴールを先に実行すると X の値が 0 に決まる．これで先程実行を中断していた下のゴールの第 1 引数の値は決まったので，実行を中断し続けている理由はなくなった．そこで，このゴールの実行を再開する．今度はガードの真偽を調べるのに障害はなく，無事 In = 0 の節が選ばれ，Y の値は 1 に決まる．これだけが KL1 の通信と同期の機構で，これ以外には何もない．

要約すると以下のようになる．

- **通信は共有変数**を用いて行なう．
- **同期はガードで必要な値を検査**すると自動的に行なわれる．

KL1 では，選択実行 (if-then-else のようなもの) の条件はすべてガードの真偽を調べることによって決める．その判断に必要な値について，まだ値が決まっていない (計算できていない) 場合は，実行を中断して値が決まるのを待ち合わせるという，自動的な同期が起きる．KL1 の変数はいったん値が決まってしまえば，あとからそれが別の値に変わるということはなく，ずっと同じ値をもち続ける．だから KL1 プログラムでは，単純な計算順序の誤りによって誤った選択をしてしまうという危険性は皆無なのである．

F. 実行機構についての補足

以上が KL1 の基本的な実行機構のすべてなのだが，ここまでの内容でいくつかまだよく説明していないことがあるので，ここで少し補足しておこう．

a. 複数の節が使えたら?

もし二つ以上の節のガードがみな真だったら，どの節を選ぶのだろう．KL1 の言語仕様としては，このような場合にどの節を選ぶかあえて決めていない[†1]．処理系の自由でどう決めてもよいことになっている[†2]．したがって，正しい KL1 プログラムでは，一つの述語についてガードを排他的に書くか，排他的でない場合にはどれを選んでも正しく動くように書かなければならない．

b. どの節も使えなかったら?

もしどの節の選択条件もみな成り立たないとわかったら，どうするのだろう．たとえば，前掲の述語 not を "not(3, X)" のような引数で呼んだらどうなるのだろう．この場合は，ゴールの実行は**失敗** (failure) になり，正常な実行からは外れる．失敗は演算のオーバフローなどと同様に，例外事象として処理する (例外の扱いについては，ここでは省略する)．

c. ボディ・ユニフィケーションで，すでに値の決まっている変数の値をまた決めようとしたら?

すでに決まっている値と同じ値なら，何もしなかったのと同じになる．違う値にしようとしたのなら，ユニフィケーションの失敗になり，やはり例外として処理する．

2.3.2 他のプログラム言語と比べて

ここまで述べてきた KL1 の特徴を，他の言語と比べて振り返ってみよう．

A. 手続き型言語と KL1

変数がその値を変えていくという手続き型言語に見られる考え方は，KL1 には存在しない．KL1 の変数は値が決まっていないか，決まっているかのいずれかで，いったん値が決まったらそれは決して変わらない．時々刻々変化する計算状態が条件判断に影響を与えることはないので，計算順序の自由度がたいへん大きくなる．この特徴は並列処理にとっては大きな利点である．

B. 関数型言語と KL1

時間変化する計算状態が問題にならないという意味で，KL1 は純粋な関数型言語に近い[†3]．並列処理に対する利点も共通している．

関数型言語と KL1 の最大の違いは**非決定性** (non-determinacy) にある．非決定性とは，まったく同じ計算を複数回行なっても違う結果が得られることがある，

†1 節間の優先度を与える機構もあるが，これは絶対的なものではない．

†2 実際，非共有メモリのマルチプロセッサシステム上の処理系では，近くにデータが揃っていてプロセッサ間通信をしなくてもガードの真偽を判定できるような節があれば，そちらを選ぶように実現することがあるので，毎回どの節を選ぶかが変わったりする．

†3 ここでいう意味では，広く使われている Lisp は関数型ではなく，手続き型言語である．

という性質である．この非決定性は複数の節のガードが真である場合に生じる．これはデバッグを考えると欠点なのだが，効率よく問題を解かせる上では利点になる．関数型言語のプログラムはどんな計算機システム上でも（順序は違っても，結局は）同じ計算をするが，KL1 のプログラムは計算機システムの構成（プロセッサの台数や，その間の通信の速度など）に応じて，自在の動きをするように書くことができる．

C.　他の論理型言語と KL1

a.　Prolog と KL1

一見 KL1 のプログラムは，同じホーン論理[†1]に基づく論理型言語である Prolog に非常によく似ている．プログラムを論理式として解釈できる点も同様で，その際にどのように対応づけて解釈するかまでもまったく同じである．しかし，プログラム言語として見たとき，KL1 と Prolog はまったく異なる言語であるといってよい．Prolog は，原則逐次処理の範囲内で，ホーン論理の自動証明系としての完全度をできるだけ上げるように設計されている[†2]．一方，KL1 は完全性の追求はあきらめ，並列処理言語としての能力を追求した言語である．両者はともにホーン論理に基づきながら，まったく違う方向に発展してきた言語で，KL1 を並列 Prolog と呼ぶのは間違いである．

b.　Concurrent Prolog, Parlog, GHC と KL1

Concurrent Prolog[4], Parlog[5], Guarded Horn Clauses (GHC) などは，いわば同じ穴のムジナで，ほぼ同じ方向を目指した言語である．KL1 はこれら committed choice 型などとも呼ばれる並列論理型言語のうちで最もシンプルな言語である GHC を，さらに簡素にした Flat GHC と呼ばれる言語をその基礎としている．

D.　オブジェクト指向言語と KL1

KL1 自身はオブジェクト指向言語ではない．だが，KL1 や他の並列論理型言語で多用されるプログラミングスタイルは，オブジェクトをプロセスとして実現する，オブジェクト指向のプログラミングスタイルである．オブジェクト指向プログラミングパラダイムのかなりの部分は，ごく素直に KL1 で記述できるのである．このプログラミングスタイルについては後に解説する．

†1　機械的な定理証明が比較的容易で，表現力もかなり大きい一階述語論理のサブセット．Prolog（カットなどを含まないピュアなもの）も KL1 も，この論理についての不完全だが健全な自動証明系とみなせる．

†2　しかし，本当に完全ではない．

2.4　KL1 で扱えるデータ

ここまでの例では，具体的なデータとしては 0, 1 などの整数値しか出てこなかった．KL1 ではほかにもさまざまな型のデータが扱える．本節では，KL1 の基本的なデータ型とそれらに対する操作について，簡単に述べる．

2.4.1　変数に型はない

KL1 では，変数についてどんな型のデータが入るかは宣言しない．ソースプログラム中の同じ節の同じ変数が，ある呼ばれ方をしたときは一つの型のデータ，別のときには別の型のデータを値とすることがある．このあたりは Prolog や Lisp と同様である．

このことには利点も欠点もある．プログラム中の変数にはどんな種類の値が入るかが，原則としては入力データを与えるまでわからないので，処理系が最適化しにくいことは欠点の一つである[†3]．また，人間がプログラムを読むときにも，変数の型という理解の助けになる情報が欠けている分だけ，強い型づけをする言語に比べると不便だろう．

一方利点としては，C の union のような特別の記法を用いなくてもさまざまな型のデータを混在させることができること，Ada の generic package のような面倒な宣言をしなくても，いろいろな型のデータに対して共用できるプログラムモジュールを簡単に作れることなどがある．

全体として，変数に型がないことは，プログラムを読むのには不便だが，書くのには便利である．このことは，実験的なプログラムを何度も書き直しながら，アルゴリズムを開発していくような場合には有利になる．これは AI 研究に，変数に型をつけなくてよい Lisp が広く使われてきた理由の一つであろう．一方，すでに確立されたアルゴリズムに基づいたプログラムを組み，

†3　この欠点はコンパイル時の解析によって型を推論すれば，ある程度はカバーできる．

それを長年にわたって保守していくような場合には不利である．

2.4.2　アトミックなデータ

アトミックなデータとは，内部構造をもたず，その値自身だけに意味があるようなデータである．KL1 で扱えるアトミックなデータ型としては以下のものがある．

記号アトム: 特に何の値を表わすわけでもなく，自己同一性だけに意味がある識別子である．プログラム中で必要になる概念などを一意に表わすのに用いる．

整数: 普通の整数値を表わし，表記も通常は普通の十進記法（に必要なら符号がついたもの）を用いる（“3”, “-15” など）．

浮動小数点数: 実数値の近似値を表わす．表記には小数点を用いた十進記法（“3.14” など）と，指数部を加えた記法（“0.314e+1” など）がある．

A.　記号アトム

KL1 の記号アトムは Lisp などと異なり，属性などの情報をもつことはない，純粋の識別子である．記号アトムとその名前の関係づけでさえ，コンパイラやオペレーティングシステムが行なうものであって，言語自体では定めるものではない．したがって，記号アトムについてできる演算は，ガードでの同一性の判断だけである．

アトムの表記は Edinburgh Prolog のアトムの表記と同様で，以下のいずれかである．

- 英小文字から始まり，英数字および下線 “_” の任意個の並び．たとえば “bit”, “pim”, “an_Atom” など．
- 特殊文字（“+”, “-”, “:”, “=” など）の任意個の並び．たとえば “+”, “:-” など．
- 二つの引用符 “’” でくくられた任意の文字列．ただし，引用符を含む場合は引用符を 2 回続ける．たとえば “’Hello world’”, “’’’quoted’’’” など．
- 特殊なアトム．具体的には “!”, “;”, “[]” の 3 種類．

B.　数値アトム

整数，浮動小数点数の数値に対する演算や比較は，言語のプリミティブとして用意した述語である**組込み述語** (built-in predicates) を用いて行なう．

整数演算としては加減乗除の四則演算があり，ガードの条件として大小比較ができる．浮動小数点数に対しても加減乗除の四則演算，ガードでの大小比較ができるが，これらは整数に対するものとは別の演算になっている．通常，これらの演算，比較には，組込み述語を直接用いるよりもマクロ記法を用いるのが便利である．

整数についてのガードでの大小比較には “<”, “>”, “=<”, “>=” を用いる．たとえば “X =< Y” は「X は Y と等しいか，または Y よりも小さい」を表わす．相等，不等の条件には “=:=” と “=\=” を用いる．

たとえば，第 1 引数と第 2 引数のどちらか大きいほうを第 3 引数の値とするようなプログラムは，以下のように書ける．

```
max(X, Y, Z) :- X >= Y | Z = X.
max(X, Y, Z) :- X =< Y | Z = Y.
```

このプログラムでは X と Y が等しい場合，どちらの節のガードも真である．前にも述べたように，このような場合にどちらの節が選ばれるかは，言語仕様としてはまったく規定しない．このプログラムでは，どちらの場合でも結果は変わらないので問題ない．二つの節を非対称的に書いて X と Y が等しい場合は片方しか選ばれないように書くよりも，素直な書き方だともいえよう．

整数の演算のためのマクロは，通常の加減乗除の記号（“+”, “-”, “*”, “/”）と括弧を用いた算術式を “:=” の右辺に，結果を値にしたい変数を左辺に書く[†1]．

たとえば，第 1 引数と第 2 引数の和を第 3 引数の値とするような述語は以下のように定義できる．

```
sum(X, Y, Z) :- true | Z := X + Y.
```

前述のように，ガードに現われる “true” は常に真であるような条件，つまり無条件を表わす．

†1　このほかにビットごとの論理和，論理積，論理排他和，反転などもあるが，ここでは詳しく述べない．

浮動小数点数についても同様のマクロがあるが，浮動小数点数用のものは先頭に“$”をつけて区別する．上述の二つの述語 max と sum を浮動小数点用に書くとしたら，以下の fmax, fsum のようになる．

```
fmax(X, Y, Z) :- X $>= Y | Z = X.
fmax(X, Y, Z) :- X $=< Y | Z = Y.

fsum(X, Y, Z) :- true |
    Z $:= X + Y.
```

2.4.3 構造をもったデータ

構造をもったデータとは，要素となるデータを集めてできているデータのことである．KL1 で扱える構造をもつデータには以下のものがある†1．

ベクタ: 任意の型のデータを要素としてもつような配列構造．要素番号を用いて各要素にアクセスできる．要素の型は一つひとつ異なっていてもよい．要素個数はベクタを作るときに決めることができる．

ストリング: ある範囲の値の整数値を要素としてもつような配列構造．基本的にはベクタと同様のデータ構造だが，要素の型と値の範囲が限定されているため，基本操作やメモリ上での表現を効率化しやすい利点がある．

コンス: 任意の型の二つの要素をもつ構造．コンス一つでは 2 要素のベクタと変わらないのだが，複数のコンスを組み合わせることによってリスト構造などを柔軟に表現できる．

以下，それぞれの構造データについて，やや詳しく見ていこう．

A. ベクタ

ベクタは，任意の型のデータを要素とするような構造である．

ベクタの表記は，中括弧対（“{” と “}”）の間に要素をカンマで区切って並べて書く．たとえば，要素 0, 1, 2 をもつベクタは “{0, 1, 2}” と表わす．ベクタの要素はどんな型でもよいのだから，もちろんまたベクタになっていてもよく，たとえば “{a, b, {0, 1, 2}, d}” は 4 要素をもち，要素の一つが 3 要素のベクタになっているものを表わしている．ベクタの要素は 0 から順に番号づけする．たとえば “{a, b, {0, 1, 2}, d}” の要素番号 1 に対応する要素は “b” である．

このようにプログラム中に書いてしまうベクタのほかに，プログラムの実行中に要素個数を指定して，新しいベクタを作ることもできる．

最初の（要素番号 0 の）要素が記号アトムであるようなベクタには，そのアトムの名前，開括弧（“(”）最初のもの以外の要素をカンマで区切って並べたもの，そして閉括弧（“)”）という特別な表記法が用意されている．たとえば，前述の “{a, b, {0, 1, 2}, d}” は “a(b, {0, 1, 2}, d)” と書き表わしてもよい．これはさまざまな構造を区別したいときに，構造に名前をつけて，その名前と詳細な内容で書き表わすようなスタイルに便利である．このような構造を特に**ファンクタ** (functor) と呼ぶ．ベクタ構造をファンクタ構造とみなす場合には，第 0 要素のアトムをファンクタ名，他の要素を引数と呼ぶ．たとえば “a(b, {0, 1, 2}, d)” というファンクタは，ファンクタ名は a, 第 1 引数は b である．

B. ストリング

ストリングは，ある範囲の値の整数値を要素として任意個もつような構造である．要素の整数として文字コードを並べ，全体として文字列を表わすような使い方が一般的である．こうした文字列は "abc" のように，二重引用符の間に文字を並べて表記する．

ストリングについても要素には 0 からの要素番号がつけられる．たとえば “"abc"” の要素番号 1 の要素は，文字 “b” に対応するコードの整数値である．文字コード系に何を使うかはシステムによる．

C. コンス

コンスは，任意の型の二つの要素をもつ構造である．二つの要素は Lisp に習って car, cdr と呼ぶ．表記にはカギ括弧対（“[” と “]”）の間に car, cdr の要素を縦棒（“|”）で区切って書き並べる．たとえば “[a|b]” は，car がアトム a, cdr がアトム b であるようなコンスを意味する．

†1 このほかにも実行可能コードを表わすものなど，さまざまなデータ構造などがあり，オペレーティングシステムをまるごと KL1 で書くことまでできるようになっているが，ここでは詳しくは述べない．

コンスそれ自体は二つの要素だけをもつ構造データだが，これを組み合わせてさまざまなデータ構造を作れる．その代表がリスト構造である．コンスの car が一つの要素，cdr がリストの残り，要素が一つもないリストはアトム “[]” で表わすものと取り決めれば，二つの要素 a, b をもつリストは “[a | [b | []]]” と，二つのコンスを用いて表わすことができる（図 2.4.1）．このままでは，リストが長くなると多数の括弧が必要になるので，これを “[a, b]” のように書き表わす．

D. 不完全データ構造

ベクタやコンスのように任意の型の要素をもてる構造データの場合，要素の値がまだ決まっていないことがあってもよい．このような構造を**不完全データ構造** (incomplete data structure) と呼ぶ．

不完全データ構造の値の決まっていない要素は，変数として書き表わす．たとえば “[a|X]” は car がアトム a, cdr が X という名前をもつ変数であるようなコンスを表わす．

不完全データ構造のなかの変数も，構造体全体をもっているゴールと別のゴールで共有することができる．変数の値を決めるのは別のゴールのほうであってもよい．つまり，誰かほかの人が詳細を決めることになっているデータ構造，ということになる．こうしたデータ構造は KL1 のプログラミングテクニックのさまざまな局面で重要な働きをつとめ，柔軟なプログラミングスタイルの源となっている．

E. 構造データのユニフィケーション

ここまででは，アトミックなデータ同士のユニフィケーションしか出てこなかったが，構造をもつデータ同士もユニフィケーションできる．

a. ガード・ユニフィケーション

ガードでのユニフィケーションは，両辺の値の一致を調べるものだが，構造体同士の場合は以下のような再帰的な規則になる．

1) まず両辺が全体として同じ型の構造体で，同じ要素個数であることを確かめる．そうでなければ値は不一致である．
2) 次に両者の対応する（同じ要素番号の）各要素について，この規則を再帰的に適用する．すべてについて一致するときのみ，全体が一致するものとする．

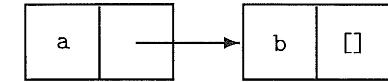


図 2.4.1 コンスによるリストの表現

要素のユニフィケーションにあたって，相手が新しく出てきた変数なら，その変数には対応する要素がわたされる．たとえば，以下のプログラムは第 1 引数にわたされたコンスの car と cdr を，第 2, 第 3 の引数に返すような述語の定義になっている．

```
carcdr(Cons, Car, Cdr) :-
  Cons = [X|Y] |
    Car = X, Cdr = Y.
```

b. ボディ・ユニフィケーション

一辺が変数のボディ・ユニフィケーションは，もう片方が構造をもっていてもいなくても同じで，その変数の値を他辺の値に決めるものである．両方ともが構造体の場合のユニフィケーションの規則は，やはり以下のような再帰的なものである．

1) まず両辺が全体として同じ型の構造体で，同じ要素個数であることを確かめる．そうでなければユニフィケーションは失敗である．
2) 次に両者の対応する（同じ要素番号の）各要素について，この規則を再帰的に適用する．

いずれにせよ，両辺とも値が決まっているようなユニフィケーションをボディで行なうのは，推奨するコーディングスタイルではない．

ここでは KL1 で扱えるデータについて概説した．なかでも要素が何なのかまだ決まっていない不完全データ構造を扱えるのが KL1 の大きな特徴である．次章では，この不完全データ構造を活用するプログラミング手法である，プロセスとストリームについて述べる．

2.5 プロセスとストリーム通信

これまでに KL1 の基本的な言語仕様を解説した．ここでは，KL1 でよく使われる**プロセス** (process) と

プロセス間をつなぐ**ストリーム** (stream) を用いたプログラミングスタイルについて解説する．このプログラミングスタイルは Shapiro と竹内によって最初に提案されたもので [6]，第五世代プロジェクトで開発されたオペレーティングシステムや種々の実験的な並列応用プログラムの多くが，このスタイルに従って書かれている．

2.5.1 略記法

本題に入る前に，いくつかの便利な略記法を紹介しよう．

A. ガード・ユニフィケーションのヘッドでの表記

ガードで引数としてわたされた変数そのものとのユニフィケーションを行なう場合，ヘッドの対応する引数の位置に，直接ユニフィケーションの相手を書いてしまうような略記ができる．たとえば，前のインバータの例にあった

```
not(In, Out) :- In = 1 | Out = 0.
```

という節は

```
not(1, Out) :- true | Out = 0.
```

と略記できる．

整数 1 のような具体的な値でなくても，引数同士のユニフィケーション (引数の値が同じ) をガードで行なう場合，たとえば

```
same(X, Y, R) :- X = Y | R = same.
```

という節は

```
same(X, X, R) :- true | R = same.
```

と，同じ変数名をヘッドに 2 回使って表現することもできる．

B. ゴール true の省略

ガードが無条件の場合，つまりガードが引数のない述語 true の呼出しだけの場合は，ガード部全体をボディとの区切りである縦棒 ("|") ごと省略することができる．たとえば，前述のインバータの例は

```
not(1, Out) :- Out = 0.
```

と書いてもよい．

ガードだけでなく，ボディも true だけならば，ヘッドとの区切りである ":-" ごとボディ全体を省略してもよい．たとえば

```
one(1) :- true.
```

という節は

```
one(1).
```

と書いてもよい．

2.5.2 プロセス

前述のとおり，KL1 の基本的な実行機構は簡約化の (並列な) 繰返しである．しかし，単純な簡約化の繰返しというだけでは，複雑な計算をわかりやすく記述するには役不足で，数多くの簡約化操作をうまくまとめあげるような概念が有用である．そのような役割をはたすのが**プロセス**という考え方である．

A. KL1 のプロセスとは

ボディに同じ述語を呼び出すゴールを一つ含むような節は，(引数は変わるが) 同じことを繰返し行なうループと見ることができる．たとえば，与えられた引数から始めて，0 になるまでカウントダウンしていくような述語は

```
count_down(0).
count_down(N) :- N > 0 |
    M := N-1, count_down(M).
```

という二つの節で定義できる．

この例で "N := N-1" とはなっていないことに気をつけていただきたい．前にも説明したが，KL1 の変数は値の置き場所を示すものではなく，むしろ値そのものにつけた名前なのである．だから N + 1 には M という，N とは異なる名前をつけているのである．

この例のボディには "M := N - 1"，つまり引き算を行なうゴールと，"count_down(M)" という再帰呼出しのゴールの二つがある．KL1 では，ボディ中のゴールの記述順序には特に意味がないので，両者の実行順はどうなるかわからない．引き算よりも前に再帰呼出しの実行を始めることもあるし，両方同時に実行することもある．しかし，述語 count_down を定義する二つの節は，両方とも選択条件として引数の値を調べているので，その値が決まるまで実行は待たされる．この引数の値は引き算の結果なのだから，実際には必ず引き算のほうが先に実行されることになる．

基本的な実行機構である一つひとつの簡約化は細切れの操作なのだが，この繰返し全体を見ると，ある程度の大きさをもった連続性のある計算過程と考えることができる．このような計算過程を**プロセス**と呼ぶ[†1]．プロセスとみなせるような，まとまった計算を行なう述語の例をいくつかあげて，KL1 のプロセスとはどんな概念なのか，どういう特徴をもつのかを，もう少し詳しく説明していこう．

B. リスト要素の和

整数を要素とするリストに対して，要素の総和を計算するような述語は以下のように書ける．

```
sum([], PSum, Sum) :-
    Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).

sum(List, Sum) :- sum(List, 0, Sum).
```

このプログラムは二つの述語の定義からなる．両者は同じ sum という名前だが，引数個数が異なる．KL1 では同じ名前でも，引数個数が異なれば違う述語として扱う．この例でもそうだが，同じ名前で引数個数が異なる述語は補助的な述語の名前として使うことが多い．

†1 KL1 では，プロセスという概念は言語仕様の一部ではなく，あるプログラミングスタイルにつけた名前にすぎないことに注意されたい．

最初の二つの節で定義する 3 引数の述語が，実際の計算をする述語である．この述語を単独に見ると，第 1 引数に与えられるリストの要素すべてと，第 2 引数にわたされる数とを足し合わせ，第 3 引数にその結果を返すものになっている．最初の節は，空のリストについては要素がないのだから第 2 引数をそのまま返せばよい，ということを表わしている．もう一つの節は，リストが空でない場合，最初の要素が One なら，この One と第 2 引数 PSum との和 NewPSum を求め，これとリストの残り部分 Rest の各要素との和が 求める総和である，ということを表わす．

最後の節で定義する 2 引数の述語がもともと定義したかった述語で，リストの要素の総和と 0 の和を計算すれば，リストの要素の総和そのものを計算することになる，という意味になる．

注意されたいのは，2 番目の節のボディにある，足し算と再帰呼出しの二つのゴールは，並列に動いてもまったくかまわないということである．再帰呼出しの実行では，節の選択条件は第 1 引数のリストが空かどうかだけに依っており，足し算の結果は節の選択に関係しない．だから，再帰呼出しだけ先にどんどん実行してしまい，足し算はあとからやってもかまわないのである[†2]．ただし，次々に呼ばれる足し算の引数の一つ (PSum) は，一段前の呼出し時に呼んだ足し算の結果になっている．だから，要素個数と同じ回数行なわれる足し算については，実行順が自動的に決まってしまう．

このように，KL1 でプロセスと呼びならわしている計算過程は必ずしも逐次的な過程ではなく，それ自体のなかに並列性をもっていることも少なくない．プロセスという概念は，プログラムを読む側の解釈にすぎないのである．

C. 自然数のリスト

ある数未満の自然数[†3]すべてを要素とするリストを作るような述語は，以下のように書ける．

†2 並列に行なってよいということは，必ずしも実際に並列に行なえば速くなるということではない．通信のためのコストなどがあるので，普通は足し算のような簡単な操作を並列に行なうメリットはない．実際の KL1 の処理系では，このような足し算を並列に行なおうとしたりはしない．この例は整数の足し算という簡単な操作だったのだが，足し算の代わりにもっと複雑な操作を各要素について行なう必要があれば，実際に並列に行なうメリットがでてくる．

†3 ここでは 0 も自然数だとしている．

```
naturals(N, M, List) :- N >= M |
    List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N1, M, Rest).

naturals(M, List) :-
    naturals(0, M, List).
```

このプログラムも同様に,2 引数の本来定義したかった述語と, 補助的な役割をはたす 3 引数の述語からなっている.

最初の二つの節で定義する述語は, 第 1 引数の値から始めて, 第 2 引数未満の整数すべてを小さい順に要素にもつリストを, 第 3 引数に返すものである. 最初の節は, 第 1 引数が第 2 引数より小さくなければ, 空のリストを返せばよい, という意味である. 次の節は, 第 1 引数のほうが小さければ, 第 1 引数を先頭要素とするリストを返せばよく, リストの残り部分は第 1 引数より一つだけ大きい値から始めて, 第 2 引数未満の整数すべてを小さい順に要素にもつリストにすればよい, という意味である.

最後の節で本来定義したかった述語を定義していて, 3 引数のほうの述語を 0 から始めるように呼べば, 与えられた引数未満の自然数すべてを要素にするリストが作れる, ということを書いてあるわけだ.

この例でも, 二つ目の節のボディにある, ユニフィケーション, 足し算, 再帰呼出しの三つは, どんな順で実行しても (並列に実行しても) かまわない.

2.5.3 ストリーム通信

上に述べたプロセスをモジュールとして結び合わせ, より複雑な計算を記述する方法を説明しよう.

A. プロセスの複合

一つのプロセスの計算結果を用いて, 別のプロセスが計算をするように, プロセスを組み合わせるプログラムを組むことができる. たとえば, 与えられた数未満の自然数の総和を求めるには, 前掲の二つのプロセスを組み合わせて, 以下のようなプログラムを書けばよい.

```
sum_up_to(N, Sum) :-
    naturals(N, List),
    sum(List, Sum).
```

このプログラムでは, 変数 List が二つのプロセスから共有されており, naturals が作ったリストを sum にわたす手段になっている.

この二つのプロセスは同じボディのなかに書かれているのだから, どんな順序で実行してもよい. 2 引数の述語 sum はガードの条件がないから, List の値が決まっていなくても実行でき, 3 引数のほうの述語 sum を呼び出す. こちらのほうは, 引数の値によってどの節を選ぶか決めなければならないので, 値が決まるまで待たされるわけである.

さて, ここで先ほどの自然数のリストを作る述語の定義をもう一度見てみよう.

```
naturals(N, M, List) :- N >= M |
    List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N1, M, Rest).
```

前にも書いたように, 二つ目の節のボディのユニフィケーション, 足し算, 再帰呼出しの三つはどんな順でも実行できる. ユニフィケーションがほかのゴールと並列に実行できるということは, まだ計算が終わらないのに答えを返せるということである. もちろん答えは全部求まったわけではなく, 結果を返す引数とユニフィケーションしているコンス構造の cdr である Rest は, 再帰呼出しゴールの実行結果が入るまでは変数のままだから, 前述の不完全データ構造である. しかし, 計算結果が全体としてはコンスであることはこのユニフィケーションですでに決まってしまっている. また, その car, つまりリストとして見たときの最初の要素は, 最初の呼出しの第 1 引数が 0 なのだから, もうそれに決まっている.

今度は総和を求めるほうのプロセスの本体である 3 引数の sum のほうを, もう一度振り返ってみよう.


```
sum([], PSum, Sum) :- Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).
```

述語 naturals の結果がコンスであることまでもう決まっているのだから, ガードの条件 (実際にはヘッド中に略記されている) の判定はでき, 二つ目の節を選べる. そして, その car (ここでは変数 One で受けている) が 0 であることも決まっている. 第 2 引数の PSum は, 最初は 0 にして呼んでいるのだから, もう足し算の引数は二つとも決まっているわけである. だから 0+0 という足し算は, naturals の実行が少し進んだだけで, もうすぐにでも実行できるようになっているわけだ.

再帰呼出しのほうはそうはいかない. まだ naturals が最初の 1 段階しか進んでいないとすると, Rest の値は決まっていない. つまり, 再帰的に呼び出された sum では第 1 引数がまだ決まっていないわけで, どの節を選べばよいかの判定ができない. この Rest は, naturals と sum の二つのプロセスの間で共有する変数になっていて, naturals がその値を決めることになる. だから sum の再帰呼出しの実行は, naturals のほうの計算がもっと進むまで待たされる. このあとに naturals のほうがもう一段進めば, sum のほうももう一段進めるようになる.

このように, 要素が不完全にしか決まっていない構造体を結果として返し, あとからその要素を確定していくことができるという仕組みは, KL1 の大きな特徴の一つである. 要素がすべて決まるまで結果を返せないのだとしたら, naturals の実行がすべて終わるまで sum の実行を始めることができない. それだけ並列度が低くなってしまうわけである[†1].

B. ストリーム通信

前節に述べた二つのプロセスは, 述語 naturals で表わされるプロセスが次々に作り出す値を, 述語 sum で表わされるプロセスが次々に使っていく, という関係にある. その仲介役をはたすのが, 要素が段々に具体

†1 繰返しになるが, プログラムに並列性があるということと, その並列性を実際の並列実行として実現するかどうかは別の問題である. 実際に並列に実行するか逐次に実行するかは, 通信コストなどの要因を考えて決めるべきである.

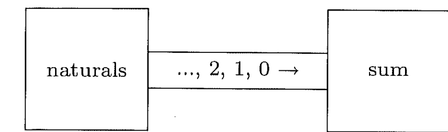


図 2.5.1 プロセス間の通信路

的な値に決まっていくようなリストという不完全データ構造だった.

このリスト構造を, データが次々に流れていくような通信路だと考えることもできる. プロセス naturals は, この通信路に次々に自然数を流していき, プロセス sum がその通信路の出口からデータを読み取っては計算を進める (図 2.5.1). データがまだ到着していなければ, 読み取り側のプロセスは待たされる.

通信路を実現しているのはデータ構造コンスである. その car にはストリームを流れるデータが入り, cdr はストリームの続きを表わす. 空のリスト [] は, 必要な通信が終わりまできてしまい, もうこれ以上メッセージがないことに対応する. このように, 通信路がデータ構造になっているので, データの流れる順序はどのようなデータ構造を作るかで完全に決まってしまう. したがって 計算の実行順序や通信遅延の違いが, そのままデータの順序に影響することはない. このような通信路を**ストリーム** (stream) という[†2].

ストリームを流れるデータは, プロセス間で受けわたされる**メッセージ** (message) であるとも考えられる. つまり, naturals というプロセスは sum というプロセスに次々に, この数も足してくれ, この数も, というメッセージを送っているわけである.

C. プロセスの状態

メッセージを受けてそれに従って仕事をする, というふうにプロセスを書き表わせることを説明したが, もし仕事の内容がメッセージだけで完全に決まるのなら, わざわざそんな書き方をする必要はない. プロセスを作ってそれにメッセージを送る代わりに, 必要な仕事をする述語を定義してそれを呼び出してしまえばよい.

メッセージ駆動のプロセスとして記述する意義は,

†2 この例の場合は合計を求めるのが目的なので, データの到着順は問題ではないのだが, 一般には順序が問題になることが多い.

プロセスには**状態** (state) をもたせることができるということにある．前述の和を求める例では，PSum という部分和を保持する引数が，その状態を表わすものになっていた．

プロセスの状態がもっと明確に現われる例を示そう．整数値を保持し，メッセージによってそれを増減するようなカウンタの機能をもつプロセスは，以下のように書ける．

```
counter(Stream):-
    counter(Stream, 0).

counter([], Count).
counter([up|Stream], Count) :-
    NewCount := Count + 1,
    counter(Stream, NewCount).
counter([down|Stream], Count) :-
    NewCount := Count - 1,
    counter(Stream, NewCount).
```

このプログラムで定義するプロセスは，第2引数としてその時々のカウンタの値を保持している．初期値は0で，あとから up, down というメッセージがくるたびにそれを増減して再帰呼出しの引数にわたすことによって，状態を更新しているわけである．このように KL1 のプロセスは**状態を引数値として保持**する．

このプロセスを呼び出すときには

```
?- counter(Stream),
   some_other_process(Stream).
```

のようにして，メッセージストリームを媒介して他のプロセスと通信できるようにするわけだが，このストリームがカウンタのプロセスとそれ以外のプロセスとを結ぶ唯一の通信手段である．状態として保持しているカウンタの値を直接他のプロセスが操作することはできない．つまり，プロセスの状態として値を保持することは**情報を隠蔽**していることになる．

この隠された情報にはどうやってアクセスしたらよいのだろう．上のプログラムではメッセージを送ってカウンタを上下することはできるが，カウンタの値が何なのかを読むことができない．これについては次節で説明しよう．

D. 複雑なメッセージ

これまでの例では，ストリームを流れるメッセージはすべてアトミックなデータ (整数値や up, down などのアトム) だけだった．もちろん，メッセージとしてもっと複雑な構造をもつデータを流してもかまわない．

よく使われるメッセージとしてファンクタ構造がある．前述のようにファンクタはベクタの一種だが，その要素番号0の要素がアトムで，構造の種類を表わしているようなものである．ファンクタをメッセージとして用いて，要素0のアトムであるファンクタ名でメッセージの種類を，他の要素 (ファンクタの引数) でメッセージの詳細を表わすのが便利である．

前述のカウンタを一つずつではなく，一度にいくつでも増減できるようにするには，増減のためのメッセージを引数をもつファンクタにして，以下のように書けばよい．

```
counter(Stream):-
    counter(Stream, 0).

counter([], Count).
counter([up(N)|Stream], Count) :-
    NewCount := Count + N,
    counter(Stream, NewCount).
counter([down(N)|Stream], Count) :-
    NewCount := Count - N,
    counter(Stream, NewCount).
```

カウンタの値を読み取る機能は，以下のようなファンクタをメッセージとする節を追加すれば実現できる．

```
counter([show(V)|Stream], Count) :-
    V = Count,
    counter(Stream, Count).
```

値を読み取るためのメッセージ show は，カウンタの値を返してもらうための引数 V を未定義のままで送る．受け取ったプロセスは，内部状態に応じてその値を決めてやるわけである．このような未定義の部分を含んだメッセージを**不完全メッセージ** (incomplete message) という．不完全メッセージも不完全データ構造の一種である．このようにすれば，1本のストリームを介して双方向の通信ができるわけである．

以上，ここではプロセスとストリームという概念を用いる KL1 のプログラミングスタイルについて述べた．このプログラミングスタイルの実現には，不完全データ構造が大切な役割をはたしている．

2.6 プロセス・ネットワーク

前節では，KL1 のプログラミングスタイルとしてのプロセスと，その間の通信に用いるメッセージストリームについて述べた．本節では，このプロセスとストリームを組み合わせて複雑なプログラムを組み上げていくための，基本的な手法を紹介する．

2.6.1 フィルタ

与えられた数未満の自然数の和を計算するプログラムの例を 2.5.3 A. 項で紹介したが，これは自然数のリストを作るプロセスと，その要素の総和を計算するプロセスの二つからなっていた．

では少し問題を変えて，与えられた数未満の自然数の平方の総和を計算するプログラムを考えてみよう．前章のプログラムを利用して作るとすると，すぐに思いつく方法は以下の二つだろう．

- 自然数のリストを作るプロセスを直して，自然数の平方のリストを作るようにする．
- リスト要素の総和を計算するプロセスを直して，リスト要素の平方の総和を求めるようにする．

これはいずれも，二つのプロセスのどちらかを改修して，必要な機能を作ろうという方針である．もちろん，このどちらの方法でもプログラムは作れるし，この問題に限っていえばそれで十分である．しかし，もし両方のプロセスともずっと複雑で，簡単には改修できないとしたら，ほかにどのような方法があるだろう．

与えられた数未満の自然数のリストを作るプログラムはすでにある．リスト要素の総和を求めるプログラムもある．とすると，整数のリストをもらって，各要素の平方を要素とするようなリストを作るプログラムをその間に入れてやれば解決である．たとえば，以下のようなプログラムを作ればよい．

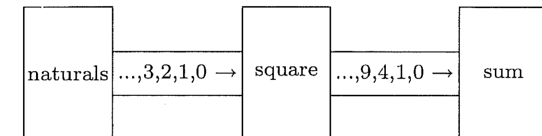


図 2.6.1 フィルタ

```
square([], Out) :- Out = [].
square([One|Rest], Out) :-
    Square := One * One,
    Out = [Square|OutTail],
    square(Rest, OutTail).
```

この述語を使えば，プログラム全体は以下のように書ける．

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    sum(Squares, Sum).
```

述語 square は，リストを入力としてリストを出力するような述語を定義している．この述語の計算過程をプロセスととらえ，入出力のリストをストリームと解釈するとどうなるだろう．このような見方をすると，このプログラムは1本のストリームから整数値のメッセージを受け，もう1本のストリームにその平方をメッセージとして送り出すプロセスを表わしている．このように，あるストリームから受け取ったメッセージに何らかの変換を施して，別のストリームに出力するようなプロセスを**フィルタ** (filter) と呼ぶ．全体のプログラムはプロセス naturals とプロセス sum が，フィルタ square を通るストリームを使って通信する，という構成になる (図 2.6.1)．

この例では，フィルタとなったプロセス square は状態をもたず，出力は入力メッセージだけに依存している．一般には，フィルタの動作は入力メッセージだけではなく，フィルタプロセスの状態に依存してもよい．フィルタプロセスは入力メッセージに応じて状態を変えられるから，入力の履歴に応じてフィルタの仕方を変えることもできる．

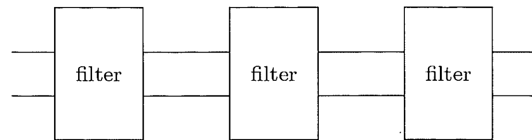


図 2.6.2 パイプライン処理

フィルタは 1 本の入力ストリームと 1 本の出力ストリームをもつ. 二つのフィルタの片方の出力をもう一方の入力につなげれば, 全体としてはやはり 1 入力 1 出力のフィルタができる. この場合, 最初のフィルタがメッセージを次のフィルタに送ってしまったあと, 次のフィルタがそのメッセージの処理をするのと並列に, 最初のフィルタは次のメッセージの処理を行なえる. このようにフィルタをたくさんつなげていけば, **パイプライン処理**ができるわけである (図 2.6.2).

2.6.2 ストリームの連結

また問題を少し変えて, こんどは与えられた数未満の自然数の平方と立方の両方の総和を計算するプログラムを考えてみよう.

前節の場合と同様の方法で, 入力 n に対して n^2+n^3 を出力とするようなフィルタを作るという方法はある. しかし, それではせっかく作った square 述語は使わずに, 別に定義しなければならない. そこで, 前述の square はそのままにして, 別に入力メッセージの立方を出力するようなフィルタを作って, これを使うことを考える[†1].

このようなフィルタ自体をどう定義すればよいかは, もうおわかりだろう.

```
cube([], Out) :- Out = [].
cube([One|Rest], Out) :-
    Cube := One * One * One,
    Out = [Cube|OutTail],
    cube(Rest, OutTail).
```

†1 老婆心ながらつけ加えると, 説明の都合上非常に簡単な例にしているので, これが解くべき問題なら全部書き直してしまったほうが早いぐらいである. ここに説明するような方法は, 本当は必要な計算がもっと複雑で, square のような述語を書き直す手間が大きい場合にこそ有効である.

この二つのフィルタと残りの naturals, sum をどうストリームで結合するかだが, 入力には両方ともプロセス naturals の出力を共通に与えればよい. これはごく簡単で

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    ...
```

と, 同じ変数を両方に書けばよい. 問題は Squares, Cubes の 2 本の出力ストリームに流れるメッセージすべてを, どうやってプロセス sum にわたすかである.

一つの方法は, まず片方のストリームのメッセージを送り込み, それが終わったらもう一方のストリームのメッセージを送るようにすることである. そのための交通整理をする述語は以下のように書ける.

```
append([], In2, Out) :- Out = In2.
append([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    append(In1, In2, OutTail).
```

この述語は第 1, 第 2 の二つの引数が入力ストリームに, 第 3 引数が出力ストリームになっている. 全体としては, まず第 1 引数にやってくるメッセージを次々に出力し, それが終わったら第 2 引数のほうのメッセージを流すようにしている.

最初の節は, 第 1 引数である一方のストリームが終わりまできたら, あとは第 2 引数であるもう一方のストリームをそのまままとめて出力ストリームとしてしまえばよい, という意味である. メッセージを一つひとつ取り出しては中継する必要はないわけである. もう一つの節は, 第 1 引数のストリームのほうにメッセージがきたら, それをそのまま出力することを意味する[†2].

この述語を使って全体のプログラムを書くと, 以下のようになる.

†2 引数をストリームとしてではなく, 単なるリストとして解釈すれば, この述語は二つのリストをつなぎ合わせたようなリストを作る, という述語になっている.


```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    append(Squares, Cubes, Both),
    sum(Both, Sum).
```

メッセージを交通整理する append では, メッセージの中身はまったく見ていない. 単にメッセージがきたかどうかだけを見て, きたメッセージをそのまま中継しているだけである. そのため, どんなメッセージが流れるストリームに対しても, 同じ append を使うことができる. KL1 の変数に型がない (どんな型の値も入れられる) ことの利点が現われる例である.

2.6.3 マージャ

前節で紹介した append を使うやり方には, ちょっと不満が残る. このやり方では, ストリーム Squares の出力が終わるまで, プロセス sum にはストリーム Cubes に流れるメッセージが一つも届かない. もし square の処理にかなり時間がかかるとすると, 並列に動いている sum は暇になってしまう. すでに cube のほうの処理がある程度進んでいれば, その出力もどんどん足し込んでいけるのに, メッセージが届かなくては何もできない.

そこで, 片方のストリームが終わりまでこなくても, もう片方のストリームの出力も中継してやれるようなやり方を考えよう. それには, どちらの入力ストリームにでもよいから, メッセージがきたらどんどん出力に流していくようなプロセスを作ればよい. そのような述語の定義は以下のようになる.

```
merge([], In2, Out) :- Out = In2.
merge(In1, [], Out) :- Out = In1.
merge([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
merge(In1, [Msg|In2], Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
```

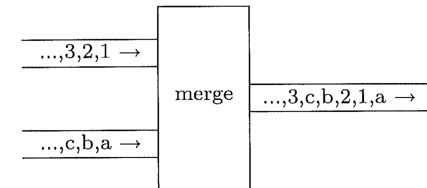


図 2.6.3 マージャ

最初の二つの節は, どちらか一方のストリームが終わりまできたら, もう一方のストリームをそのまま出力につないでしまえばよい, という意味である. 残りの二つの節が, どちらか一方にメッセージがきたときにそれを中継して出力する, ということをするためのものである.

このような, 複数のストリームからの入力を一つのストリームにまとめるプロセスを**マージャ** (merger) という. 前節に述べた append もマージャの一種ともいえるが, まず片方のストリームへのメッセージを流し, それが終わってからもう一方のストリーム, となっているので, より制限が強い.

両方のストリームともにメッセージがきているときに, この述語 merge がどのような動作をするかに注目しよう. この場合, 3 番目, 4 番目の節は両方とも適用条件を満たしている. このようなときにどちらが選ばれるかは, 言語仕様としては決めていない. 同じプログラムを同じ入力データで動かしても, マージャが実際にどのように動くかによって, 流れていくメッセージの順番は毎回違うかもしれない (図 2.6.3). これは,KL1 の特質の一つである非決定性が現われている例である[†1]. この非決定性はプログラムのデバッグには厄介な性質だが, この例のように並列性を上げるために必要な場合がある. なお, 別々のストリームからのメッセージがどんな順で出力されるかは非決定的だが, もともと 1 本のストリームのなかで前後関係があったメッセージは, 出力中でもその前後関係を保っている.

この述語を使えば, 全体のプログラムは以下のようになる.

†1 もちろん, 前節の append のような非決定性のない書き方をすることもできる.

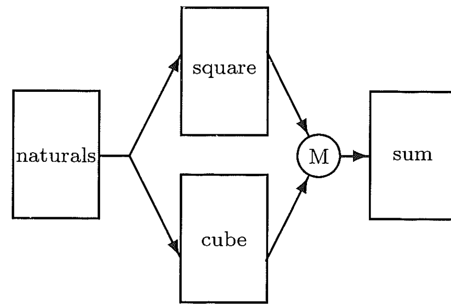


図 2.6.4 マージャを用いたネットワーク

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

これなら両方のストリームのどちらからでも，メッセージがくるごとにどんどんプロセス sum に送りつけることができ，並列性の面で append を使ったプログラムよりも有利になっている．プロセスの全体像は図 2.6.4 に示すようになっている．

2.6.4　組込みのマージャ機能

KL1 プログラムで定義するマージャにはいくつかの問題がある．その一つは，入力ストリームの数である．KL1 では任意引数個数の述語を定義することはできないので，一定数の入力をもつようなマージャしか定義できない．いくらでも多くの入力ストリームをもつようなマージャを作るためには，入力ストリームが増えるたびに (あるいは何本か以上増えるたびに) 新しいマージャを入れていき，マージャの木構造を作っていかなければならない．また，メッセージがマージャの木構造を通り抜けるときには，木構造の深さ程度の数のプロセスを経由することになるので，木構造のバランスが悪いとひどく効率の悪いプログラムになる．入力の増減に応じて構造を変えてバランスを保つように書くこともできるが，プログラムはかなり複雑になる．

たとえ木構造をうまくバランスさせても，入力が n 本のマージャを表わすためには，最低でも深さ $\log n$ の木構造を作らなければならない．一つのメッセージがマージャの木構造を通り抜けるための手間は $\log n$ に比例する程度になる．一方，普通の手続き型の並列言語を考えると，ロック操作と破壊的代入を組み合わせて，任意個数の入力をもつマージャをメッセージが一定の手間で通り抜けるように作ることができる．計算の手間のオーダが違うようでは，決して効率のよいプログラムなど書けない．

これを解決するために，KL1 では組込みのマージャを用意している．組込み述語のマージャは 2 引数で，“merge(In, Out)” のように呼び出す．このままでは，第 1 引数のストリームにメッセージを流すと (つまり，第 1 引数を car 要素にメッセージをもつようなコンス構造とユニファイすると)，第 2 引数に中継する (第 2 引数をそのメッセージを car にもつような別のコンス構造とユニファイする) だけである．マージャとして働き始めるのは，第 1 引数をコンスではなく “{In_1, In_2, ..., In_n}” のようなベクタ構造とユニファイしたときである．こうするとマージャは n 入力のマージャとして働き始める．入力の一つ “In_k” をさらにまた同様のベクタとユニファイすれば，いつでも入力ストリーム数を増やせる．入力ストリームを減らすには，いらなくなった入力ストリームを閉じれば (単に “[]” とユニファイすれば) よい．すべての入力ストリームが閉じられれば，出力ストリームも閉じられる (“[]” になる)．

組込みのマージャは入力ストリームの数によらず，常に一定の手間でメッセージを中継できるし，その一定の手間も KL1 でマージャを書いた場合よりもはるかに小さい．

2.6.5　ディスパッチャ

平方と立方の総和を求めるプログラムでは，平方を作るフィルタ，立方を作るフィルタの両者に同じ入力メッセージを与えればよかった．では，そうはいかない場合 — 入力メッセージのうち，あるメッセージは一つのフィルタを，他のメッセージは別のフィルタを通したい場合はどうしたらよいだろう．

例として，今度は与えられた数未満の自然数について，偶数は平方し，奇数は立法したものの総和を求めることを考えよう．まず，入力を偶数か奇数かに応じて，別々のストリームに振り分けて出力するようなプ

ロセスを作ればよい．そのプログラムは以下のようになる．

```
dispatch([], Odd, Even) :-
    Odd = [], Even = [].
dispatch([One|Rest], Odd, Even) :-
  One/2 =\= 0 |
    Odd = [One|OddTail],
    dispatch(Rest, OddTail, Even).
dispatch([One|Rest], Odd, Even) :-
  One/2 =:= 0 |
    Even = [One|EvenTail],
    dispatch(Rest, Odd, EvenTail).
```

もうプログラムの細かい説明をするまでもあるまい．この述語で実現されるプロセスは，入力ストリームが 1 本，出力ストリームが 2 本あり，入力メッセージの内容に応じてどちらのストリームに出力するかを決めている．このようなプロセスを**ディスパッチャ** (dispatcher) と呼ぶ．

これを用いると，プログラムの全体は以下のようになる．

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    dispatch(Naturals, Odd, Even),
    square(Even, Squares),
    cube(Odd, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

プロセス構成を図 2.6.5 に示す．

2.6.6　差分リスト

複数のプロセスが出力するメッセージを，順不同にマージするのではなく，各プロセスの出力はまとめて，プロセスごとに順序づけしたい場合がある．これは append を使っても実現できるのだが，プロセス数が多くなるとそれに応じて多数の append を使う必要があり，それぞれがメッセージの中継をするので，全体としてはメッセージ数とプロセス数の積に比例する程度の中継の手間がかかってしまう．このような場合に

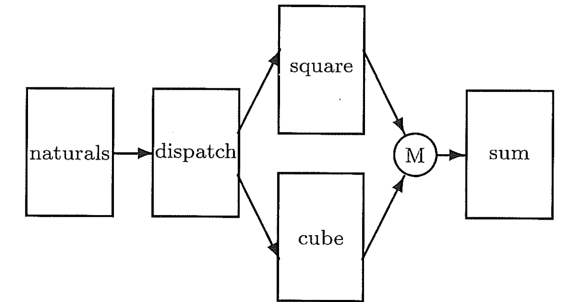


図 2.6.5 ディスパッチャを用いたネットワーク

使うのが，**差分リスト** (difference list) と呼ばれるテクニックである．

普通のスタイルだと，各プロセスは出力を終えると出力ストリームを閉じる．つまり，各プロセスの終了処理の節は以下のようになっている．

```
p(..., O) :- O = [].
```

差分リストを使う場合は，自分の出力を閉じてしまうかわりに，その末尾を次のプロセスの出力の先頭とつなげてしまう．このために，各プロセスは一つずつ余分な引数をもち，それを次のプロセスの出力にする．つまり終了処理の節を，以下のように変えるわけだ．

```
p(..., O, Next) :- O = Next.
```

全体のプログラムは以下のようになる．

```
p(..., O) :-
    p1(..., O,  O1),
    p2(..., O1, O2),
    ...
    pN(..., ON, []),
```

こうすればわざわざ append プロセスを挿入する必要もなくなる．

2.6.7　サーバ

もう一つの重要なプロセスネットワークの構成要素となるプロセスに，複数のプロセスからアクセスされ

る共通のデータを貯めておく**サーバ** (server) がある．サーバに対して，ストリーム経由でそこにアクセスするプロセスを**クライアント** (client) と呼ぶ．

サーバプロセスの典型的な述語構造は以下のようなものになる．

```
p([message(I, O)|S], State) :-
  compute(I, State, O, NewState),
  p(S, NewState).
```

すでに示したカウンタはサーバの一例である．

ここでは，KL1のプロセスをストリームで結合していく基本的な手法を紹介した．実際のプログラムは，こうしたさまざまな手法を組み合わせて構成していくことになる．

2.7 プログラム動作の指定

ここまでに，並列に動作できるKL1プログラムをどのようにして組んでいくかについて一通り説明してきた．今度は，ここまでに述べた並列実行の可能性を，実際にどのようにして物理的な並列実行に結びつけていくかについて説明しよう．並列動作できるように書いたプログラムを実際に並列に動作させるための指定を，KL1では**プラグマ**と呼んでいる．

2.7.1 並列実行指定の方針

まずKL1では，並列実行指定 (プラグマ) についてどのような方針を取ったのか，それはなぜなのかについて述べる．

A. 負荷分散はプログラムで指定する

実際の並列計算機に計算を効率よく行なわせるためには，二つのことを考えなくてはならない．

負荷の分散 行なわなければならない計算が1台のプロセッサに集中してしまうと，そのプロセッサの処理が終わるまで計算が終わらなくなってしまい，並列に計算している意味がない．したがって，全体の問題をいくつかの部分問題に分割し，多くのプロセッサに均等に分散して，どのプロセッサにも同程度の負荷がかかるようにしなければならない．

通信の削減 全体の問題をいくつかの部分問題に分割して分散させるには，まず問題を配るための通信が必要で，最後には結果を集めるための通信が必要になる．また，効率のよいアルゴリズムを使うには部分問題が完全に独立にできず，計算途中でも通信が必要になることも多い．通信が多くなると，そのコストのためにかえって効率が落ちてしまうこともある．したがって，なるべく部分問題を解くプロセッサ間の通信が少なくなるような問題の分割が重要である．

この負荷分散と通信削減は二律排反関係にあり，あまり分散しすぎると通信が多くなりすぎ，かといってあまり通信を減らそうとすると分散できない．この問題が起きないような問題の分割法を見つけ，また 両者のトレードオフ点を見つけることは，並列処理ソフトウェアの研究の最重要課題といえよう．

特に問題の部分問題への分割の仕方は，解くべき問題が何なのか，どのようなアルゴリズムを使うのかに大きく依存する．現在のソフトウェア技術では，部分問題への分割とその分散についての指針を明記していないようなプログラムが与えられても，それに対していつでも自動的に効率的な負荷分散を行なえるようにすることは，まず不可能である．もちろん，たとえば行列のかけ算を多数行なうなど，あらかじめ行なうべき計算の量を予測しやすい問題領域に限定すれば，かなり効率的な自動並列化を行なうこともできる．しかし，いわゆる知識情報処理のような，計算結果によって次に行なうべき計算が動的に大きく変わることが当たり前の分野では，自動負荷分散は非常に難しい．

現在の技術がこういう水準にあるので，KL1はこの困難な自動負荷分散の方式の研究ツールとして役に立てるものと位置づけている．したがって，負荷分散は言語処理系で自動的には行なおうとせず，プログラマが指定するものとした．ただし，プログラマがさまざまな負荷分散指定を行なうときに，できるだけ簡単にできるように工夫している．それについては次節に述べる．

B. 二つの並列性を分離する

KL1では2種類の並列性を別々に指定するものとしている．2種類の並列性とは，以下のようなものである．

論理的並列性 プログラムのどの部分が並列に動作してもよいかを指定するもの．次に述べる物理的な並列性と区別して，並行性と呼んだりもする．これはプログラムの正しさにかかわるもので，指定が誤っていればプログラムは正しく動作しないかもしれない．KL1では，ガードで節の選択条件を指定すると，選択条件が判定できるデータがそろうまで自動的に実行を遅らせるので，並列に動作してよいかどうかは暗黙に指定することになる．

物理的並列性 論理的には並列に動作してよい部分のうち，どれを実際に並列に動かすかを指定するもの．この指定による並列動作は，もともと並列動作してよいもののうちから指定するのだから，プログラムの正しさには影響しないが，プログラムの実行効率を大きく左右する．KL1では，これは**プラグマ** (pragma) と呼ばれる機能を用いて指定する．

論理的並列性と物理的並列性の指定を分離すれば，プログラムの正しさに影響を与えることなく，負荷分散の仕方だけを変えることができる．分散方式の研究を行なうためにはさまざまな分散方式を実験する必要が生じるが，その際に，ただでさえ困難な並列プログラムのデバッグを 毎回改めてやり直す必要がなければ，実験の効率を著しく高めるだろう．

なお，プラグマはプログラムの正しさとは分離された効率のためだけの指定なので，言語処理系はこれに従わないほうが効率がよいと判断できれば，必ずしも従わない場合もあってよい．

以下，このプラグマの指定方法について概説する．

2.7.2 ゴール分散プラグマ

計算の分散には実行ノード指定のためのプラグマを用いる．指定には

Goal@node(*Node*)

のような形式でボディゴールにプラグマを付加する．ここで指定する**ノード** (node) とは，個別のプロセッサ，または内部では自動負荷分散するようなプロセッサの集まりである．現在のところ，指定にはノードについた一連番号を用いている．

ゴール分散を用いる具体例をあげると，たとえば以下のようになる．

```
p([One|Rest], N, State) :-
    q(One, State, New)@node(N),
    N1 := N + 1,
    p(Rest, N1, New).
```

この例では，次々に到着するメッセージについての処理 (q) を，順番に各ノードに分散しているわけである．何も指定のないゴール (この場合なら足し算と再帰呼出しのゴール) は，元のゴールを実行したのと同じノードで実行する．

2.7.3 ゴール優先度指定プラグマ

並列に実行できるゴールが複数あり，プロセッサの数がそれよりも少ないときは，どのゴールから順に実行するかが効率を大きく左右する場合が少なくない．そこで，どのゴールを先に実行したほうが効率上有利かの示唆を与えるためのプラグマが，ゴール優先度指定プラグマである．指定はボディゴールに

Goal@priority(*Priority*)

のようなプラグマを付加して行なう．この *Priority* が具体的な優先度を指定する部分だが，複雑になるのでここでは詳細は述べない．

優先度指定プラグマは必ずしも守られるとは限らない．必ず守ろうとすると効率的な実装が難しくなり，かえって効率低下の原因となるからである．また，優先度の指定は一つのノード内でだけ有効で，他のノードにもっと優先度の高いゴールがあっても，それを移動してきて先に実行したりはしない．ノード間にわたるような優先度管理を常に行なおうとすると，優先度管理がシステム全体のボトルネックになってしまうからである．

2.7.4 節優先度指定プラグマ

複数の節の選択条件が真である場合，どの節を選ぶかは言語仕様としては定めていない．どれを選んでも正しく動くようにプログラムを書くべきであることは前に述べた．しかし，どの節を選ぶかによって実行効率

に影響を与える場合もある. そこで, どちらの節も選べる場合に, どの節を選ぶと有利かについての示唆を与えるためのプラグマが, 節優先度指定プラグマである. 節の優先度指定には, まず優先してほしい節 (複数でもよい) を先に書き, 他の節との間に “alternatively.” という節のようなものを書く.

節の優先度も, 必ずしも守られるとは限らない. やはり必ず守ろうとすると効率的な実装が難しくなるからである.

2.7.5 荘園

ここまで説明してきたように, KL1 ではゴールの一つひとつを並列に動かすことができ, この一つひとつについて負荷分散や優先度を制御することができる. 複雑に絡み合うデータの並列処理には, こうした細かい制御も重要なのだが, もっとおおまかな制御も必要である.

並列処理では, 一つの問題を複数の異なる方法で解いてみて, 早く答えが出たものを採用する, などという戦略をとることがある. 先にどれかの方法で答えが出たら, まだ答えが出ていないものの実行は止めてしまいたい. あるいは, ある方法が有望で答えが出そうだという予測が立ったら, 有望なものに全力を傾注するために, ほかはちょっと実行を中断したい (とはいえ, 有望と思えた方法が結局はダメかもしれないので, いったん中断したほかの方法による実行も再開できるようにしておきたい) ということもある.

こうした処理のために, 非常に数多くのゴール (PIM の上のある程度おおがかりな実験的応用プログラムの場合, 数百万個に上ることも珍しくない) を一つずつ制御するのは, プログラミングの手間も大変だし, 処理上も効率よく行なうのが難しい. そこで KL1 では, あるゴールを親に, そこから派生したゴールすべてをグループとして制御するための**荘園**と呼ぶ制御構造を導入した[†1].

荘園機能を用いると, そのグループ中のゴール全部の実行の中断, 再開, 放棄や, グループ全体の使ってよい計算時間[†2]などの計算資源の消費の制御ができる. また, 失敗などの例外事象の影響がグループ内からグループ外に波及することの防止や, 実行のトレースなどのデバッグ機能の実現手段としての機能も荘園がもっており, 並列推論マシンのオペレーティングシステム PIMOS の実装には不可欠な機能となっている. 荘園は何重にでも自由に入れ子構造にすることができる. PIMOS 自身の開発環境である Virtual PIMOS (PIMOS 全体を PIMOS の中の一つのタスクとして動かし, デバッグするためのシステム) も, この機能を用いて容易に実現されている.

この節では, プログラムを実際に並列に動作させるための指定である**プラグマ**と, 並列に動作する数多くのプロセスをグループとして制御するための**荘園**の機構について 簡単に述べた. プラグマはあくまで処理系に対する効率的実行のための示唆にすぎず, 処理系の動きを完全に指定するものではないことを再度記しておく.

2.8 一階述語論理と KL1

論理型言語というのは「問題を述語論理で記述すればそのままプログラムとして動く」という宣伝がよくなされていた. この影響か, 論理型言語は人工知能研究向きの特殊なプログラム言語で, 一般のプログラミングとは無縁であるという誤解が, まだ一部に根強くあるようだ. そうした偏見に染まることなく, KL1 の並列プログラム言語としての言語仕様を理解していただきたかったので, ここまでの説明では KL1 と一階述語論理との関係についてあえて触れずにきた. ここまでの説明を理解していただければ, KL1 は それほど風変わりな言語ではないということも, ご理解いただけたのではないだろうか.

実際, KL1 は (Prolog とは違って)「問題を述語論理で記述すればそのままプログラムとして動く」を最初から考えていない. 並列処理の記述に必要な仕様を素直に採り入れたら, いや, むしろ従来のプログラム言語に意識的・無意識的に採用されてきた逐次処理特有の機能を排除したら, ごく自然にこのような言語になったのである.

そうはいっても, KL1 の設計は論理型言語として出発し, その仕様の根幹には論理型言語としての血が脈々と流れている. ここでは, 一階述語論理と KL1 の

†1 荘園という名前に深い意味はないが, すでに日本語の shoen が論文などを通じて世界に広がってしまったので, いまさら変えられずにいる.

†2 現在の PIM 上の処理系では, 物理的な時間ではなくゴールの簡約化の回数の合計で制御している.

関係について, ごく簡単に触れておきたいと思う. なお, 以下の説明の大部分は KL1 だけでなく, Prolog などの他の論理型言語にもあてはまるものである.

2.8.1 プログラムは公理の集まり

KL1 プログラムを構成する節は

述語名 (引数, ...) :- ガード | ボディ.

という形式をしている. ここで縦棒 (“|”) を取り除いて, ガードとボディの区別をなくしてしまうと, 節の形は

述語名 (引数, ...) :- ゴール, ...

という形式である.

前に述べたように, ガードにあるユニフィケーションを, ヘッド中に直接相手を書くように略記してしまうことができた. ガードとボディの区別をなくしてしまったので, 元はボディにあったユニフィケーションにも, この規則を適用することにする. たとえば, ストリームを結合するプログラム

```
append([], In2, Out) :- Out = In2.
append([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    append(In1, In2, OutTail).
```

について, このような書換えをし, 変数名や述語名を少し変えると

```
a([], O, O).
a([M|I1], I2, [M|O]) :- a(I1, I2, O).
```

となる.

このような変換をした結果は, 一階述語論理の公理として解釈することができる. その意味づけは, “:-” の右辺が成り立つならば, 左辺が成り立つ (右辺が空なら, 左辺は無条件に成り立つ) ということで, 上の append の例の二つの節を論理式として書けば, それぞれ

$$\forall o\, A([], o, o) \tag{2.1}$$

$$\forall m, i_1, i_2, o\, A(i_1, i_2, o) \rightarrow A([m|i_1], i_2, [m|o]) \tag{2.2}$$

ということになる. 日本語で書き下せば, (2.1) は「どんな o についても, $A([], o, o)$ が成り立つ」, もう少し解釈を入れると「空リストと何かを結合したものは, その何かである」という意味である. (2.2) は「どんな m, i_1, i_2, o についても, $A(i_1, i_2, o)$ が成り立てば $A([m|i_1], i_2, [m|o])$ が成り立つ」, 解釈を入れて「i_1 と i_2 を結合したものが o になっているのなら, $[m|i_1]$ と i_2 を結合したものは $[m|o]$ になっている」ということになる.

2.8.2 実行は証明過程, 計算結果は反例

このように節を公理に対応づけると, KL1 のプログラムの実行過程は, トップレベルに与えられた命題の証明過程であると考えることができる. たとえば, 以下のようなトップレベルの入力を考える (モジュール名は省略した).

```
?- a([1,2], [3,4,5], L).
```

これを論理として解釈すると

$$\forall l\, \neg A([1,2], [3,4,5], l) \tag{2.3}$$

つまり「どんな l をもってきても, それに対して $A([1,2],[3,4,5],l)$ は成り立たない」, いい換えれば「[1,2] と [3,4,5] を結合したようなものは存在しない」という意味である. プログラムの実行は, この命題が成り立たないことを証明する過程である.

ある命題が成り立たないことを証明するには, 反例をあげればよい. 論理型言語による計算とは, この反例を見つけることである. その反例自身が計算の結果になっている. 上の append なら, $\forall l\, \neg A([1,2],[3,4,5],l)$ の反例, すなわち $A([1,2],[3,4,5],l)$ を満たす l を見つけてやればよい. これは, [1,2] と [3,4,5] を結合したらどんなリストになるかを計算していることにほかならない.

KL1 や Prolog が属するホーン論理に基づく論理型言語では, 反証したい命題から始めてトップダウンに, 公理を用いてブレークダウンしていく方法でこの証明を行なう. このようなやり方を後向き推論 (backward

reasoning) などとも呼ぶ[†1]. 実際の証明手続きを少し追ってみると, 以下のようになる.

1) 公理である (2.2) から

$$A([2],[3,4,5],x) \qquad (2.4)$$

が成り立つような x があれば, $l=[1|x]$ が元の命題 (2.3) の反例になっていることがわかる. そこで, 命題 (2.4) を満たすような x がないかを探すことにする.

2) ステップ 1) とまったく同様に (2.2) から

$$A([],[3,4,5],y) \qquad (2.5)$$

を満たすような y があれば, $x=[2|y]$ が命題 (2.4) を成立させるような例になっていることがわかる. そこで, 今度は (2.5) を満たすような y がないかを探すことにする.

3) 公理 (2.1) によって, $y=[3,4,5]$ としたら, (2.5) が成り立つことがわかる.

以上から, $l=[1|x]=[1,2|y]=[1,2,3,4,5]$ が元の命題 (2.3) の反例になっていることがわかった.

2.8.3 Prolog と KL1

ここまでの説明は, 原則としては KL1 であっても Prolog であっても同じことである. 上に述べた証明のステップでは, 証明がうまく進むように, 適当な公理 (プログラム上では節) を選んでいったので, 一本道の簡単な証明ができた. しかし, 証明したい命題に適用できる公理が数多くあるもっと複雑な問題になると, 自動的に適切な公理の選択をすることは不可能である. そのような場合にどうするかが問題になる.

Prolog は, 最初の公理をまず選んで証明を進め, それで行き詰まったら公理を選ぶところまで後戻りして次の公理を選んでみるという, バックトラッキング機構を採り入れた. これで公理の選び方の問題は半分は解決したのだが, とりあえず選んだ公理を用いた証明が, 行き詰まることなく無限ループに入ってしまうことがあるので, すべての場合でうまくいくわけではない. Prolog のセマンティクスを変えずに並列処理するシステムの研究も行なわれているが, このあたりの方式については基本的には同じである.

バックトラッキングの処理は並列処理との相性があまりよくない. 何人かのグループで, ある方針に従って共同作業をしているときに, 行き詰まったからこれまでやってきた方法は捨てて別の方法でやろう, などと各人が勝手にいっているようでは, 到底効率のよい作業はできない. 誰か管理者が全体の作業状況を見ていて, 一斉に方針変更するように指示を出す, という方針ならもう少しましになるだろうが, そうやって管理者が把握できる人数は知れている. 計算機による並列処理の場合も, まったく同じような状況が起きる.

そこで KL1 では, どの公理を選ぶかを決めたら決して後戻りしない方針をとった. そのかわり, どの公理を選ぶかの条件となる部分を「ガード」として分離し, その条件を正確に判断できる情報が集まるまでは選択を待つことによって, 誤った選択をしないようにしたわけである. このような方針をとる論理型プログラム言語は, 自らの選択に賭けるというニュアンスから committed choice 型の並列論理型言語などと呼ばれることもある.

Prolog も KL1 も, ホーン論理についての健全な証明系になっている. つまり, どちらでも実行がうまく終了すれば, 出てきた結果がプログラムを公理系と解釈した場合の正しい定理になっているのである. このことは, プログラムの理論的解析などにとって便利な性質である.

一方, どちらも 完全な証明系ではない. Prolog でも KL1 でもうまく証明できない定理がある. カットや組込み述語などの付加機能のない純粋な Prolog では, この不完全性は, 実行が終わらないことがあるという形で現われる. KL1 では同様に実行が終わらないことがあるのに加えて, 失敗という形で終わることもある点が異なっている. このため, プログラム言語処理系をそのままホーン論理の自動証明器としてみると, Prolog に比べて証明できる範囲がやや狭まっているわけだ. KL1 はその部分をあきらめることによって, 並列プログラム言語として必要な機能を充実しているのである.

†1 自動定理証明では, 逆に公理から始めて証明できる命題を生成していく, ボトムアップの方法もある. エキスパートシステムなどでよく使われる前向き推論 (forward reasoning) というのも同じである.

2.9 むすび

KL1 の言語仕様と主要なプログラミング技法について概説してきた. KL1 のベースになった GHC という言語は, ごく簡潔な言語仕様しかもっていない. それがこれだけ多様なプログラミング技法を生み, これほど自由に並列処理を記述することができたことは, 筆者自身を含め言語設計にかかわった当事者たち自身も驚くほどである.

KL1 のプログラミング技法はまだまだ日進月歩の状態であるし, この程度の紙数では書き足りない点も多々あった. しかし, この解説で KL1 とはどんな言語なのか, どのようにプログラムを書けるのかについて, おおまかな感じだけでもつかめていただけたら, 筆者の感じている驚きを少しでも共有していただけたなら幸いである.

3.1 はじめに

並列推論マシンPIMは，並列論理型言語KL1を効率よく実行する大規模並列計算機である．KL1言語の実装に適したハードウェアを備えるとともに，知識処理に代表される動的で均質さの低い大規模計算の取扱いに適合したハードウェア構造をもつ．

以下では，並列推論マシンPIMの，マシンアーキテクチャとハードウェア構成について，一通りの内容をあまり細部に立ち入らないようにして解説する．KL1言語を効率よく実行するマシンの構成がどのようなものかの，イメージをつかんでおいていただくことと，マシンの規模や実現技術に関する基本的なデータを紹介することを目的としている．

まず3.2節ではマシンアーキテクチャの特徴をまとめ，続いて3.3節でPIMの各モデルの概略構成を説明する．そして続く各節では，PIM各モデルの構成要素おのおのに使用している技術について簡単に紹介していく．

3.2 マシンアーキテクチャの特徴

並列推論マシンPIMの，アーキテクチャ上の特徴を以下にまとめる．KL1向けに効率のよい記号処理を指向した部分，動的で均質さの低い大量計算問題向けの部分などがあるが，標準的構成要素技術を用いた部分もある．

a. 分散メモリ型 MIMD マシン

PIMの構造をマクロに見るならば，分散メモリ構造をもつ大規模MIMD計算機である．最大数百の計算ノードが高速のネットワークで接続された構造をもつ．基本的な方式の選択の一つとして分散メモリ構造を選んだ理由は，第一に規模の拡張性のよさと実現の容易さのためである．第二に，記号処理言語にとって重要なガーベジコレクション(GC)を含むメモリ管理に関しては，局所的なメモリ管理と大域的管理を分離するのに適当な構造であることなども選択理由である．

b. クラスタ構造

8台のプロセッサが共有バスと共有メモリにより密結合されたのがクラスタである．多数のクラスタが高速のネットワークで接続され，**マルチクラスタ構造**により全体システムを構成する．一つのクラスタのなかでは，言語システムのもつスケジューリング機能によって自動的な負荷バランスを実現しているため，プログラマは一つのクラスタを大きな計算能力と大容量メモリをもった一つの計算ノードとして取り扱うことができる．

クラスタ構造は，クラスタ内のプロセッサ間通信に関して，遅れ時間が小さくバンド幅が広いという特徴を提供している．クラスタ構造には次のような利点がある．一つ目は，クラスタ内では局所性の低い問題でも効率よく扱えることである．この種の問題については，クラスタという部分構造をもつほうが，全体が分散メモリ構造からなるシステムに比べて，次の2点で優れている．すなわち，通信オーバヘッドが低減できること，より小規模の問題に対してもよい並列処理効率が得やすいことである．

二つ目は，メモリの利用効率を高めることができ，分散メモリ接続に比べてシステム中の総メモリ容量を少なくしても，よい並列処理効率を得やすいことである．

この理由は，プロセッサが遅れ時間のきわめて短い通信路で接続されているために，通信遅れの影響をなくす(レイテンシの隠蔽)のに多数のプロセスを配置する必要がなく，その結果としてプロセッサ当たりの必要メモリ量を小さくできることである(次項参照)．またプロセッサごとの必要メモリ量が動的に変動しても，8プロセッサでメモリ共有するため，全体の必要量を平均化できることである．メモリのコストがシステム中に占める割合はきわめて大きいため，メモリサイズの低減はシステムコストの削減に効果が大きい[†1]．

マルチクラスタ構造の欠点といえば，一階層の分散メモリ接続に比べ構造が複雑なことである．けれども共有バス共有メモリによる密結合プロセッサは，次第に標準的なシステム構成要素になりつつあり，これを利用したマルチクラスタ構造の実現も次第に容易となることであろう．

c. プロセッサ性能に対し大容量のメモリ

筆者らの対象とする問題は，遠隔プロセス間で非同期的な通信を発生することが多いが，このような問題では計算ノード当たりに必要なメモリ量は，データ並列性をもつ問題の同期的処理の場合に比べて大きくなる．すなわち，プロセスが通信の応答を待つ平均時間は同期的処理に比べて長く，プロセッサの稼働率を維持するためには，待ち時間に処理するための余分な仕事が必要になる．その分大きなメモリを必要とする．

d. 高速ネットワーク

MIMD型並列計算機における高速の接続ネットワークは，すでに標準的な技術が確立されつつある．しかしながら，筆者らの対象とする問題では，データ並列性を備えた問題の同期的処理と比べ，プロセッサ負荷とネットワーク負荷の重さの比率が異なっているようである．分散メモリハードウェア上に大域名前空間を実現するような処理は，ネットワークメッセージの送受のたびに名前の管理テーブル参照などのプロセッサ負荷を発生する．つまり単純なデータ転送だけの場合に比べプロセッサの相対負荷が大きく，ネットワーク通信のバンド幅はしばしばプロセッサ性能で制限される．ネットワークバンド幅への要求はその分小さくなっている．

他方，KL1で書かれた応用では，遠隔プロセス間の同期のために少量のデータ転送が高い頻度で発生する．この場合，バンド幅より応答時間の短さが重要になる．

e. 並列キャッシュメモリ

クラスタ内の各プロセッサは，ライトバック型の並列キャッシュメモリを備える．基本的なキャッシュ制御方式は，標準になりつつある技術の一つを用いるが，いくつかの最適化と工夫を備える．プロセッサごとに独立のタクスを実行する場合に比べ，一つの問題を複数プロセッサが協力して解く場合には，プロセッサ間通信に起因するバス負荷がきわめて大きく，バス負荷の低減が課題となる．共有データの参照パターンの特徴を生かして，バス負荷を低減するバスプロトコルなどを工夫している．また排他制御機構として，キャッシュの状態制御を利用した語単位のロック機構を実現している．

f. 専用プロセッサ

プロセッサは，タグ操作やデータ型判定機構，参照ポインタを自動的に手繰り寄せる機構など，KL1を始めとする記号処理言語の実現に適した機構を備える．タグ操作機構は，単一代入言語で暗黙の同期を実現する場合にも役立つ．

プロセッサは，中間言語KL1-Bにコンパイルされたプログラムを効率よく実行することを目的に，それぞれのPIMの実行方式に合わせて設計された独自の機械語命令を備える．

パイプライン実行制御やRISC命令の技術も用いているが，これらは標準的技術である．

3.3 五つのPIMモデルの概要

5種のPIMモデルを開発してきたが，それぞれ異なるアーキテクチャおよび要素技術の組合せで実現しており，モデルごとにそれぞれ異なる研究開発上の役割を与えている．

a. PIM/p

PIM/pは最も規模の大きいモデルで，最大512台の要素プロセッサを接続できる(口絵1)．PIM/pは，アーキテクチャの研究用とソフトウェアの研究開発環境実現の両方の目的をもつ．

PIM/pは図3.3.1のマルチクラスタ構成を取る．最大64クラスタが接続できる．接続ネットワークはハイ

†1 たとえば，512台の要素プロセッサそれぞれに付随するメモリを一斉に増設するときのコストを想像されたい．

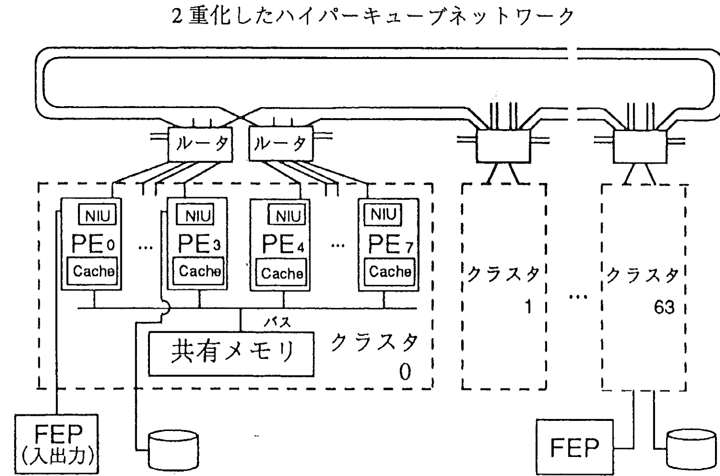


図 3.3.1 PIM/p の全体構造

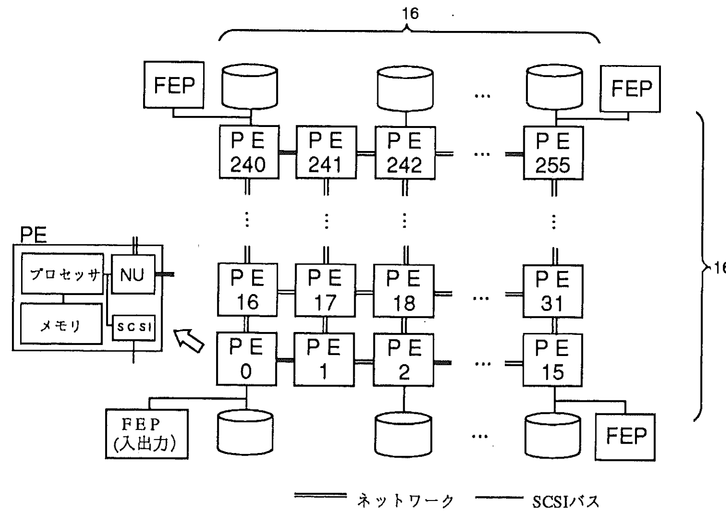


図 3.3.2 PIM/m の全体構造

パーキューブ構造であり，二組の同仕様のネットワークを各クラスタに接続してネットワークバンド幅を広げている．

クラスタは8台のプロセッサを含み，それらが共有バスと共有メモリで密結合されている．プロセッサは並列キャッシュ，ネットワークインタフェース NIU，I/O デバイスインタフェース (SCSI バス) を含む [7]．

PIM の要素プロセッサはどのモデルでも SCSI バスをもち，フロントエンドプロセッサ (FEP) とハードディスクが接続される．FEP は PSI-III(PSI-UX[†1])[8] であり，高機能のマンマシンインタフェース装置として機能する．

†1 三菱電機で製品化したものをこう呼んでいる．

b. PIM/m

PIM/m はソフトウェアの研究開発マシンであり，マルチ PSI/V2 との間に，オブジェクトコード レベルでの互換性を備えている．図 3.3.2 に示すように，最大で 256 台の要素プロセッサが 2 次元格子ネットワークで接続される (口絵 2)．最大 20 GB (32 台) のハードディスクと，たくさんの FEP が接続できる [9]．

c. PIM/c

PIM/c もマルチクラスタ構造を取り，最大 32 クラスタ，256 台のプロセッサが，クロスバーネットワークで接続される [10]．アーキテクチャ研究と，一部のソフトウェア開発に使用する．

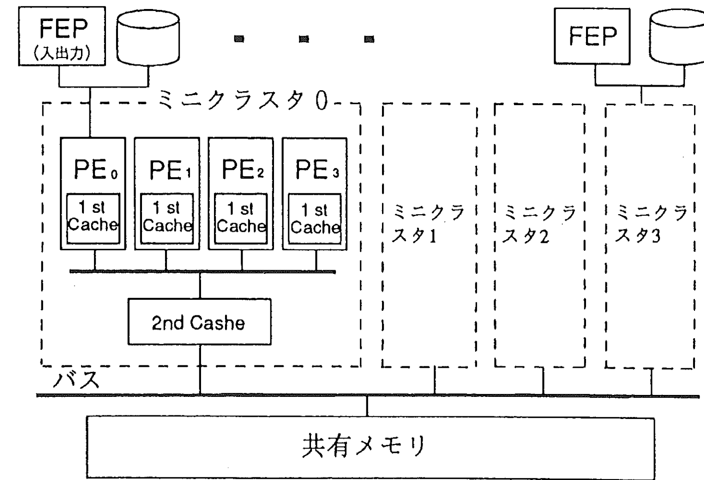


図 3.3.3 PIM/k の全体構造

表 3.3.1 各 PIM の全体構造のまとめ

	Topology	Number of Clusters	Total Number of PEs	Memory Size/Cluster
PIM/p	hypercube × 2	64	512	256 MB
PIM/m	mesh	256	256	80 MB
PIM/c	crossbar	32	256	160 MB
PIM/k	—	1 †	16	1 GB
PIM/i	—	2	16	320 MB
Multi-PSI/V2	mesh	64	64	80 MB

(† : four mini-clusters included)

d. PIM/k

PIM/k はクラスタ内のアーキテクチャを研究するためのシステムである．階層構成をもつキャッシュを実現しており，クラスタ構造をとりながらより多くのプロセッサを接続することを目指している [11]．4 台のプロセッサがローカルバスとセカンドキャッシュを共有し，ミニクラスタを構成する．四つのミニクラスタが共有のメモリバスを介して共有メモリに接続されている (図 3.3.3)．

e. PIM/i

PIM/i も，クラスタ内のアーキテクチャ研究用のシステムである．LIW 型の機械語命令をもつプロセッサや，放送型の並列キャッシュを試す (他の PIM の並列キャッシュは，無効化型である)[12]．

f. マルチ PSI/V2

マルチ PSI/V2 は，PIM の各モデルに先駆けて開発された並列推論マシンの初期の実験機である．64 台の要素プロセッサが 2 次元格子ネットワークで接続されている．その構成は，小規模な PIM/m とほぼ同等である．要素プロセッサとして PSI-II[13] の CPU ハードウェアを使用したためにこの名前がつけられた．KL1 処理系の研究開発と，並列ソフトウェアの研究開発に重用された．

各 PIM モデルの全体的構造について表 3.3.1 にまとめる．プロセッサ，ネットワーク，キャッシュシステムなどの構成要素の仕様について，以下の各項で述べる．

表 3.4.1 要素プロセッサの仕様

	Instruction set	Cycle time	LSI fabrication	Line interval
PIM/p	RISC + macro instruction	60 nsec †	standard-cell	0.96 μm
PIM/m	CISC (micro programmable)	65 nsec	standard-cell	0.8 μm
PIM/c	CISC (micro programmable)	50 nsec †	gate-arrays	0.8 μm
PIM/k	RISC	100 nsec	custom	1.2 μm
PIM/i	RISC	100 nsec †	standard-cell	1.2 μm
Multi-PSI/V2	CISC (micro programmable)	200 nsec	gate-arrays	2.0 μm

(† are design specifications. They are under testing with longer cycle time.)

表 3.5.1 ネットワーク

	# PEs in a cluster	# NIs in a cluster	Transfer Rate †
PIM/p	8	8	33 MB/sec ‡ ×2
PIM/m	1	1	8 MB/sec
PIM/c	8	1	40 MB/sec ‡
PIM/k	16	—	—
PIM/i	8	1	—
Multi-PSI/V2	1	1	10 MB/sec

(PE = processing element, NI = network interface)
(†: per channel, full duplex ‡: design specifications)

3.4 要素プロセッサ

KL1 言語を実行するときは，実行時のデータ型の判定を頻繁に行なう必要があるため，PIM の要素プロセッサはいずれもタグアーキテクチャをとっている．これはマルチ PSI も同様である．

PIM/p, PIM/i, PIM/k の要素プロセッサは RISC 型の命令セットを備え，PIM/m と PIM/c はマイクロプログラム制御により CISC 型の命令を実現している (表 3.4.1)．前者の機械語命令は，KL1-B よりさらに低いレベル (単純) であり，後者は，KL1-B がほとんどそのまま命令セットになっている．

PIM/p のプロセッサは，マクロコールと呼ぶ特徴のある機能を備えており，低いコストのサブルーチン呼出しが可能である [7, 14]．これによりコンパイル結果のオブジェクトコードサイズを小さく抑えながら，実行時オーバヘッドも増加させないで済む．PIM/p はほかにも KL1 実装用の命令として，ポインタを自動的に手繰る命令や，実行時塵集めの MRB 方式 [15] をサポートする命令などを備える．PIM/p のプロセッサは 4 段のパイプライン実行を行なっている．

PIM/m [9] のプロセッサは，マイクロプログラム制御で 5 段のパイプライン実行を行なう．命令セットは KL1-B そのもので，マルチ PSI/V2 とオブジェクトコード レベルでの互換性がある．高度なデータ型判定機構や，ポインタを自動的に手繰る機能をもつ．

PIM/i のプロセッサは，LIW(long instruction word) 型の命令セットを実現している．

3.5 ネットワーク

各 PIM のネットワークの仕様を表 3.5.1 にまとめる．

PIM/p では各プロセッサがネットワークインタフェース NI をもち，4 個の NI がネットワークノードを成すルータに接続されている．二組のハイパーキューブネットワークを使用して広いバンド幅を実現している．

PIM/m は，マルチ PSI と同じく 2 次元格子ネットワークを用いる．PIM/p と PIM/m は wormhole routing と呼ばれるメッセージ転送方式をとり，可変長の行き先アドレスつきメッセージが，ネットワークノードごとに動的に経路制御を受けて，目的プロセッサに届けられる．

PIM/c はクラスタに 1 台ずつ，クラスタコントローラと呼ばれる特別のプロセッサをもつ．このプロセッサは共有バスに接続され，ネットワークインタフェースの役割をはたす．ネットワークにはバンド幅の広いクロスバスイッチを用いている．

表 3.6.1 キャッシュシステムの仕様

	Coherence Control		Mapping	Cache Size	
	Protocol	# States †		Instruction	Data
PIM/p	invalidation	4	4 way	64 KB	
PIM/m	—	—	direct-map	5 KB	20 KB
PIM/c	invalidation	5	2 way	80 KB	
PIM/k	hierarchical invalidation	4	(1st) direct-map	128 KB	256 KB
			(2nd) 4 way	1 MB	4 MB
PIM/i	broadcasting	6	direct-map	160 KB	160 KB
Multi-PSI/V2	—	—	direct-map	20 KB	

(† does not include *locking* state.)

3.6 キャッシュシステム

KL1 プログラムはプロセッサ間で非同期的な通信を頻繁に発生し，これが共有バスの負荷をたいへん重くする．これを軽減するために，並列キャッシュのプロトコルを最適化した [16, 17]．バスの使用率低減に効果がある [18]．すべての PIM は，表 3.6.1に示すような，ライトバック型に属するキャッシュプロトコルを使用している．またキャッシュブロックの状態制御を活用した，語単位のロック機構 (排他制御機構) を効率よく実現している．

第4章 KL1言語処理系の実装

4.1 はじめに

ここでは並列推論マシン PIM 上の KL1 言語処理系の実装方式について解説する．本言語処理系は，並列推論マシンシステムの実現にあたってハードウェアの開発に捧げられたと同等か，もしくはそれ以上の情熱を注いで研究開発されてきたものであるといえるだろう．そのなかには，並列論理型言語処理系の実現技術に関するエッセンスがぎっしり詰まっており，FGCS プロジェクトが生み出した現在の世界最高水準に達する研究成果の一つとなっている．また，これらの技術は KL1 のような論理型言語に限らず，同様に破壊的代入を許さないような他言語 (たとえば純粋な関数型言語) の並列処理系を実現する場合にも共通に利用することのできる，基本的で重要な技術と考えられる．

ここでは，そのような KL1 言語処理系の基本的な実現方式についての解説を試みる．誌面の関係から，残念ながら処理系全体の細部にまで立ち入った説明を行なうことはできないが，基本的で重要な考え方ないし方式についてはなるべく多くを取り上げ，KL1のような言語を大規模な並列計算機に実装する場合の，全体的なイメージあるいは勘どころをつかんでいただけるように心がけたつもりである．

以下ではまずはじめに，KL1 言語の特徴に関する簡単な復習を行ない，そのような言語を効率よく実行するために言語処理系が備えるべき特徴をまとめる．次に言語処理系の方式全般についての概要を説明し，おおまかなイメージを与える．そのあと，言語処理系の諸機能を以下の4種類に大別して個々の実現方式を紹介する．すなわち

- 基本言語機能の実装
- 拡張言語機能の実装
- 効率向上のための機能と実現
- ノード間処理の実現

である．この部分では多くの図を用いることで，重要な方式をなるべく平易に解説するよう試みた．また，途中にはコラムとして KL1 のコンパイルコードに関する説明を挿入させてもらった．WAM(Warren Abstract Machine)[22] をごぞんじの読者には興味をもっていただけるのではないかと思う．

4.2 言語処理系の概説

4.2.1 KL1 言語の特徴

はじめに KL1 言語のもつ優れた特徴について簡単に復習しておこう．これらの言語機能は，知識処理をはじめとする，動的で均質さの低い大規模な計算問題のプログラミングや，効率のよい並列処理には必要不可欠のものである．KL1 言語処理系のほとんどはそれらの機能を効率よく実現することを目指したものであるともいえるだろう．

言語機能に関する特徴のうち，はじめの3項目は，論理的並列性 (concurrency) の記述に関するもので，KL1 の基礎となった GHC[1] の仕様に含まれる部分である．これらは，複雑な構造をもつ並列プログラムの記述能力を高めるための機能といえるだろう．すなわち

- **データフロー同期**

- **小粒度並列プロセス**
- **非決定性 (non-determinacy)**

である．これらの特徴により，KL1 の記述力は非常に高いものとなっているのだが，言語を実装する側からみると，そのような記述レベルの高い[†1]言語を効率的に実行させるために，頭を悩ませなければならないところともなっている．

そして，次にあげる三つの特徴は，Flat GHC から KL1 へと進化した時点で導入された拡張機能に関するものといえる．すなわち

- **プラグマ (実際の並列実行の指定)**
- **荘園機能 (ゴールの集合を制御する機能)**
- **効率向上のための組込み機能**

である．これらにより，プログラムの並列実行の制御が容易に行なえるようになり，また，PIMOS と呼ばれるオペレーティングシステムを KL1 によって記述することも可能となった．これはオペレーティングシステム自体が並列処理をコントロールし，また自身並列に動けるように設計/記述されなければならない筆者らのシステムにとっては非常に重要なポイントであった．PIMOS のような，それ自身複雑な並列プログラムである OS を，迅速に，そしてバグを少なく作るためには，OS も機械語のように低レベルな言語ではなく，KL1 のような高水準言語で書かれる必要があったのである．

4.2.2 KL1 言語処理系の特徴

さて，ここでは以上のような特徴をもった言語である KL1 の言語処理系を実装する上でのいくつかのポイントについての説明を行なおう．これらの言語実装方式に関する特徴のほとんどは，これまで説明してきた KL1 言語の特徴を最大限に生かしながら，なおかつ高い実行効率を実現することを目指したものである．

A. 小粒度並列プロセス

小粒度の並列プロセスおよび，それらの間での通信と同期を低いコストで実現した．実行優先度管理を取り入れたプロセススケジューリングなども，OS ではなく，言語処理系レベルで効率よく実現している．

†1 つまり高効率を得るためのコーディングのしづらい

これらが効率よく実現できたことで，多数の小粒度プロセスを活用した並列性の高いプログラミングを，現実の応用プログラムで用いることが可能となった．プロセス個数が多いと，負荷バランスの制御が容易になる，という効用もある．

B. 暗黙の通信

並列プロセス間の通信と同期は，プロセス間で共有された論理変数に対するユニフィケーションとして暗黙のうちに実行される．これはプロセスが単一のプロセッサ内に置かれた場合でも，別々のプロセッサに分散された場合でも同一である．後述するように，プロセスは，負荷分散プラグマ "@node(X)" を付加するだけでプロセッサ間を移動するが，このとき移動するプロセスが参照ポインタをもっていたならば，プロセッサ間での遠隔参照が自動的に生成されるのである．

この方式により，ハードウェアの分散メモリ構造は，プログラマにとってほとんど見えないものとすることができ，プログラミング時にプロセス間の通信と同期を設計するのにも，それらのプロセスがどこに置かれるか (同一プロセッサ内か別プロセッサか) といったことを意識する必要がなくなっている．

C. メモリ管理

KL1 のような単一代入言語では，ガーベジコレクションを含む動的なメモリ管理方式を効率よく実現することが不可欠であるが，KL1 の場合，使用するメモリ領域の割りつけはユーザの手をわずらわすことなく，言語処理系が自動的に行なう．ここでいうメモリ領域にはリストデータや配列データなどのユーザがプログラム中で陽に用いるデータのほか，プログラムコード，そして各種の制御用データなども含んでいる．また，それらメモリ領域が不要になったときも

- クラスタ内における，実行時ガーベジコレクション (参照カウント法のサブセット (4.6.2項 参照) を使用) と停止回収型のコピー方式ガーベジコレクションの組合せ
- 遠隔参照ポインタの実行時ガーベジコレクション (重みづけ参照カウント法 (4.7.1項 参照) を使用)

という複合方式のガーベジコレクションによって，自動的に回収されるのである．

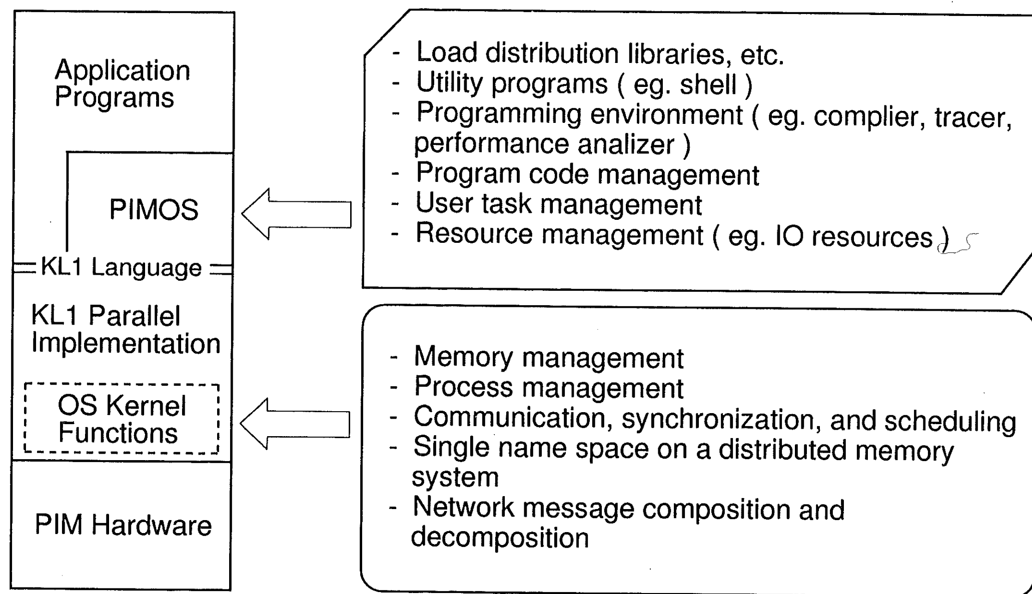


図 4.2.1 KL1 の実装と OS 機能の関係

D. 荘園の実装

KL1 におけるプロセスというのは粒度が小さすぎて，OS などが実行管理の単位として扱うには不適当である．そこで，そのような小粒度プロセスの集合を“荘園”としてまとめ，その荘園を実行管理の単位として扱うことにした．

荘園は UNIX におけるプロセスとほぼ対応づけて考えることができるが，UNIX とは異なり，KL1 では荘園の生成・管理は基本的に言語処理系がサポートする機能である．OS は言語処理系が提供する種々のプリミティブを用いて各種タスク (≒荘園) の制御を行なう．

KL1 言語処理系には，複数のプロセッサに分散されたゴールを荘園によって効率よく一括管理・制御するための数々の方式が実装されている．

E. OS 核を取り込んだ言語実装

図 4.2.1 に，KL1 の言語処理系と OS 機能との関係を示す．KL1 の言語処理系は，いわゆる OS 核の機能の多くをなかに取り込んだものとなっている．メモリ管理，プロセス管理とスケジューリング，通信と同期，大域名前空間，通信メッセージの組立て・送受などの機能は，言語実装のなかで実現しており，一方 PIMOS は，プログラミング環境やユーザインタフェースをはじめとする上層の OS 機能を実現している．

言語実装のなかに OS 核機能を取り込んだ理由は，それらの機能をなるべく低いコストで使いたいがためである．たとえば，小粒度の並列プロセス，それらの間の通信・同期，さらに暗黙の遠隔通信・同期，優先度つきスケジューリングなどは，すべて OS 核機能に関係するものであるが，これらは言語処理系のサポートなしでは，まったく書けないか，もしくは非常に非効率的な実現となってしまっただろう．

4.3 処理方式の概要

ここでは，どのような方式で言語処理系が設計されたのかを解説する．方式の基本となるのはゴール書換えモデルであり，それを基にして各種の処理が実現されている．

4.3.1 ゴール書換えモデル

前章までで説明したとおり，KL1 のプログラムは述語の集合からなり，各述語は

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n.$$

のようなガードつきホーン節の集合として表わされる．ここで，H,G_x,B_y はそれぞれ，節を構成する述語ヘッド，ガード述語，ボディ述語を表わしており，いずれも “symbol(Arg$_1$,...,Arg$_N$)” のように “述語名を表わすシンボル+0 個以上の引数” という形をしている．

論理的な意味を考えれば，上記の節は通常のホーン節と同様に

> $G_1 \wedge \cdots \wedge G_m \wedge B_1 \wedge \cdots \wedge B_n$ が満たされるならば H が成り立つ

と読むことができる[†1]わけだが，言語処理系の実装を行なう場合には，これとは逆に手続き的な意味を考えることが有用である．すなわち，上記の例では

> 述語 H が呼び出されたならば，その述語を構成する各節のなかからガード述語 $G_1, \ldots, G_m$ を受動的に満たすような節を選び，その節の各ボディ述語 $B_1, \ldots, B_n$ を呼び出す

となるわけである．

ここで，呼び出された述語中にガード述語を満たし得る節が複数ある場合でも，そのなかの一つの節だけが任意に選ばれ (コミット) 他の節は捨てられる，という KL1 の特徴をも考え合わせると，上記の処理は単純なゴールの書換え処理とみなし得ることがわかるだろう．この特徴により，KL1 では複数の選択肢があるところでも Prolog のようにバックトラックによる再試行に備える必要がないため，常に一本道で処理が進行するからである．

以上のような考察から，KL1 言語処理系はゴール書換えモデルを基本として設計された．図 4.3.1 にこのモデルの概念図を示してある．図中のゴールプールは実行可能なゴールを格納しておくための仮想的な容器であり

1) ゴールプール中からゴールを取り出す
2) ゴールに対応する述語の節のなかからガード部が成功する節を選択する
3) その節のボディゴールを新たに生成してゴールプールに戻す

というような処理の繰返しによって，KL1 プログラムは実行されるのである．

また，OS や多くのアプリケーションプログラムが必要とする実行優先度管理を言語レベルで効率よく実現するために，上記ゴールプールは実行優先度の異なる複数[†2]のサブゴールプールからなっており，各ゴールはその実行優先度に応じた適当なサブゴールプールに格納される．実行されるゴールは，その時点で空でない最大優先度のサブゴールプールから取り出されるわけである (図 4.3.2)．

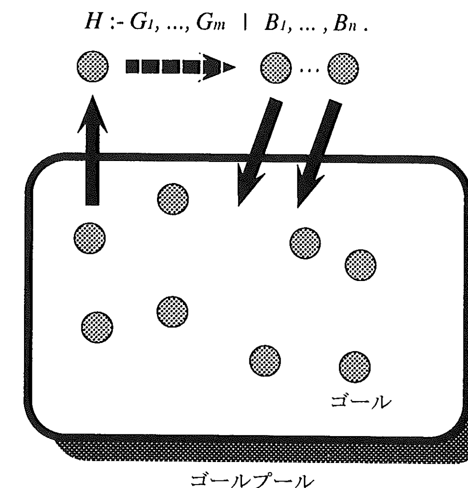


図 4.3.1 ゴール書換えモデル

このように分解能のよい実行優先度制御を言語レベルで効率的にサポートしたことで，KL1 は非常にプログラマビリティの高い言語となっており，たとえば後述する BestPath プログラムなどでは，この機能を用いてオーダのよいアルゴリズムをエレガントに実現することに成功している．

4.3.2 中断メカニズムによるプロセス間同期処理

KL1 では再帰呼出しゴールによってプロセスを表現し，プロセス間通信は各ゴールに共有される変数を介して行なわれるのが通常のプログラミングスタイルである．図 4.3.3 に最も単純なプロセス間通信の例をあげた．このプログラムでゴール `producer/1` は`msg` を生成し続け，またゴール `consumer/2` は `msg` を読み出し続ける．このような連続した通信を実現するために，ここでは CDR 部が変数であるようなリストデータ (`[msg|X]` など) を用いている．このような通信方

†1 ただし，$G_1 \ldots G_m$ に関しては受動的に，つまりヘッド引数を具体化することなく満たされなければならない．

†2 システムのコンフィギュレーションによって異なるが，数十～数千規模になる．

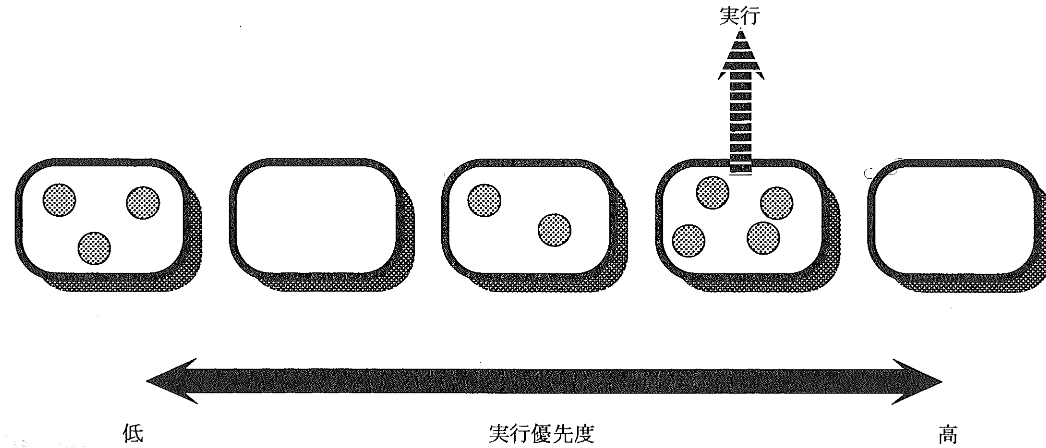


図 4.3.2 優先度別ゴールプール

```
?- producer(X), consumer(X).

producer(X) :- true | X = [msg|X2],
                    producer(X2).
consumer([msg|X]) :- true | consumer(X).
```

図 4.3.3 KL1 による単純なプロセス通信の例

法をストリーム通信と呼んでいることは 2.5.3 項で述べたとおりである．

さて，図のプログラムで実行の候補として consumer/1 のほうがproducer/1 より先に選ばれた場合[†1]，その consumer/1 の実行は producer/1 が msg を生成するまで待たされなければならない．これを可能にするのが KL1 の実行中断メカニズムであり，この機構によって，KL1 では複数のプロセスが互いに同期を取りながら実行することが可能となっている．

この例における consumer/1 のように，ガードユニフィケーションを満たせないためにただちに実行できないゴールは，図 4.3.4 のように，その実行中断の要因となった変数の保持する具体化待ちゴールプールのなかに入れられて，その変数が具体化されるのを待つことになる．いずれそのような具体化待ちゴールをもつ変数がボディユニフィケーションの実行により具

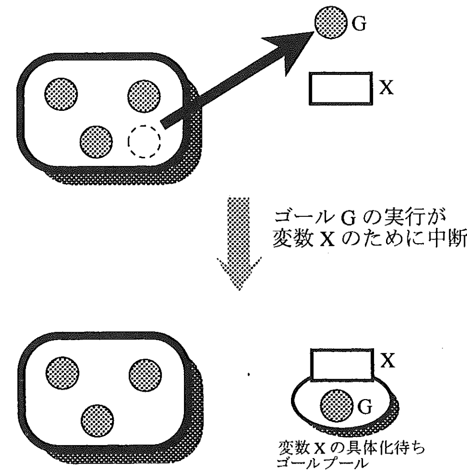


図 4.3.4 ゴールの中断処理

体化されたときには[†2]，待っているゴール群を通常のゴールプールに移すという処理が行なわれる．

この方式では，いったん実行が続けられないことが判明したゴールに関しては，その要因となった変数の具体化が保証されるまでは実行の候補となることがないため，そのようなゴールを単に通常のゴールプールに戻すような方式と比較して，不必要な実行可能性のテストを繰り返し行なわずに済む分，効率的な実現といえるだろう．

†1 KL1 では実行可能な複数のゴール間では実行順序に規定がないため，こういうことも起こり得る．

†2 先の例では "X = [msg|X2]" というユニフィケーションに相当する．

4.3.3 ノード間処理

以上説明してきた動作概要は，プロセッサ間にわたる処理にも言語処理系によって自動的に拡張される．ユーザは負荷の分散を実際に行なうところでのみ明示的な指定を行なうだけで，そのあとに続く処理に必要なプロセッサ間通信などは，すべて暗黙のうちに処理されるのである．

なお，前述のように PIM にはアーキテクチャの異なる数種類のマシンがあり，ある種のマシンでは共有メモリにより密結合されたプロセッサ群がクラスタを形成し，さらに複数のクラスタがネットワークによって疎結合される，という階層構造をなしている．この場合には，ここでいう外部参照処理が必要になるのはクラスタ間にわたる処理を行なう場合のみである．そこで，以下ではあいまいさを排除するために，外部参照処理が必要となる単位を "プロセッサ" ではなく "計算ノード" もしくは単に "ノード" と表わすことにしよう．PIM/m のようにすべてのプロセッサが疎結合されているマシンの場合には "ノード" はプロセッサに対応し，また PIM/p のように階層構造を採用したマシンの場合には各クラスタが "ノード" に対応する．

さて，KL1 では計算ノード間で負荷の分散を行なう場合，負荷分散プラグマを用いて

$$\ldots, GOAL\texttt{@node}(X), \ldots$$

のように記述する．この結果，$GOAL$ の実行は X で指定されるノードで行なわれることになるのである．言語処理系はこのような負荷分散を行なう場合

- 宛先ノードとの通信路の確立
- データ形式の変換
- 場合によっては論理変数の同一性保持のためノード間にわたる参照構造の生成

など，ノード間通信に必要な処理をすべて自動的に処理する．そのため，図 4.3.3 に示したプロセス間通信を行なうプログラムは，たとえば図 4.3.5 のように記述することで，容易にノードにまたがった処理へと拡張することが可能である．この場合，ユーザが行なう必要があるのは負荷分散プラグマ (@node(0) & @node(1)) の付加のみであり，それ以降のノード間にわたるプロセス間通信は言語処理系によって自動的に行なわれる．

```
?- producer(X)@node(0),consumer(X)@node(1).

producer(X) :- true|X = [msg|X2],
                   producer(X2).
consumer([msg|X]) :- true|consumer(X).
```

図 4.3.5 ノード間にわたるプロセス通信の例

ここまでのまとめとして，図 4.3.6 にシステム全体にわたる実行状態の概要を図示してみよう．この図はある瞬間に実行中のシステムのスナップショットを撮ったものと考えていただきたい．各ノードでは，複数のプロセッサが固有のゴールプール中のゴールを用いてゴール書換え処理を行なっている．いくつかのゴールやデータからはほかのノードを参照する外部参照構造が形成されているだろう．また，図に示したとおり，ある瞬間にはノード間を移動中でどのノードからも完全に離れているゴールやデータさえ存在し得るのである．

4.3.4 荘園

最後に，荘園の実現方針についても簡単に触れておきたい．前述のとおり，OS などが実行制御・管理の単位として扱うのが荘園である．

KL1 では，すべてのゴールは必ずいずれかの荘園に属するように管理されており，概念的には図 4.3.7 のように荘園はゴールの集合を外界から隔離する働きをもっているといえる．荘園と外界との接点は図に示した Control Stream と Report Stream と呼ばれる 2 本のストリームだけであり，ある荘園を管理するプロセスは Control Stream へ適当なメッセージを送ることで必要な制御を行ない，また，荘園内の実行状況は Report Stream を通して外界へ報告される．

荘園によってゴールの集合を外界から隔離することは，たとえば OS などの制御プログラムが子タスク実行中に発生したゴールの失敗などの例外事項に巻き込まれて一緒に失敗してしまう，といった事態の防止策にもなっている．

さて，このような荘園を効率的に実現するために，KL1 言語処理系は種々の組込み述語を提供している．

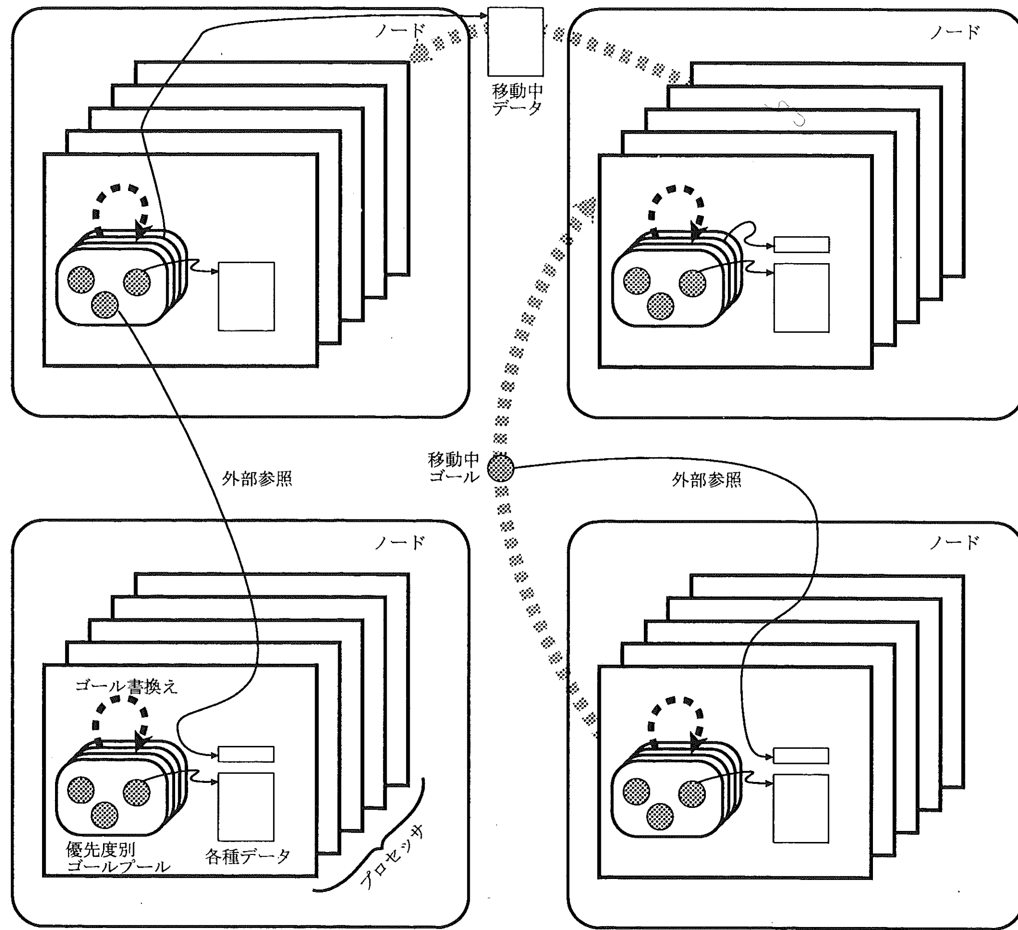


図 4.3.6 システム全体における実行の概要

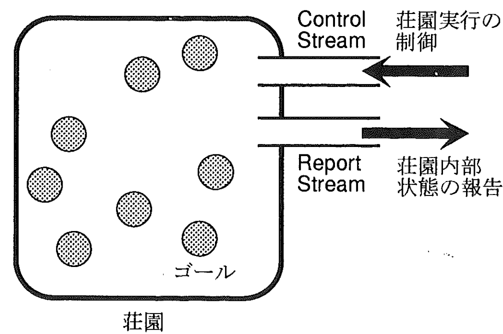


図 4.3.7 荘園の概念図

たとえば荘園生成用の述語が呼ばれた場合，言語処理系は新しい荘園を生成し，その荘園の Control Stream と Report Stream とを呼出し元に返す．そして，述語呼出し時に引数としてわたされるゴールを生成した荘園のなかに投入するのである．荘園は自由に入れ子にすることが可能であり，言語処理系は自ら管理する荘園の実体といえる荘園レコードを 1 荘園に対して 1 個生成し[†1]，各荘園レコードを荘園の入れ子構造に対応するツリー状構造をなすように管理している (図 4.3.8).

また，OS などによるタスクの資源管理を可能とするために，荘園は消費可能な一定量の資源[†2]をもっており，ゴールの実行は常に所属する荘園の資源消費を伴うことになっている．そのため, 荘園レコード中には実行中/停止中といった荘園の実行状態に関する情報のほかに，消費可能資源量およびそれまでに消費した資源量といった資源管理に関する情報も格納されている．ゴールは，所属する荘園が資源の枯渇などの原因で停止状態になっている場合には，実行することはできず，Control Stream から荘園へ実行再開の指示がくるのを待たされる．このような原因によるゴールの実行中断を実現するために，処理系は 4.3.2項 での説明とほぼ同様な機構が用意している．各荘園レコードには固有の再開待ちゴールプールが備えられているのである．

さらに，負荷分散プラグマによってゴールが複数ノードに分配される場合でも，荘園によるゴールの管理は正しく行なわれなければならない．そのため，荘園はノード間にまたがった構造をとることになり，荘園を含むシステムの全体的な概念図は図 4.3.9 のようになるだろう．この図のとおり，移動中のものも含めてすべてのゴールはいずれかの荘園に属し，また荘園の存在は計算ノードとは独立である．このように分散した環境での荘園構造の管理を効率的に行なうために，KL1 言語処理系では文献 [19] や以降の節で説明するような数々の工夫がなされているのである．

さて，以上で KL1 言語処理系の概要に関する説明は終わりである．以降の各項ではより詳細な言語処理系の実装方式に関して説明を加えていくことにしよう．

†1 実際の処理系ではノード間に分散されるゴールをすべて荘園レコードで集中管理するのでは処理が重くなるため，各ノードには荘園のローカルな出先機関である里親レコードが置かれている．

†2 現在はゴールの呼出し回数を資源として使用している．

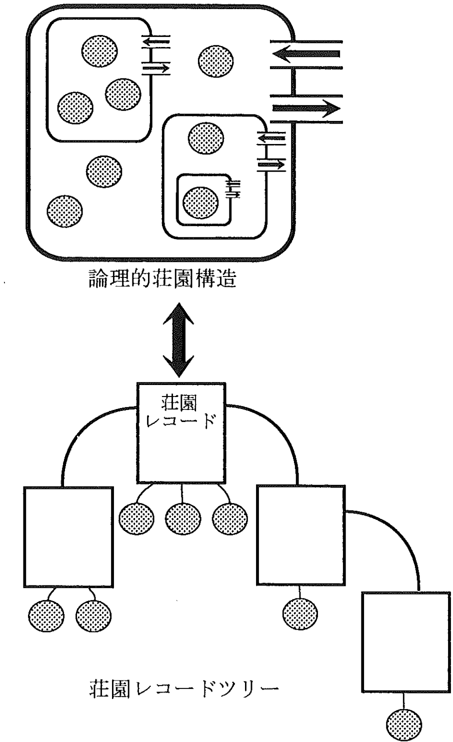


図 4.3.8 荘園レコードツリー

4.4 基本言語機能の実装

この節では，メモリ管理，ゴールの書換え，ユニフィケーション，組込み述語といった KL1 の GHC 部分である基本言語機能の実装方式について示す.

4.4.1 メモリ管理

KL1 は，Prolog や Lisp と同様にユーザはメモリ管理から解放されており，メモリは必要時に自動的に割り当てられる．KL1 処理系では，次の 3 段階の GC をサポートしている．

(a) 多重参照ビット (MRB) を用いた局所実時間 GC
(b) コピーイング方式による局所一括 GC
(c) プロセッサ間分散データの実時間 GC

(a) で回収されたセルはその大きさごとにフリーリストにつながれ，再利用される．(a) および (c) については，後の節で詳述する．

PIM では，コピーイング方式の GC を採用しており，ヒープエリアを二つに分割して用いる．そのほかに，GC によってアドレス位置が移動されない固定長の制御情報エリアをもつ．制御情報エリアには，処理系のための各種情報，フリーリストの先頭，後述するゴールスタックテーブル，プロセッサ間通信用のバッファ・輸出入表，組込み述語の中断のためのコードなどが置かれる．

4.4.2 ゴールの書換え

実行可能ゴールは，実行可能ゴールスタックと呼ばれる LIFO 型のキューで管理される．スタックの一番上に積まれている実行可能ゴールが取り出され，書換えが行なわれ，書き換えられたゴール群は再びスタックに戻される．すなわち，ゴールの書換え順に関して深さ優先のスケジューリングになる．

書き換えられたゴール群は，それぞれ図 4.4.1 に示すゴールレコードにその実行環境が保存され，実行可能ゴールスタックに積まれる．実行環境として，引数

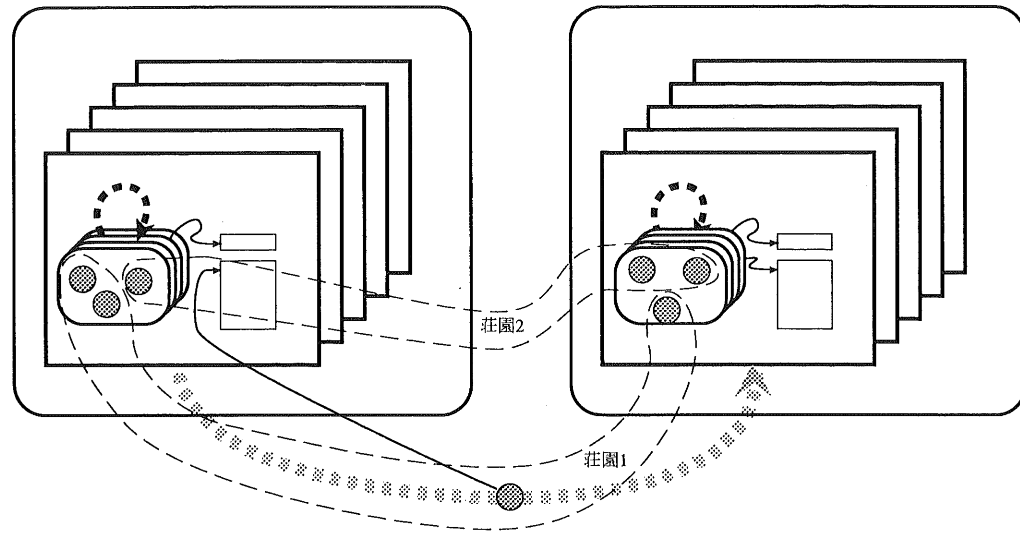


図 **4.3.9** 荘園を含むシステム全体の概念図

goal!	引数個数 N
	引数 0
	引数 1
	:
	引数 $N-1$
	:
code!	実行コードの先頭アドレス
	優先度
	里親
	次のゴール

図 **4.4.1** ゴールレコード

のほかに，実行コードの先頭アドレス，実行優先度，荘園機能のプロセッサごとの代理人である里親が保存される．実行優先度，里親については後述する．ゴールレコードは 16 語のセルで，ベクタ，ストリングなどの構造体と同様に，16 語セルのフリーリストから取り出される．引数が 11 個を越える場合は，ゴールレコードのほかに必要な大きさのセルを確保して，ゴールレコードにつなげる．図 4.4.2 に示すように，ゴールスタックはこれらのゴールレコードのリンクによって実現されており，通常のプログラミング言語の call/return のような連続領域のスタックとは異なる．

図 4.4.2 に，以下に示す述語の書換えの例を示す．

```
p(A00,A01) :- true | g1(A10,A11,A12),
      g2(A20,A21), g3(A30,A31,A32,A33).
```

実行中のゴール p/2 が g1/3，g2/2，g3/4 という 三つのゴールに書き換えられる．

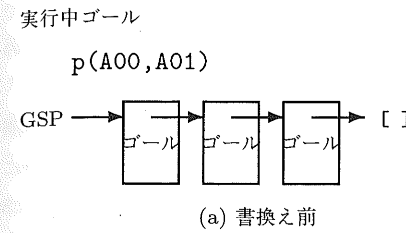
GSP は，実行可能ゴールスタックの先頭を指すポインタとする．KL1 の言語仕様では，書き換えられたゴール群の実行順序は規定していないが，人間の自然な感覚にあわせて，左から順に行なわれるように，g3/4，g2/2，g1/3 のように，逆順にスタックに積まれる．すべての書換えが終わると，ゴールスタックの先頭のゴールを取り出して書換えを行なう．このとき，最後に積まれたゴールを直後に取り出すという無駄な処理を省くために，g1/3 はスタックに積まないで，即座に実行するという最適化が行なわれる．また

```
p([]) :- true | true.
```

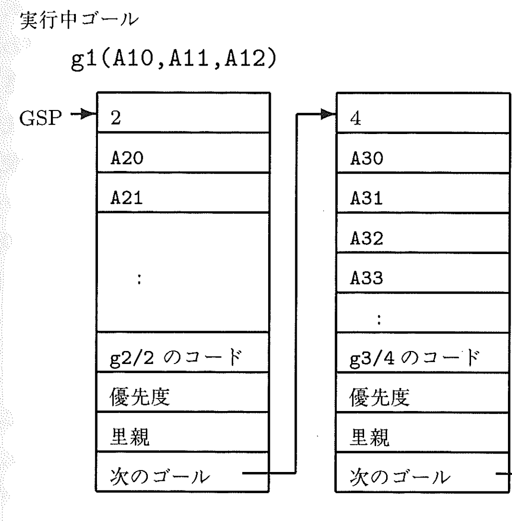
のように，ゴールを生成しないような述語はその終了時に，実行可能ゴールスタックの先頭ゴールレコードを取り出して，その実行を開始する．

4.4.3 ユニフィケーション (中断/再開)

論理変数が値をもたない (未定義の) ためにゴール実行が中断すると，そのゴールは中断の原因となった



(a) 書換え前



(b) 書換え後

図 **4.4.2** ゴールの書換え

論理変数から指され，変数が値をもつのを待つ．その間は，別のゴール実行が行なわれる．変数に値が決まると，中断していたゴールは実行可能となり，実行可能ゴールスタックに積まれる．ここでは，ゴールの中断/再開の直接のきっかけとなるユニフィケーションについて述べる．

ユニフィケーションには，ガード部で記述される受動的なものと，ボディ部で記述される能動的なものの 2 種類がある．

A. ガード部のユニフィケーション

候補節の選択，未定義変数の具体化待ち，構造体要素の取出しに用いられる．未定義変数の具体化待ちによって，プロセス間の同期が行なわれる．未定義変数

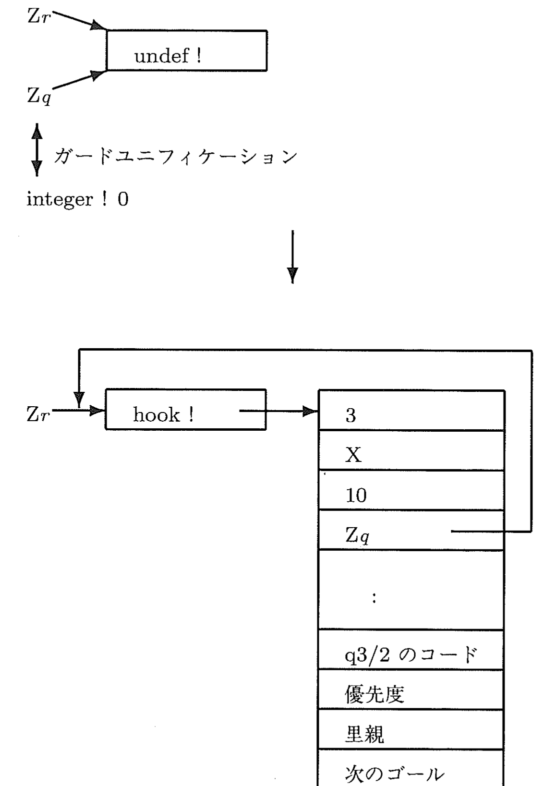


図 **4.4.3** 中断したゴール

への代入は行なわれず，中断状態に入る．述語が中断される KL1 プログラムの例を以下に示す．r(Z) の実行の前に，変数 Z が未定義のまま q(X,10,Z) が実行されたものとする．

```
p(X) :- true | q(X,10,Z), r(Z).
q(X,Y,0) :- true | s(X,Y).
r(Z) :- true | Z = 0.
```

図 4.4.3にゴール中断の過程を示す．まず，未定義変数 Z のためのセルが確保され，ゴール r/1 とゴール q/3 からの 2 本のポインタによって指される．この変数セルが 二つのゴールの同期，データ受けわたしの場となる．同図では，両者のポインタを区別するために，それぞれ Z_r，Z_q で示した．

まず，未定義変数 Z_q と整数 0 とのガードユニフィケーションによって，述語 q/3 の実行は中断され，Z_q は具体化待ち変数 (フック変数) になり，図 4.4.3 に示すようなゴールレコードを指す．このゴールレコード

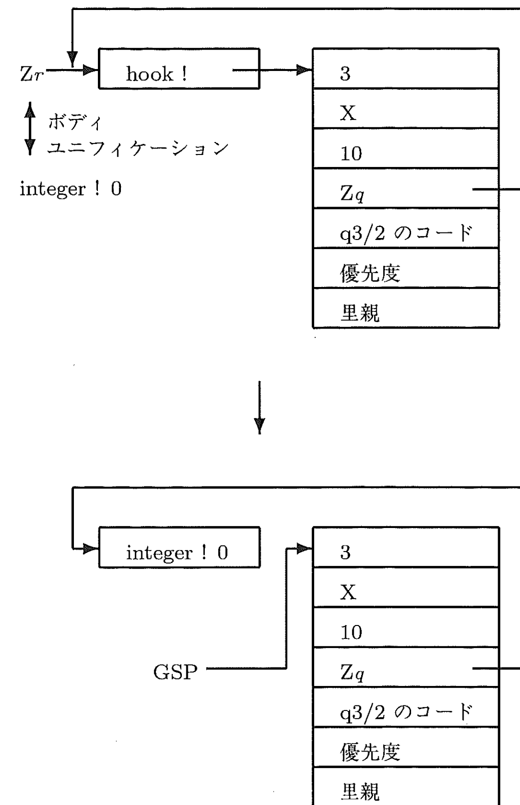


図 4.4.4 中断した述語の再開

には，q/3 の実行前の状態が保存される．変数 Z のもう一つのポインタである Zr に値が代入されると，述語 q/3 の先頭から再開される．

述語の中断はガードユニフィケーションのほかに，単に具体化を待つ組込み述語 (wait/1) や後述するガード組込み述語の入力待ちなどによっても生じる．

```
p0(0) :- true | q(1).
     % ガードユニフィケーション
p1(X) :- wait(X) | q(X).
     % wait/1
p2(X) :- X > 0 | q(X).
     % 組込み述語の入力
```

B. ボディ部のユニフィケーション

ボディ部でのユニフィケーションでは主に，未定義変数への値の代入や，フック変数の具体化による中断状態のゴールの再開を行なう．構造体同士のユニフィケーションや，構造体要素の取出しを行なうこともできる．しかし，ユニフィケーションの失敗は荘園への例外事項の報告となる．

フック変数の具体化による中断状態のゴールの再開は，プロセス間の同期とデータの受けわたしを同時に行なっているとみることもできる．図 4.4.4 は図 4.4.3 で中断したゴールを指すフック変数と，整数 0 とのボディユニフィケーションによる再開の例を示している．

ゴール r/1 からのポインタ Zr を経由して，受けわたしの場となった変数セルに 整数 0 を書き込み，さらに，ゴールレコードを実行可能ゴールスタックの先頭に登録する．Zr 自身はその後，ゴール r/1 内で使用されないので消滅する．ゴール q/3 からのポインタは整数 0 のセルを指すことになるので，変数 Z の値は整数 0 になる．

4.4.4 組込み述語

KL1 でも Prolog のような他の論理型言語と同じように，基本述語は，組込み述語として提供している．次のような基本述語が組込み述語として提供されている．組込み述語の引数はすべて入力あるいは出力のいずれかに決められており，無印が入力引数，^ 印が出力引数を示す．

1) タイプチェック

   atom(X), integer(X), vector(X,^Size) など

2) 比較

   less_than(Integer1,Integer2) など

3) 演算

   add(Integer1,Integer2,^Output), floating_point_add(Float1,Float2, ^Output) など

4) 構造体操作

   vector_element(Vector,Position, ^Element,^NewVector), set_string_element(String,Position, ^OldElem,NewElem,^NewString)

5) 特殊機能

   apply(Code,Args), current_node(^NodeNumber,^TotalNode), system_timer(^High,^Low) など

組込み述語は高速に実行するために，ユーザ定義述語のようにゴールレコードを作って実行可能ゴールスタックに積むのではなく，引数をレジスタでわたして直接実行される．呼出しの方式は PIM の機種によって，あるいは出現頻度によって異なるが，(a) ファームウェアで直接実行，(b) マクロコール (低レベルのサブルーチン呼出し)，(c) 低レベル命令へのコンパイルなどにより実現されている．

組込み述語は，ガード部で記述できるものと，ボディ部で記述できるものの 2 種類に分けられる．

1) ガード組込み述語

   主に，節の選択，同期に用いられる．整数加算のオーバフローのようなエラーが生じた場合，その組込み述語は失敗し，次の候補節に進む．入力引数が未定義で実行が開始できない場合は，いったん中断し，次の候補節を試みる．すべての候補節を試みてすべて失敗の場合は，述語が失敗したことを意味し，例外事項としてエラー報告する．また，中断した節があった場合は，ガードユニフィケーションで中断したときと同様に，その述語のゴールを生成し，未定義の入力引数をフック変数としてそのゴールレコードを指す．

2) ボディ組込み述語

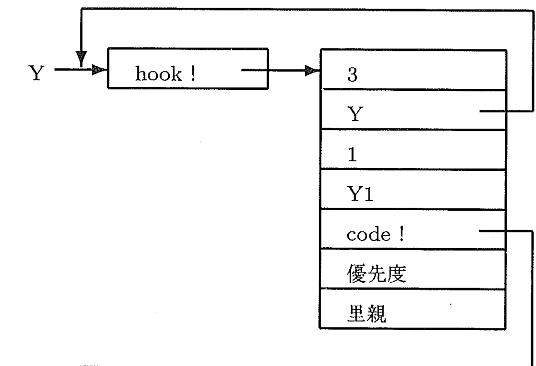
   主にタイプチェック，比較以外の基本操作に用いられる．ボディではエラーが生じた場合，他の候補節はないので，そのまま例外事項として荘園に報告する．入力引数が未定義の場合は，値が確定するまで実行が開始できないので，ゴールレコードをつくって中断する．

   以下に組込み述語が中断する例を示す．inc/2 が q/2 の前に実行されたものとする．

```
p(X,Y1) :- true | inc(Y,Y1), q(X,Y).

inc(N,N1) :- true | N1 := N + 1.
```

入力引数が未定義で中断した場合，ゴールを生成し，未定義の入力変数はフック変数としてそのゴールレコードを指す．このとき，コンパイルコードのなかには，このゴールを実行するコードはない．



中断された組込み述語を
実行するためのコード群

```
d_add(In1,In2,Out) :- true |
     add(In1,In2,Out).
d_subtract(In1,In2,Out) :- true |
     subtract(In1,In2,Out).
d_multiply(In1,In2,Out) :- true |
     multiply(In1,In2,Out).
   .......
```

図 4.4.5 組込み述語が中断したときの内部状態

PIM では，中断された組込み述語の実行のためにすべてのボディ組込み述語のための実行コードをシステムがもち，中断したゴールレコードはその該当部分を指す．中断が解除されたときにそのコードを実行する．図 4.4.5 にその内部状態を示す．

4.5 拡張言語機能の実装

4.5.1 割込み処理

PIM では，KL1 処理における多種多様のイベント処理を節の終了時にまとめて行なう．KL1 の節では組込み述語の実行，ユーザ定義述語のエンキューが行なわれるが，その間は要求のみにとどめ，それに対する処理は行なわず，節の終了時に行なう．以下に割込み要因とそれに対する処理を示す．

1) 局所一括 GC 要求

   KL1 では一部，組込み述語によって動的に要求されるものを除いて，コンパイラは節内で使用されるメモリを静的に見積もることができる．PIM

KL1 のコンパイルコード

```
p([In|Rest],List) :- true | List = [Result|Tail], q(In,Result), p(Rest,Tail).
p([],List) :- true | List = [].
```

図 (a)　KL1 プログラム

```
(01)        try_me_else(L3)                % p(
(02)        switch_on_non_list(r0,L2)      %   [
(03)        read_variable(r2)              %    In|
(04)        read_variable(r3)              %       Rest],List) :- true |
(05)        put_reused_structure(r6,r0,2)  %    R06 = [
(06)        write_variable(r4)             %           Result|
(07)        write_variable(r5)             %                  Tail]
(08)        get_bound_value(r1,r6)         %    List = R06,
(09)        set_value(g1,r3)               %      Rest,
(10)        set_value(g2,r5)               %           Tail
(11)        enqueue_goal(p/2)              %    p(          ),
(12)        put_value(r0,r2)               %      In,
(13)        put_value(r1,r4)               %         Result
(14)        execute(q/2)                   % q(          ).
(21)  L2:   try_me_else(L3)                % p(
(22)        wait_atom(r0,[])               %   [],List) :- true |
(23)        get_atom(r1,[])                %    List = []
(24)        proceed                        %
(31)  L3:   suspend(2,p/2)
```

図 (b)　KL1-B コード

KL1-B は KL1 プログラムを効率よく実行するための中間言語 (抽象命令セット) で，各 PIM に共通である．KL1-B の役割は Prolog における WAM(Warren Abstract Machine)[22] と同様である．KL1-B コードをどのようにして実行するかは，機種依存である．たとえば，CISC マシンである PIM/m は，ファームウェアによるインタプリタにより直接実行し，RISC 風マシンである PIM/p は，さらに機械語レベルまでコンパイルして実行する．図 (a) に示す KL1 プログラムは入力ストリームからデータを受け取り，述語 q/2 にて加工し，その結果を出力ストリームに流すプログラムである．図 (b) にコンパイル結果を示す．レジスタ番号は 0 から始まるものとする．図中，r はレジスタ，L はラベル，g はゴールレコードのオフセットを示すものとする．

(02)-(04) で入力リストのガードユニフィケーションを行なう．(05)-(07) で出力リストを生成する．このように構造体に関しては，KL1 では逐次実行を前提とした Prolog 用の WAM とは違って，ガードでは同期，要素取出し，ボディでは生成，代入といったようにはっきりと区別することができ，別命令を用いる．(05) で出力リスト用のリストセルを確保する．入力リストのセルが単一参照の場合はすでに不要となっているので，そのセルをそのまま使用する．単一参照でない場合は，新たにリストセルをフリーリストから取り出す．(08) で，出力ストリームのボディユニフィケーションを行なう．(09)-(11) にて，述語 p/2 の引数をゴールレコードに設定し，そのレコードを実行可能ゴールスタックに積む．(12)-(14) にて，述語 q/2 の引数をレジスタに設定したあと，その実行を行なう．実行可能ゴールは LIFO 型のスタックで管理されているので，後ろから順にスタックに積んでいく．最後のゴールはスタックに積んでも即座に取り出されることになるので，スタックには積まないで直接実行する．

実行が失敗したとき，あるいは入力ストリームが未定義で中断するときは，(11),(21) の try_me_else 命令で指しているラベル L3 の (31) の suspend 命令に進み，失敗/中断の処理を行なう．

では述語開始時に一定量のメモリを確保し，節の終了まで一括 GC が起きないことを保証することができる．ただし，現在の処理系では十分に大きな量を確保して見積りはしていない．

新しく任意のサイズのベクタやストリングを生成する new_vector や new_string のように，動的にメモリを要求する組込み述語が実行されたが，空き領域がなく確保できない場合は，その時点で GC を行なうのではなく，GC 要求フラグを立て，その組込み述語は中断し，節の終了時に GC を行なう．

2) 高優先度のゴールの出現

実行中のゴールより高い優先度をもつゴールがエンキューされたとき割込み要因となり，節の終了時に検出されて実行が移る．4.5.2項にて詳しく述べる．

3) 外部割込み

ネットワークからのパケットの到着や入出力装置からの割込みの処理を行なう．

4.5.2　実行優先度

KL1 には，ゴールの実行優先度をゴールごとに指定する機能がある．複数のゴールが同一のノード内で実行可能である場合，原則として優先度の高いゴールから先に実行する．優先度はノード内のみで有効であり，ノード間にまたがっては保証されない．したがって，優先度はシステム全体としては絶対的なものではなく，効率向上のための指針にすぎない．実行順序の保証は，変数を介した待合せによって行なうべきものである．KL1 にはゴールの優先度を指定する場合，所属する荘園の上限/下限で決まる範囲内の割合で指定する方法と，その親ゴールの優先度との相対値で指定する方法の 2 種類が用意されている．たとえば，割合で指定する方法では次のようなプラグマを用いて記述する．

Goal @ priority(割合)

優先度には，物理優先度と論理優先度があり，それぞれのゴールは論理優先度をもつ．ここまでの記述で単に優先度といっていたものは，論理優先度を指す．処理系内にはあらかじめ指定されたレベル数の物理優先度があり，処理系は論理優先度を物理優先度に変換してから用いる．したがって，物理優先度の範囲が小さい場合は，論理優先度を変えても物理優先度が変わるとは限らない．

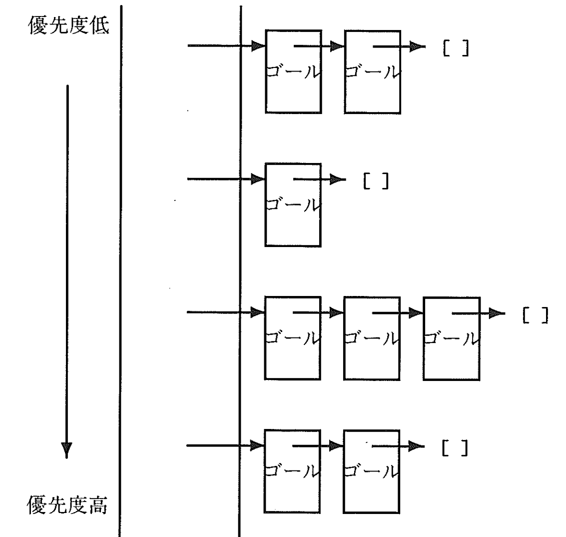


図 4.5.1 物理優先度別ゴールスタックテーブル

PIM では，物理優先度の数だけの実行可能ゴールスタックをもつことにより，実行優先度を実現している．たとえば，マルチ PSI, PIM/m では，通常 16 K 個のゴールスタックをもつ．この幅は，マシン立上げ時にパラメータとして与えることもできる．

図 4.5.1 に，物理優先度ごとに設けられたゴールスタックエントリテーブルを示す．それぞれは，主記憶上の GC による移動がない制御情報エリアにとられている．現在実行中のゴールの物理優先度すなわち，その時点の最高の物理優先度をもつゴールスタックは，高速化のためにレジスタ GSP 上にキャッシュされる．

4.5.3　KL1 プログラミングサポート

KL1 には，そのデバッグや性能チューニングについて，並列言語であるがゆえの難しさがある．PIM のオペレーティングシステム PIMOS はデバッガ，プログラムチューニングツールといったプログラム開発環境をもつ．それらの機能の多くは言語処理系によって実現されている．KL1 言語処理系は，次のようなプログラミングサポートを行なっている．

- トレース/スパイ

  KL1 のトレーサは，ゴールの親子関係に注目してトレースする．あるゴールが書き換えられて生じた子ゴール群を報告し，指定に応じてその子ゴールをトレースするといったように行なう．このように，並列に他のゴールが実行されても，注目している部分のみを報告する．スパイについても同様で，スパイをかけた述語が注目しているゴールの子孫のゴールで実行されたときのみ報告する．

- 永久中断ゴール検出

  KL1 におけるゴールの中断は，その入力引数が決定していないときに起きる．しかし，変数名のミススペルなどのエラーのため，参照パスが途切れて，永久に入力引数が決定しないといった状態がある．この中断ゴールは，プログラムからのポインタが切れているので，ソフトウェアからはその原因を探る方法がない状態にある．処理系は，コピーイングによる一括 GC や，MRB による即時 GC によってこのような永久中断ゴールを検出し，報告する機能をもつ [20, 21]．

- 荘園プロファイラ

  指定した荘園下で呼び出された述語の回数および中断した回数を報告する．性能チューニングを行なうときに，注目すべき述語を特定することができる．

- プロセッサプロファイラ

  個々のプロセッサの実行状態を報告する．通常実行，アイドル，GC，プロセッサ間通信処理の各時間やプロセッサ間通信のパケットの種類などのデータを得ることができる．PIMOS ではこれらのデータを用いて，視覚的にリアルタイムにプロセッサの稼働状況をユーザに提供することができ，負荷の偏りを検出することができる．

これらを用いた開発環境および実装方法については，次の PIMOS の章のプログラミング開発環境の節で詳しく述べる．

4.6 効率向上のための機能と実現

4.6.1 組込みストリームマージャ

2.6.4項で述べたように，KL1 では組込みストリームマージャを用意している．このマージャは，入力ストリームの数によらず一定のコストでデータを中継することができる．

マージャはたとえば，図 4.6.1のように KL1 で記述することができる．このマージャ(merge/3) の構成は図 4.6.2 (a) であり，マージ処理は

1) MHOOK セルとリストのユニフィケーション
2) MHOOK セルにフックしていたゴール merge/3 のリジューム
3) merge/3 の第 3 引数への出力
4) merge/3 のサスペンド

となる．1) は「マージャへデータを送る」というユニフィケーションとは異なった特別な処理になるのであるが，ここでは通常のユニフィケーションが行なわれ，それによってリジュームされたゴールがマージ処理 2) ～ 4) を行なう．三つの処理 2) ～ 4) のうち本当に必要なのは 3) だけであるが，ゴール実行の形でマージ処理を実現するために 2) と 4) (リジュームとサスペンド) が加わってしまった．

これに対して組込みマージャ(merge/2) では，マージ処理をゴール実行の形ではなく，1) のユニフィケーションの処理の一環として実現している．これには以下のことが必要であった．

- 入力ストリーム変数とリストのユニフィケーションの段階で，マージ処理であることがわからなくてはならない．
- 入力ストリームがすべて閉じられたことが検出できなくてはならない．

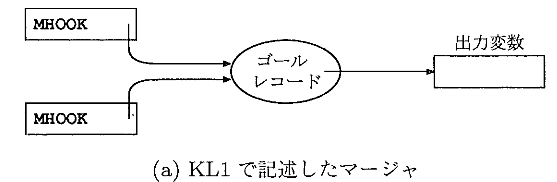
以上のことは「MHOOK セルとそれにフックしている (サスペンドしている) ゴール」(図 4.6.2 (a)) の代わりに「MGHOK セルとそれが指すマージャレコード」(図 4.6.2 (b)) を設けることによって実現した．すなわち

- MGHOK セルはマージ処理専用のデータセルなので，入力ストリーム変数とリストのユニフィケーションの段階でマージ処理であることがわかる．

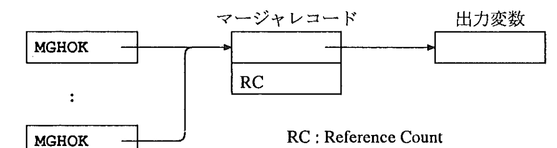

```
merge([A|Is1],Is2,Os) :- true | Os = [A|Os2], merge(Is1,Is2,Os2).
merge(Is1,[A|Is2],Os) :- true | Os = [A|Os2], merge(Is1,Is2,Os2).
merge([],Is,Os) :- true | Is = Os.
merge(Is,[],Os) :- true | Is = Os.
```

図 4.6.1 KL1 で記述したマージャ(merge/3)

- マージャレコードの RC (Reference Count) は入力ストリームの本数を示し，入力ストリームの増減に従って増減する．このため，RC を −1 した結果ゼロになったら，すべての入力ストリームの閉じられたことが検出される．



(a) KL1 で記述したマージャ



(b) 組込みマージャ

図 4.6.2 マージャの構成

組込みマージャ(merge/2) の処理を以下に示す．

開始： merge/2 を呼ぶと，MGHOK セルとそれが指すマージャレコードが作られる (図 4.6.3 (1))．RC の値は 1 である．

入力ストリームの増加： 新しい入力ストリーム変数を要素とするベクタをそれまでの入力ストリーム変数 (MGHOK セル) とユニファイすると，入力ストリームが増える．このときマージャレコードの RC は +1 される (図 4.6.3 (2))．

入力ストリームへのデータの入力： 入力ストリーム変数とリストをユニファイすると，データがマージャへ流れ出力変数にユニファイされる (図 4.6.3 (3))．

入力ストリームの減少： 入力ストリーム変数と NIL をユニファイ すると (入力ストリームを閉じると)，RC は −1 される (図 4.6.3 (4))．

終了： 最後の入力ストリームが閉じられると RC がゼロになり，すべての入力ストリームが閉じられたことが検出され，マージ処理が終了する (図 4.6.3 (5))．

4.6.2 多重参照ビット (MRB)

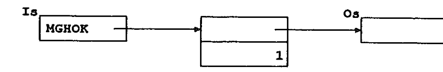
KL1 は論理型言語であるので破壊的代入操作がない．たとえば，一つの変数 X に対して二つのユニフィケーション X = a と X = b を行なおうとした場合，(たまたま) 先に試みられたユニフィケーションは代入操作として行なわれて成功し，試みが (たまたま) あとになったユニフィケーションは「値が一致するかどうかのチェック」として行なわれて失敗する．構造体についても「要素を書き換える」という操作はない．

A. 構造体の要素の更新

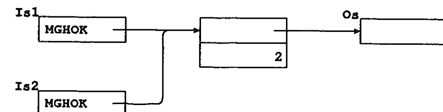
構造体の要素の更新は「新しい値を要素とする構造体を新しく作りそれを参照する」という処理によって行なう．具体的には

```
set_vector_element(Vec,
                   Pos,Elem,NewElem,NewVec)
```

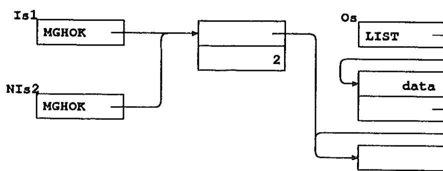
という組込み述語が更新処理を行なう．Vec に更新したい構造体 (ベクタ)，Pos に更新したい要素の場所 (インデクス値)，NewElem に新しい要素の値をそれぞれ指定すると，Pos 番目だけが NewElem で，残りは Vec とまったく等しい構造体が作られ NewVec に，また Pos 番目にあった要素の値が Elem に結合される．たとえば



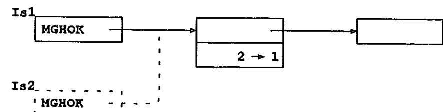
(1) merge(Is,Os) の実行 (マージ処理の開始)



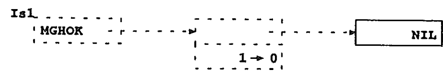
(2) Is = {Is1,Is2} の実行
(入力ストリームを増やす)



(3) Is2 = [data|NIs2] の実行
(マージャへデータを流す)



(4) Is2 = [] の実行 (入力ストリームの終了)



(5) Is1 = [] の実行 (マージャの終了)

図 4.6.3 マージ処理

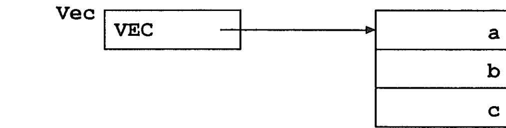
```
Vec = {a,b,c,d,e}, Pos = 3,
NewElem = x
```

として呼び出すと

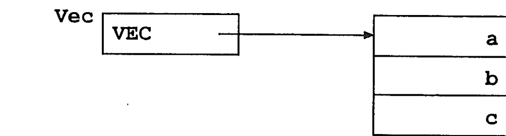
```
Elem = d, NewVec = {a,b,c,x,e}
```

となる (Vec の内容はそのままである).

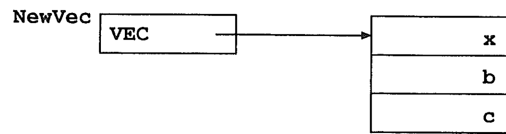
この組込み述語を正直に実装すると，Vec と同じ大きさの領域を新しい構造体 NewVec のために割りつけ，Pos 番目を除くすべての要素をコピーすることになるが，これには以下の問題がある.



(1) ベクタ {a,b,c} は多重参照/単一参照である



(2a) Vec が**多重参照**の場合
set_vector_element(Vec,0,_,x,NewVec) を実行すると，ベクタの領域が割りつけられ，Vec の要素がコピーされて NewVec が作られる



(2b) Vec が**単一参照**の場合
set_vector_element(Vec,0,_,x,NewVec) を実行すると破壊的書換えにより NewVec が作られる

図 4.6.4 set_vector_element 組込み述語の実行

メモリの大量消費： 構造体を割りつけるだけのメモリを更新のたびに消費してしまう．たとえば，1K ワードのベクタを 1K 回更新すると 1M ワードのメモリが消費される.

一定コストでの更新が不可能： 構造体を新しく作るコストは構造体の大きさに比例する．このため構造体が大きいと更新の手間もそれに比例して大きくなる.

set_vector_element/5 のほかに Vec を参照するゴールが存在する場合は Vec をそのまま保存する必要があるので，上述の処理を行なわなければならない.

たとえば

```
Vec = {a,b,c},
q(Vec),
set_vector_element(Vec,0,_,x,NewVec),
```

では q/1 が Vec を参照するため，Vec を保存する必要がある．このため NewVec は新しく作らなければならない (図 4.6.4 (2a))．しかし，そのようなゴールが存在しないのであれば，Vec を保存する必要がないので，要素を破壊的に書き換えても問題はない．たとえば

```
Vec = {a,b,c},
set_vector_element(Vec,0,_,x,NewVec),
```

では，Vec を参照するのは set_vector_element/5 だけなので，Vec を保存する必要がない．このため，Vec の要素を破壊的に書き換えて NewVec を作ることができる (図 4.6.4 (2b)).

B. 単一参照と多重参照

アトム，リスト，ベクタなどの具体値 (の書き込まれたデータセル) が一つのゴール (組込み述語も含む) からのみ参照されている状態を**単一参照**と呼び，二つ以上のゴールから参照されている状態を**多重参照**と呼ぶ．変数の場合は二つのゴールから参照されている状態が**単一参照**であり，三つ以上のゴールから参照されると**多重参照**となる．これは，変数に対しては「値を書き込むために使われる参照」と「読み出すために使われる参照」の両方が必要だからである．書き込むための参照しかないのでは書いても読まれることがないし，読み出すための参照しかないのでは書き込まれることのない値を永久に待ってしまう．このため変数については，二つの参照の存在することが単一参照となる．単一参照と多重参照の例を以下に示す.

単一参照

1) p :- true | q(X), r(X).
変数 X は二つのゴールだけから参照されるので単一参照である.

2) q(X) :- true | r(X).
X への参照は q/1 のなかで増加しないので，(q/1 が呼ばれた段階で単一参照ならば) X は単一参照 (のまま) である．たとえば，q/1 が以下のように呼び出された場合

```
p :- true | q(A), r(A).
p :- true | q({a,b}).
p :- true | q([x,y,z]).
```

変数 A や，(X に結合された) ベクタ {a,b} およびリスト [x,y,z] は単一参照である.

多重参照

1) p :- true | q(X), r(X), s(X).
変数 X は三つのゴールから参照されるので多重参照である.

2) q(X) :- true | q1(X), q2(X).
X への参照は q/1 のなかで増加するので，q/1 が呼ばれた段階では単一参照であっても，q/1 の実行中に X は多重参照になる．たとえば，q/1 が以下のように呼び出された場合

```
p :- true | q(A), r(A).
p :- true | q({a,b}).
p :- true | q([x,y,z]).
```

変数 A や，(X に結合された) ベクタ {a,b} やリスト [x,y,z] は q/1 の実行によって多重参照になる.

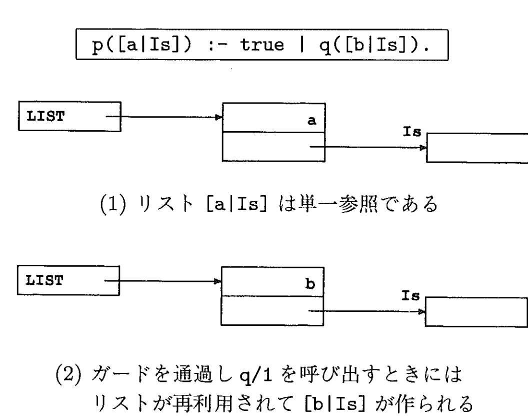
3) p :- true | X = {a,b,c,d}, q(X), r(X).
(X に結合された) ベクタ {a,b,c,d} には複数の参照があるので多重参照である.

単一参照の具体値に対しては，以下のような効率的な処理が可能である.

破壊的代入： set_vector_element/5 の実行において破壊的な書換えが可能になる.

即時の回収： 参照が消滅したことを即時に検出できるので即時の回収が可能になる．たとえば

```
p(X) :- wait(X) | true.
```



(1) リスト [a|Is] は単一参照である

(2) ガードを通過し q/1 を呼び出すときには
リストが再利用されて [b|Is] が作られる

図 4.6.5 単一参照のためリストの再利用が行なわれる例

では，X に結合された具体値が単一参照ならばガード通過後にその具体値を回収することができる．

即時の再利用： 参照消滅の即時検出ができるので即時の再利用も可能になる．たとえば

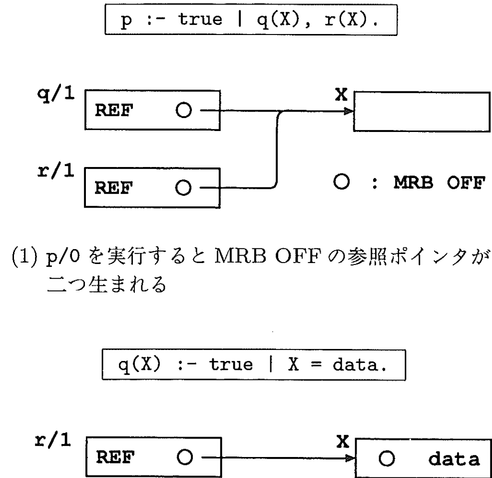
```
p({A,_}) :- atom(A) | q({c,d}).
p([a|Is]) :- true | q([b|Is]).
```

では，{A,_} や [a|Is] が単一参照ならばこれを再利用して {c,d} や [b|Is] を作ることができる．ベクタやリストを新しく割りつける必要はない．リストが再利用される例を図4.6.5に示す．

C. MRB の操作

KL1 で記述されたプログラムの実行では単一参照データが多い．一方，上述したようにデータの単一参照を検出できるメリットは大きい．多重参照ビット (Multiple Reference Bit，以下 MRB と略す) はこれを可能にするものである [15, 23, 24, 25, 26, 27]．

MRB は参照ポインタの性質を示すものであり，参照ポインタと値につく．ポインタの MRB はそのポインタの単一参照/多重参照を示すもので，単一参照ならば OFF，多重参照では ON となる．値の MRB は，その値が書き込まれたときに経由した参照ポインタの性質を示すものであり，OFF は「単一参照ポインタを経由して書き込まれた」ことを，ON は「多重参照ポインタを経由して書き込まれた」ことを示す．



(1) p/0 を実行すると MRB OFF の参照ポインタが二つ生まれる

(2) q/1 を実行すると値が MRB OFF で書き込まれる

図 4.6.6 MRB の操作 — 変数の生成と値の書込み (MRB OFF)

以下に MRB の操作を示す．

変数の生成時： 変数の参照が二つならば変数への参照ポインタの MRB を OFF(単一参照) に設定する (図 4.6.6 (1))．参照が三つ以上ならば ON(多重参照) に設定する (図 4.6.7 (1))．

値の生成時： 値の MRB を OFF (単一参照) に設定する．

参照数が増えるとき： 参照ポインタの MRB を ON にする．ポインタの指すデータセルに対しては何も操作を行なわない (図 4.6.8)．

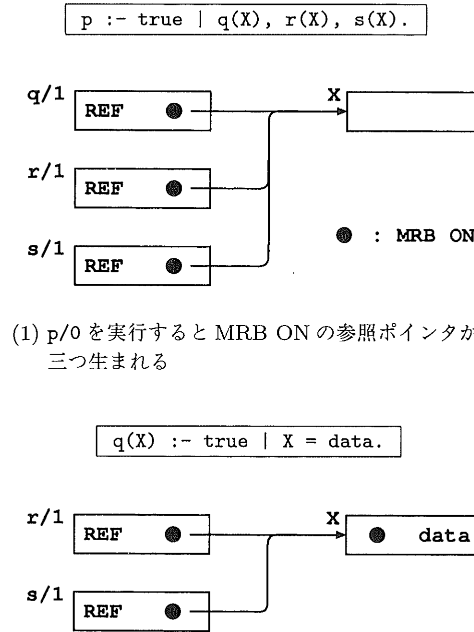
OFF の参照ポインタからの書込み： 値の MRB を OFF にして書き込む (図 4.6.6 (2))．

ON の参照ポインタからの書込み： 値の MRB を ON にして書き込む (図 4.6.7 (2))．

参照数が減少したときの MRB の操作はない．したがって，いったん ON になった MRB が再び OFF に戻ることはない．

D. MRB による単一参照の検出

MRB OFF のポインタから読んだ具体値の MRB が OFF ならば，その具体値は単一参照である．このように，単一参照であると認められるには，ポインタだけではなく値の MRB も OFF でなくてはならない．この理由を以下に述べる．



(1) p/0 を実行すると MRB ON の参照ポインタが三つ生まれる

(2) q/1 を実行すると値が MRB ON で書き込まれる

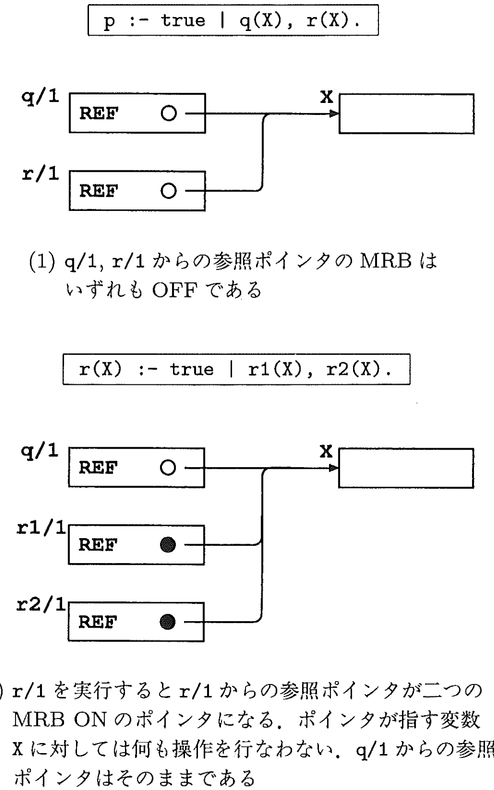
図 4.6.7 MRB の操作 — 変数の生成と値の書込み (MRB ON)

2 本の MRB OFF の参照ポインタとともに変数が生成されたあとに，一方のポインタを経由して参照数が増えて多重参照になることがある．このとき，もう一方のポインタは MRB OFF のままである (図 4.6.9 (1))．

ここで MRB ON のポインタを経由して値を書き，MRB OFF のポインタを経由して値を読む場合は，ポインタの MRB だけでは多重参照であることがわからない (図 4.6.9 (2))．具体値も MRB をもち，ポインタの MRB と合わせて調べることによって多重参照であることが検出できる．

E. 参照カウント方式との比較

MRB は，ポインタの単一参照/多重参照を示すビットをポインタとデータ本体にもたせるものであった．一方，参照カウント方式は，データを参照するポインタの数をカウントするカウンタをデータ側にもたせ，参照ポインタの増減時にカウンタを操作するものである．このため，値の読み書きを行なわなくても参照数が増減しただけでデータ本体側 (にあるカウンタ) を操作しなくてはならない．たとえば，次のような値の読み書きを行なわない述語を実行するときにも，データ本体をアクセスしなければならない．



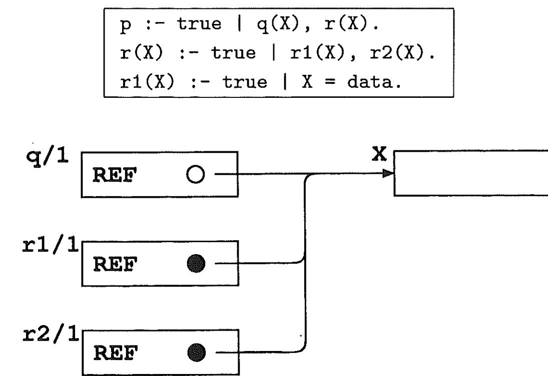
(1) q/1, r/1 からの参照ポインタの MRB はいずれも OFF である

(2) r/1 を実行すると r/1 からの参照ポインタが二つの MRB ON のポインタになる．ポインタが指す変数 X に対しては何も操作を行なわない．q/1 からの参照ポインタはそのままである

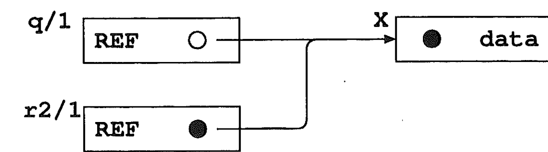
図 4.6.8 MRB の操作 — 参照数の増加

```
p(X) :- true | p1(X), p2(X).
p(_) :- true | true.
```

これに対して MRB では，データ本体を操作するのは値を書込む/読出すときだけであり，MRB の操作のためだけにデータ本体をアクセスすることはない．



(1) p/0 および r/1 の実行により X は多重参照となったが
q/1 からのポインタは MRB OFF のままである



(2) r1/1 を実行すると値が MRB ON で書き込まれる.
q/1 からのポインタは MRB OFF のままである

図 4.6.9　値の MRB の役割

4.7　ノード間処理の実現

4.7.1　外部参照の管理

A.　ノードごとに独立した GC の必要性

知識情報処理では動的かつ不均質な処理を行なうため, ノードによりメモリの消費速度が異なる. ほとんどメモリを消費しないノードがある一方, あるノードはほんの数秒でメモリを使い切ってしまうかもしれない.

このため, 各ノードが独立して GC を行なえることが必要不可欠である. ノードがすべて同期して行なう GC (大域 GC) しかできないのでは, メモリ消費速度の最も速いノードがメモリを使い切るたびに大域 GC が起動されてしまう. 最も悪いノードにみんなが合わせることになってしまうわけである.

B.　独立した GC を行なうには — 輸出表の必要性

他のノード内のデータセルを指すポインタを**外部参照ポインタ**と呼ぶ (図 4.7.1). この外部参照ポインタをノード番号とノード内アドレスの対で表わすことは可能だが, それではノードごとに独立した GC を行なうことはできない. GC を行なうとそのノード内のデータセルのアドレスが変化するので, そのノードを指している外部参照ポインタをすべて更新しなければならなくなる. これを行なうには, GC を行なったノードは自ノード内のすべてのデータセルのアドレス変化状況を他のすべてのノードへ伝えなければならない. それには莫大なデータ通信が必要であり, 事実上できない.

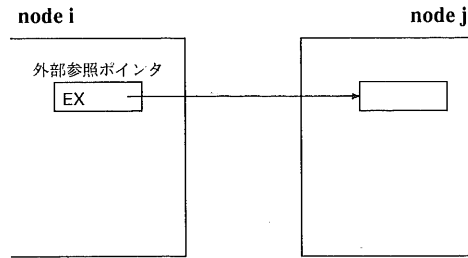


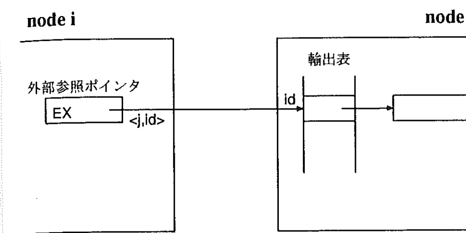
図 4.7.1　外部参照ポインタ

PIM では**輸出表**という名前のアドレス変換テーブルを各ノードがもち, 他のノードのデータセルはこの輸出表を経由して指す方式をとっている (図 4.7.2 (a)). 外部参照ポインタはノード番号と輸出表のオフセットの対からなる.

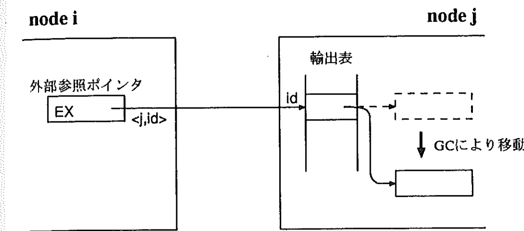
あるノードが GC を行なうとそのノードの輸出表を更新する. 輸出表のオフセットは変化しないので, 他のノードは GC 前と同じ外部参照ポインタでデータセルを参照できる (図 4.7.2 (b)). このため GC を行なったノードは他のノードに対して何も行なう必要はなく, 各ノードは独立に GC を行なうことができる.

C.　輸出処理と輸入処理

引数が変数であるゴールを他のノードへ投げ出すとき, 外部参照ポインタが生まれる. 送信元ノードは変数を輸出表に登録し, 変数のアドレスをノード番号と輸出表のオフセット値の対に変換して送る. これを受信したノードは外部参照ポインタを生成する. アドレスを変換して送ることを**輸出処理**と呼び, 変換されたアドレスを受信して外部参照ポインタを生成することを**輸入処理**と呼ぶ (図 4.7.3).



(a) 他のノードのデータセルは
輸出表を経由して指す



(b) GC によってデータセルのアドレスが変化
しても他のノードには影響しない

図 4.7.2　輸出表

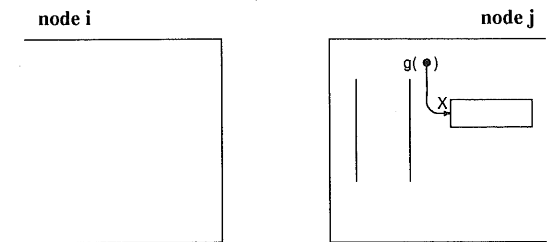
同じ変数を含むゴールを何度も投げることがある. このとき同じ変数を何度も輸出することになるが, 輸出のたびに新しく輸出表へ登録し, 毎回異なったオフセット値を得る方式には以下の問題がある.

- 輸出の数だけ輸出表エントリが必要になる.
- 同じデータが輸出されても輸入したノードからは異なったものに見えてしまうので, 輸入の数だけ読み書き処理を行なってしまう.

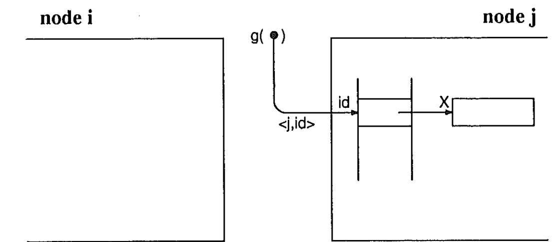
このため, PIM では以下に示す方式を採用した.

- 輸入側ノードでは外部参照セルが**輸入表**を経由して輸出側ノードを指す (図 4.7.4).
- 単一参照パス[†1]は高々二つしか存在しないので, パスが単一参照の場合は輸出のたびに新しく輸出表へ登録してアドレス変換を行なう. 輸入側でも輸入のたびに輸入表エントリを割りつける (図 4.7.5).
- 多重参照パス[†1]はいくつでも存在するので, パスが多重参照の場合は, 同じデータは同じ輸出表エントリに登録する. 輸入側では同じ外部参照ポインタは一つにまとめる (図 4.7.6).

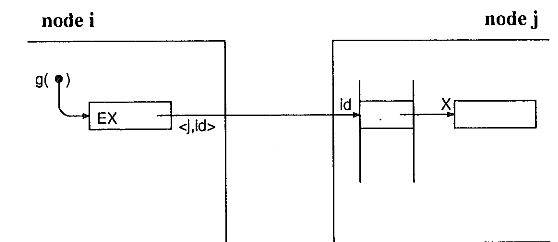
†1　単一参照/多重参照については 4.6.2 項 多重参照ビット (MRB) を参照のこと.



(1) ゴール g は変数 X を引数としてもつ



(2) 変数 X を輸出表に登録しゴールを投げ出した



(3) ゴールを受け取り外部参照ポインタを生成した

図 4.7.3　輸出処理と輸入処理

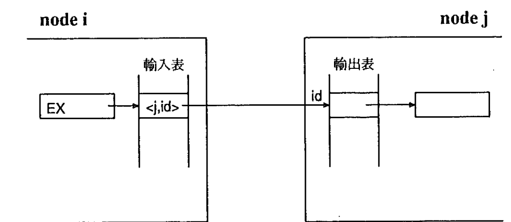
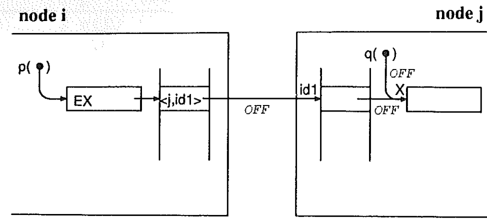
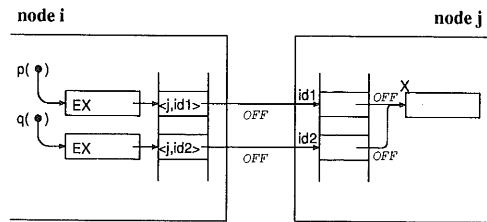


図 4.7.4　外部参照セルは輸入表を経由して
輸出側ノードを指す

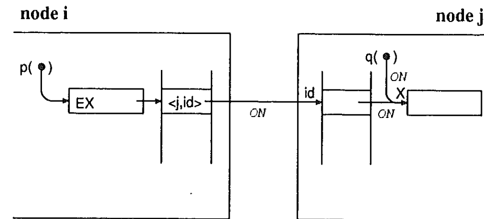


(1) ゴール p と q は同じ変数 X を引数としてもつ．
X への参照は単一参照 (OFF) である

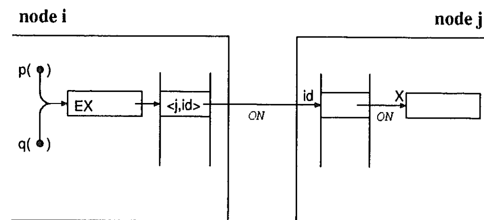


(2) q をノード i へ投げ出すと p とは異なった
輸入表エントリを経由して X の本体を指す

図 4.7.5 単一参照の場合は輸出のたびに新しく
輸出表へ登録する



(1) ゴール p と q は同じ変数 X を引数としてもつ．
X への参照は多重参照 (ON) である



(2) q をノード i へ投げ出すと p と同じ輸入表
エントリを経由して X の本体を指す

図 4.7.6 多重参照の場合は同じ外部参照ポインタは
一つにまとめる

D.　輸入表の必要性

輸出表の GC を行なうには，外部参照ポインタを解放したとき，そのポインタが指しているノードへ解放したことを伝えることが必要である．

PIM では，ノード内の独立した一括 GC としてコピー方式 GC を採用している．KL1 の実行ではアクティブなデータセルはわずかなことが多いので，処理時間がアクティブデータの数に比例するコピー方式 GC が有利だからである．このコピー方式 GC では，アクティブな外部参照ポインタのみが検出され，ゴミである外部参照ポインタはただ消滅してしまうだけである．したがって，どの外部参照ポインタが解放されたか明示的にはわからず，ポインタが指すノードへ解放を伝えることができない．

輸入表を設けることで，コピー方式 GC によって解放された外部参照ポインタを検出することができる．輸入表にはすべての外部参照ポインタが登録されている．GC で，アクティブであることが検出された外部参照ポインタが指す輸入表エントリには，マークをつけていく．GC 終了後に輸入表をたどると，マークのついていないエントリの示す外部参照ポインタが解放されたポインタであることがわかる (図 4.7.7)．

E.　輸出表の GC

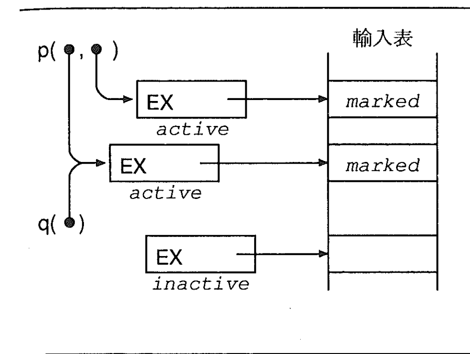
それを指すすべての外部参照ポインタが解放されたならば，輸出表エントリを解放することができる．

単一参照データの輸出表エントリと外部参照ポインタは一対一に対応するので，外部参照ポインタの解放の連絡を受けたら対応する輸出表エントリを解放することができる (図 4.7.8)．これに対して，多重参照データの輸出表エントリは一般に複数の外部参照ポインタから参照されているので，何らかの工夫が必要になる．

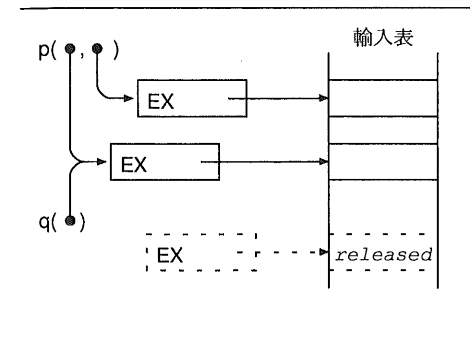
F.　単純な参照カウント方式 GC では不十分

参照カウント方式 GC は，参照するポインタの数を記憶する参照カウントを指されている側がもち，ポインタの増減に従って参照カウントを増減させるものである．参照するポインタのなくなったことは，参照カウントがゼロになることで検出される．

しかし，多重参照データの輸出表の GC にこの参照

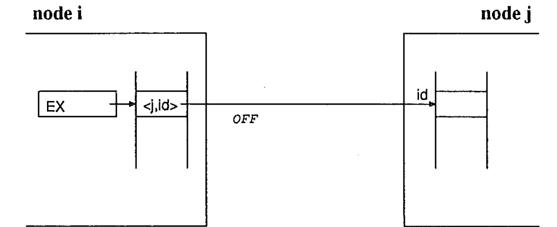


(1) アクティブな外部参照ポインタが指す
輸入表エントリにはマークをつける

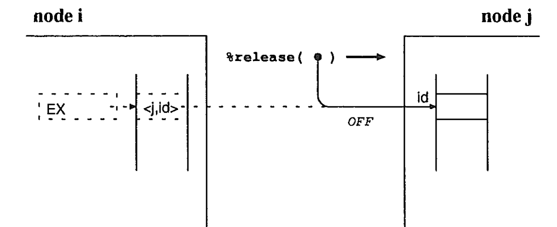


(2) マークのついていないエントリは
解放された外部参照ポインタを示す

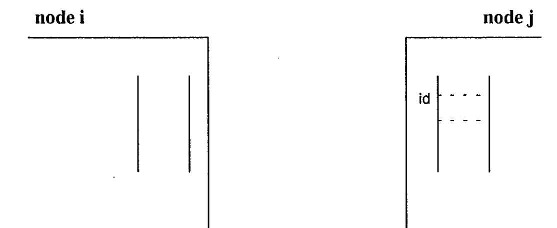
図 4.7.7 ノード内一括 GC における，解放された
外部参照ポインタの検出



(1) ノード i は単一参照である外部参照
ポインタをもっている



(2) 外部参照ポインタを解放し
%release メッセージを送信した



(3) ノード j は %release を受信して
輸出表エントリを解放した

図 4.7.8 単一参照である輸出表エントリの解放

カウント方式 GC を単純に適用しようとしてもうまく働かないことがある．以下にその例を示す．

1) ノード i の輸出表エントリをノード j の外部参照ポインタが指している．参照カウントの値は 1 である．

   この状態でノード j がこの外部参照ポインタをノード k へ輸出した．輸出することによって輸出表エントリを指すポインタが増えるので，ノード j は参照カウントを +1 するメッセージ (%increment) をノード i へ送信した．このメッセージがいつノード i に到着するかの保証はない (図 4.7.9 (1))．

2) ノード k は外部参照ポインタを受信し，外部参照セルを生成した (図 4.7.9 (2))．

3) ノード k は外部参照ポインタを解放した．輸出表エントリを指すポインタが減るので，ノード k は参照カウントを −1 するメッセージ (%decrement) をノード i へ送信した (図 4.7.9 (3))．

4) %decrement が %increment より先にノード i へ到着すると参照カウントがゼロになるため間違って輸出表エントリが解放されてしまう (図 4.7.9 (4))．

輸入側でポインタを増やすと輸出側の参照カウントを +1 する必要がある．しかし，これを要求するメッセージがいつ到着するか保証がないため，間違った解放の起こる可能性がある．

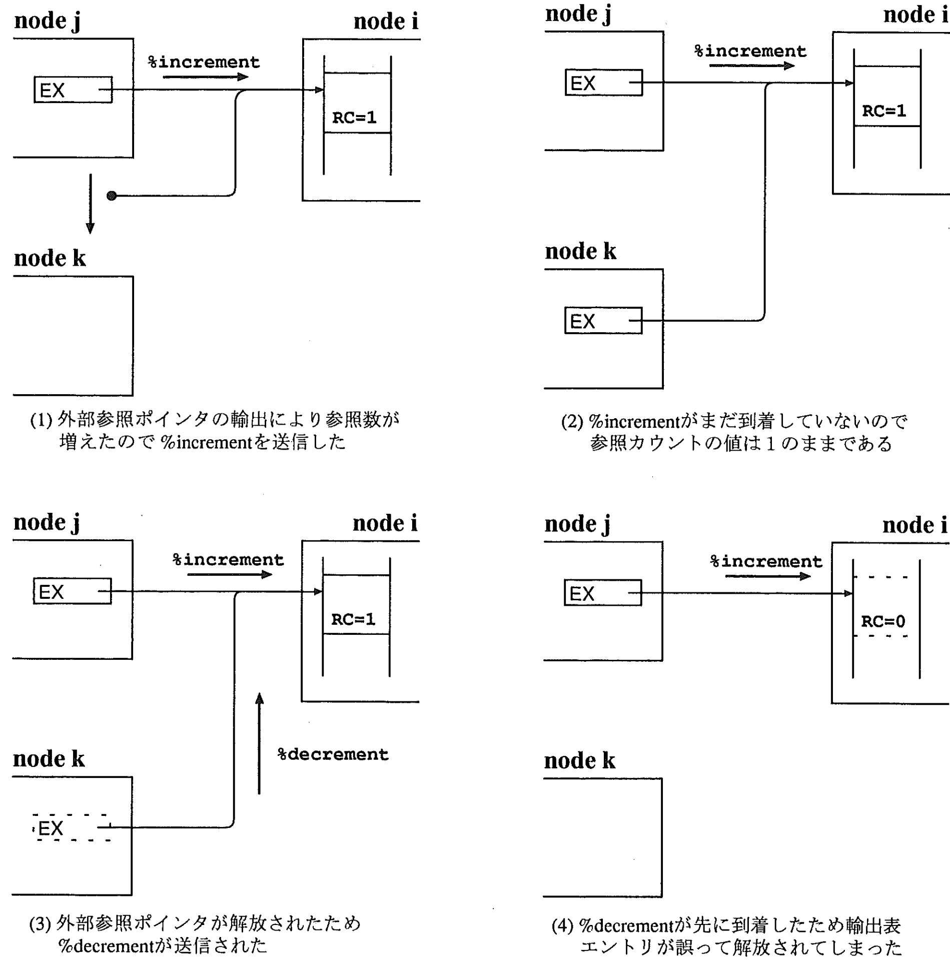


図 4.7.9 参照カウント方式 GC ではうまく行かない例

G. WEC 方式 GC

PIM では，輸出表エントリだけでなく輸入表エントリとメッセージ中の外部参照ポインタにも参照カウントをもたす“重みづけ輸出カウント (Weighted Export Counting) 方式” [28, 29, 30] を採用した．この方式では，輸出表エントリのカウンタを操作することなく輸入側でポインタを増やすことができる．

各参照カウントは参照の数ではなく重みづけした値をもつ．この重みづけした値を “Weighted Export Count (WEC)” と呼ぶ．そして「ある輸出表エントリの WEC」と「その輸出表エントリを指す輸入表エントリと外部参照ポインタの WEC の値の合計」が等しくなるように制御する[†1]．この制御により，ある輸出表エントリへの「参照ポインタが存在しない」ことと，その輸出表エントリの「WEC の値がゼロになること」は同値になる．

輸出を行なう場合，外部参照ポインタに適当な値をつけて送る．輸出表エントリの WEC は，つけた値の分だけ増やす．外部参照ポインタを受け取ると，ついていた値を輸入表エントリの WEC に加える (図 4.7.10 (1),(2))．

†1 通常の参照カウント方式では「ある輸出表エントリの参照カウント」と「その輸出表エントリへの参照ポインタの数」が等しくなるように制御する．

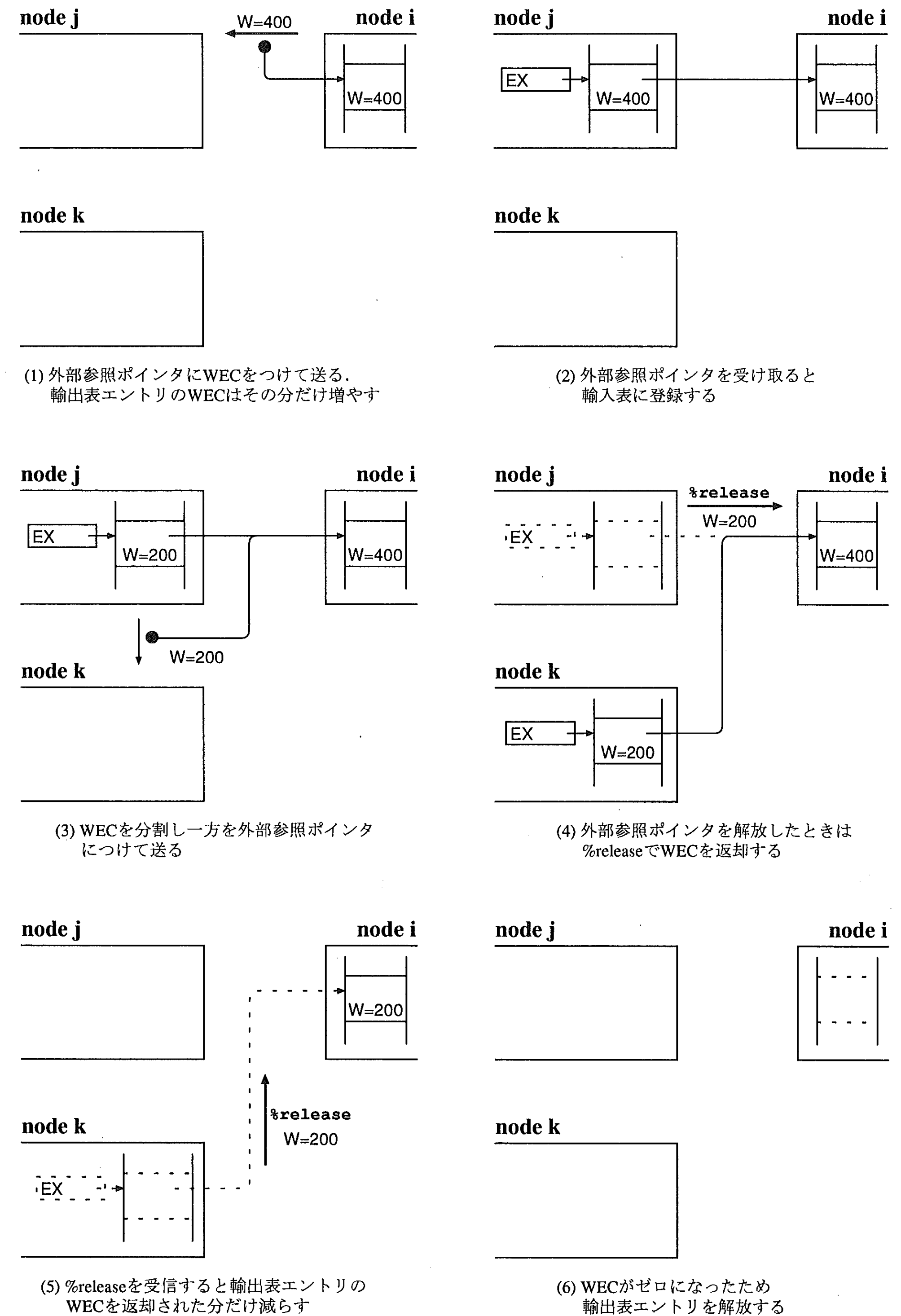


図 4.7.10 WEC 方式による輸出入処理

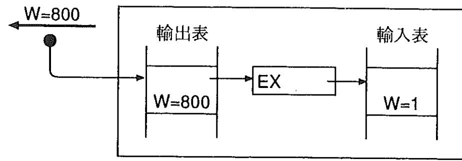
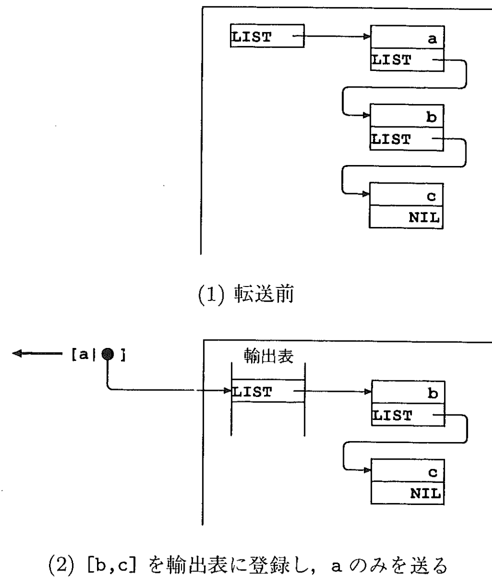


図 4.7.11 間接輸出



(1) 転送前

(2) [b,c] を輸出表に登録し，a のみを送る

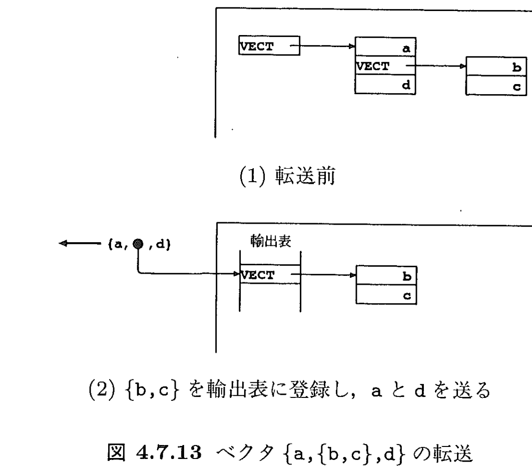
図 4.7.12 リスト ([a,b,c]) の転送

輸入側で外部参照ポインタを輸出する場合は，輸入表エントリの WEC を適当な二つの値に分割し，一方を外部参照にポインタにつけ，もう一方をエントリの新しい WEC とする (図 4.7.10 (3)).

外部参照ポインタを解放したときはメッセージ (%release) を輸出側へ送り，WEC の値を返却する (図 4.7.10 (4)).

%release を受信すると，対応する輸出表エントリの WEC から返却された値を引く (図 4.7.10 (5)). 引いた結果ゼロになったならば，そのエントリを参照する外部参照ポインタの存在しないことが保証されるので，輸出表エントリを解放する (図 4.7.10 (6)).

H. 間接輸出



(1) 転送前

(2) {b,c} を輸出表に登録し，a と d を送る

図 4.7.13 ベクタ {a,{b,c},d} の転送

輸入表エントリの WEC の値が 1 になると WEC を分割することができないので，そのままでは外部参照ポインタを輸出することができない．これに対処するため，PIM では間接輸出を採用した[†1]．これは外部参照ポインタを輸出表に登録し，これを指す外部参照ポインタを生成して輸出するものである (図 4.7.11).

4.7.2 ノード間にわたるユニフィケーション

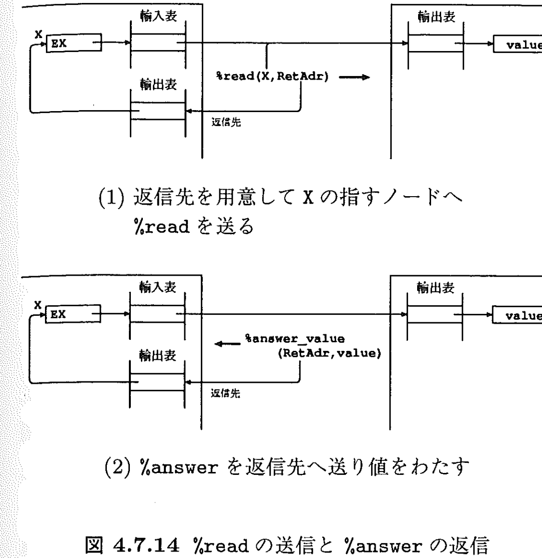
A. データ転送の方針

PIM では「必要となってはじめてデータ転送を行なう」という遅延転送方式を採用している．これは「データの転送は，それが本当に必要になるまで遅らせたい．そのデータを使わずに済むかもしれないので」という考えに基づいている．遅延転送方式は，積極的にデータ転送を行なう方式と比較して処理遅延の生じる可能性があるが，そのデメリットよりも不必要なデータ転送を回避できるメリットが勝ると考えた．

たとえば，ユニフィケーションで外部参照ポインタに出会うとノード間にわたるデータ転送が起きるが，データが構造体の場合はデータの全体は転送せず，トップレベルのみを送る．以下にリストとベクタの例を示す．

- リスト [a,b,c,d] を転送する場合，[b,c,d] を輸出表に登録し，[a|X] を送る．ここで X は [b,c,d] を指す外部参照ポインタである (図 4.7.12).
- ベクタ {a,{b,c},d} を転送する場合は {b,c} を輸出表に登録し，{a,Y,d} を転送する．ここで

†1 間接輸出を行なわず，輸出表から WEC の補給を受ける方式も考えられる．



(1) 返信先を用意して X の指すノードへ %read を送る

(2) %answer を返信先へ送り値をわたす

図 4.7.14 %read の送信と %answer の返信

Y は {b,c} を指す外部参照ポインタである (図 4.7.13).

B. ガードユニフィケーション

```
p(X) :- X = a | true.
```

というクローズ p/1 が定義されている状況で，ゴール p(X) を実行することを考える．ここで X が外部参照ポインタであると以下のノード間データ転送が起こる．

1) 読み出し要求メッセージ

%read(X,RetAdr)

が X の指すノードへ送信される．RetAdr は，返信先として新しく登録された外部参照アドレスである (図 4.7.14 (1)).

2a) %read を受信したとき，X が示すデータセルが値 value に具体化されていたならば

%answer_value(RetAdr,value)

を返信先へ送信して値をわたす (図 4.7.14 (2)). value がアトミックデータならば value 全体が送信されるが，リストやベクタの場合は前述したように，トップレベルのみが送信される．

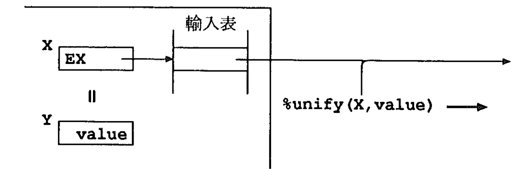


図 4.7.15 具体値と外部参照ポインタのユニフィケーション

2b) X の示すデータセルが具体化されていない場合は，具体化されるまで %answer_value の送信は行なわない．「具体化されていない」ことの伝達は行なわない．

2c) X が外部参照ポインタ Y であった場合は，%read を転送する．すなわち，Y が示すノードへ

%read(Y,RetAdr)

を送る．返信先 (RetAdr) はそのままであるため，%answer_value は最初に %read を送信したノードへ直接返信される．

C. ボディユニフィケーション

```
p(X,Y) :- true | X = Y.
```

というクローズ p/2 が定義されている状況で，ゴール p(X,Y) を実行することを考える．X,Y のいずれか一方が外部参照ポインタであると，以下に示すノード間ユニフィケーションが行なわれる．

a. 具体値と外部参照ポインタのユニフィケーション

X が外部参照ポインタで Y が具体値 value であった場合，%unify(X,value) が X の指すノードへ送信される (図 4.7.15). %answer と同様に，value がアトミックデータならば value 全体が送信されるが，リストやベクタの場合はトップレベルのみが送信される．

b. 外部参照ポインタ同士のユニフィケーション

いずれか一方の外部参照ポインタが指すノードへ %unify を送信し，送信先ノードで再びユニフィケーションを行なう (図 4.7.16). 外部参照ポインタがさらに外部参照ポインタを指している場合は，再び外部参照ポインタ同士のユニフィケーションとなるが，1 回の %unify 送信で外部参照ポインタが必ず 1 段手繰ら

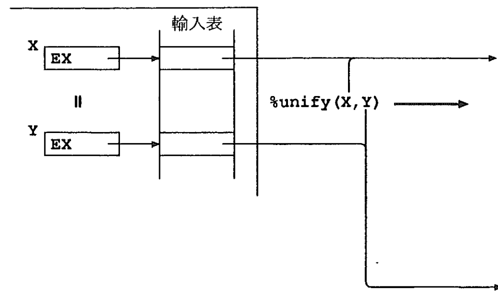
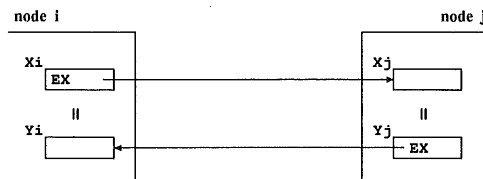
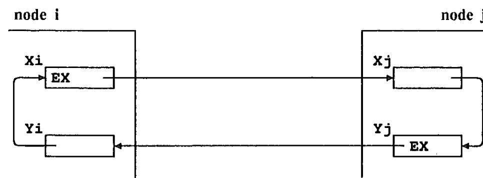


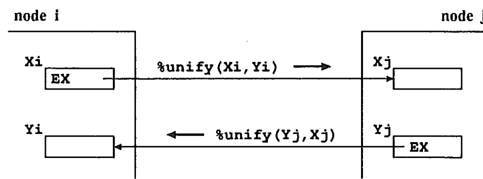
図 **4.7.16** 外部参照ポインタ同士のユニフィケーション



(1) ユニフィケーション実行前



(2a) 単純にパスを張るとループが生じる



(2b) %unify を送信すると永久に終わらない

図 **4.7.17** 変数と外部参照ポインタのユニフィケーションの問題点

れるため, 有限回の %unify 送信で前述した「具体値と外部参照ポインタ」のユニフィケーション, あるいは後述する「変数と外部参照ポインタ」のユニフィケーションに落ち着くことが保証される.

c. 変数と外部参照ポインタのユニフィケーション

一方が外部参照ポインタ, 他方が変数の場合は, 処理は単純ではない. 変数同士のユニフィケーションでは一方から他方へ参照パスを張るが, これと同様にノード内で変数から外部参照へ単純に参照パスを張る

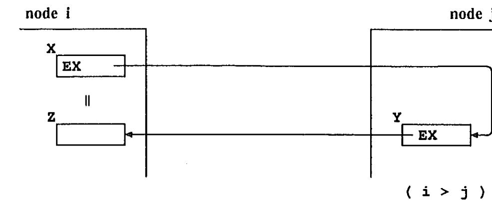


図 **4.7.18** ノード番号の大小比較だけでは不十分である

とループが生まれてしまう可能性がある. たとえば, 図 4.7.17 (1) のノード i, j のそれぞれで参照パスを張ると, 図 4.7.17 (2a) に示すようにループが生じてしまう.

一方ループの生成をおそれて, 外部参照ポインタと具体値の場合と同様に %unify を送信したのではユニフィケーションが終わらなくなってしまう. 図 4.7.17 (1) ではノード i, j 間を %unify が永久に往復してしまう (図 4.7.17 (2b)).

間接輸出を行なわないのであれば, ノード番号の大小関係を利用してループの生成を防ぐことができる. たとえば

- 外部参照ポインタがノード番号の大きいほうから小さいほうを指すものならば, ノード内で参照パスを張る
- そうでないならば %unify を送信する

とすればよい. 図 4.7.17 (1) で $i > j$を仮定すると, ノード i では参照パスを張るが, ノード j は %unify を送信するためループは生じない.

しかし, PIM では間接輸出を行なうため, この方式はうまく働かない.「パスを張ってはいけない (ノード番号小→大) 外部参照ポインタ」が, 間接輸出により「パスを張ってよい (ノード番号小→大) 外部参照ポインタ」に見えてしまうことがあるからである.

図 4.7.18は, (ノード番号小→大である) 外部参照ポインタ Y が間接輸出された状況を示している. ここで外部参照ポインタ X と変数 Z のユニフィケーションを行なうと, X はノード番号大→小の外部参照ポインタに見えるのでノード内部でパスを張ってしまい, ループが生まれてしまう.

D. Safe/Unsafe 属性の導入

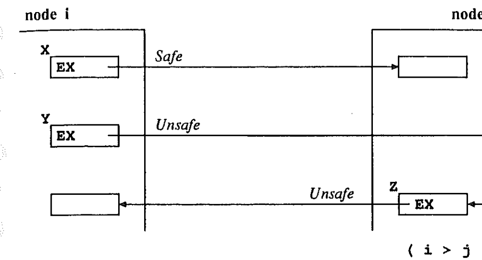
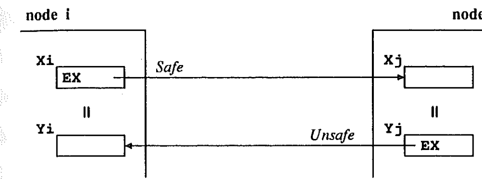
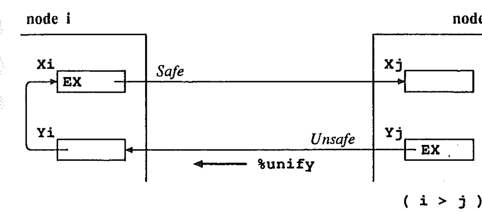


図 **4.7.19** Safe/Unsafe 属性



(1) Xi は Safe 外部参照ポインタ, Yj は Unsafe 外部参照ポインタである



(2) ノード j は %unify を送信し, ノード i ではパスを張る

図 **4.7.20** Safe/Unsafe 属性を用いたユニフィケーション規則

上述した問題に対処するため, PIM では以下に示す **Safe/Unsafe 属性**を導入した [29, 30, 31] (図 4.7.19 参照).

Unsafe 外部参照ポインタ : ノード番号の小さいほうから大きいほうを指す外部参照ポインタ, および Unsafe 外部参照ポインタを指す外部参照ポインタ

Safe 外部参照ポインタ : Unsafe 外部参照ポインタでない外部参照ポインタ

そして変数と外部参照ポインタのユニフィケーションは以下のルールに従う (図 4.7.20参照).

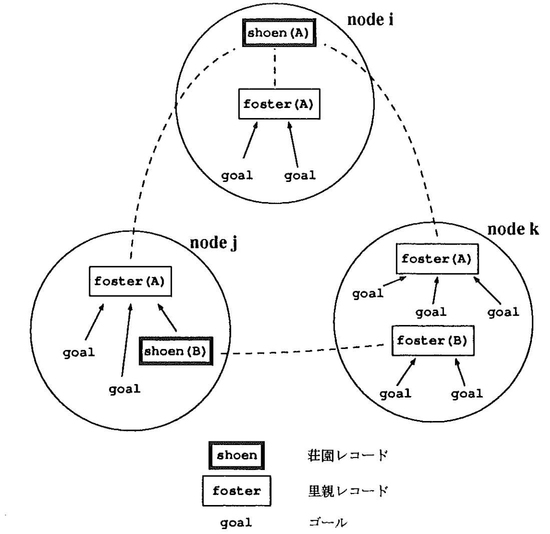


図 **4.7.21** 荘園の構成

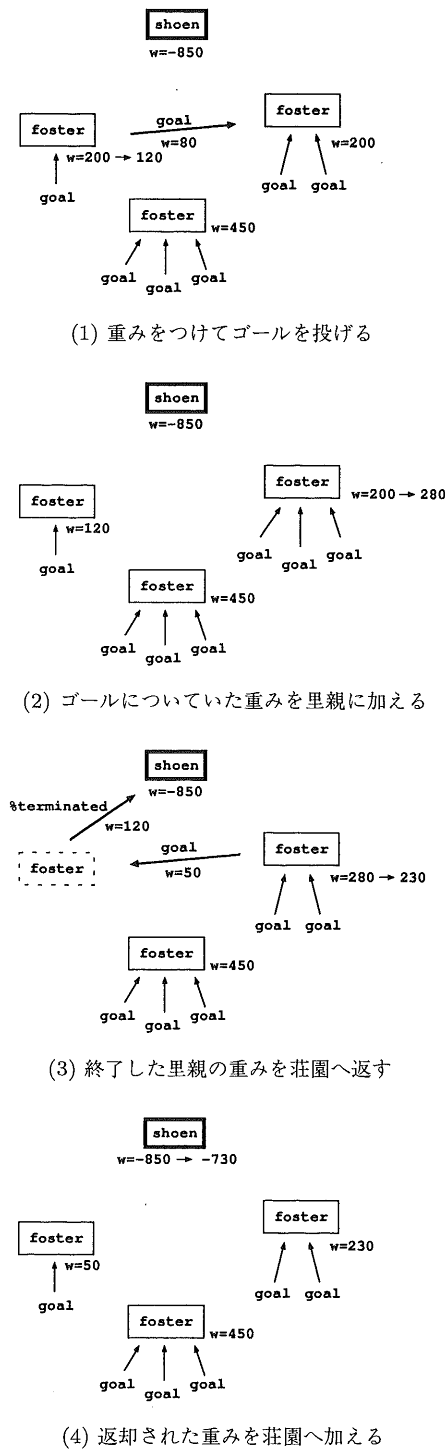
- Safe 外部参照ポインタならばノード内で参照パスを張る.
- Unsafe 外部参照ポインタならば %unify を送信する.

図 4.7.18の Y は「ノード番号の小さいほうから大きいほうを指す」ため Unsafe となり, X は「Unsafe 外部参照ポインタを指す」ため, やはり Unsafe となる. その結果, 参照パスは張られず %unify が送信され, ユニフィケーションが正しく行なわれる.

4.7.3 ノード間にわたるゴール管理

2.7.5項で述べたように, KL1 ではゴール群を管理する**荘園**という制御構造が存在する. ここでは, ゴール群がノード間にわたった場合の荘園の管理方式を述べる.

荘園は, 一つの荘園レコードと (一般に) 複数の里親レコード, その荘園に属する (一般に) 多数のゴール群からなる. 図 4.7.21は荘園の構成例である. 荘園 A のゴールはノード i, j, k に存在し, (荘園 A の子荘園である) 荘園 B のゴールはノード k にのみ存在する. 里親レコードは荘園機能を局所的に実現するものであり, 実行制御や資源, 状態などに関する情報を保持する. 対応する里親レコードの存在しないノードにゴールが到着すると里親レコードが作られる. また, それ



(1) 重みをつけてゴールを投げる

(2) ゴールについていた重みを里親に加える

(3) 終了した里親の重みを荘園へ返す

(4) 返却された重みを荘園へ加える

図 4.7.22 ゴールの送受信，里親の生成と終了

に属するゴールがすべて終了すると里親レコードは消滅する．この結果，ゴールの存在するノードには，必ず対応する里親レコードが存在する．また里親レコードが存在するならば，それに属するゴールが存在する．図 4.7.21 の例では，荘園 A の里親レコードはノード i, j, k に存在し，荘園 B の里親レコードはノード k にのみ存在する．

ゴールは里親レコードへのポインタをもち，里親レコードに記憶されている実行制御などの情報を適当なタイミングで調べ，「強制終了や実行停止の要求が出ていないか」「資源は尽きていないか」などを確認する．里親レコードを設けずゴールが直接荘園レコードを指す方式では，実行制御などの情報を調べるためにはノード間通信を行なって荘園レコードを参照する必要があり，荘園レコード参照がボトルネックになってしまう．里親レコードを設けることによって，このボトルネックを回避している．

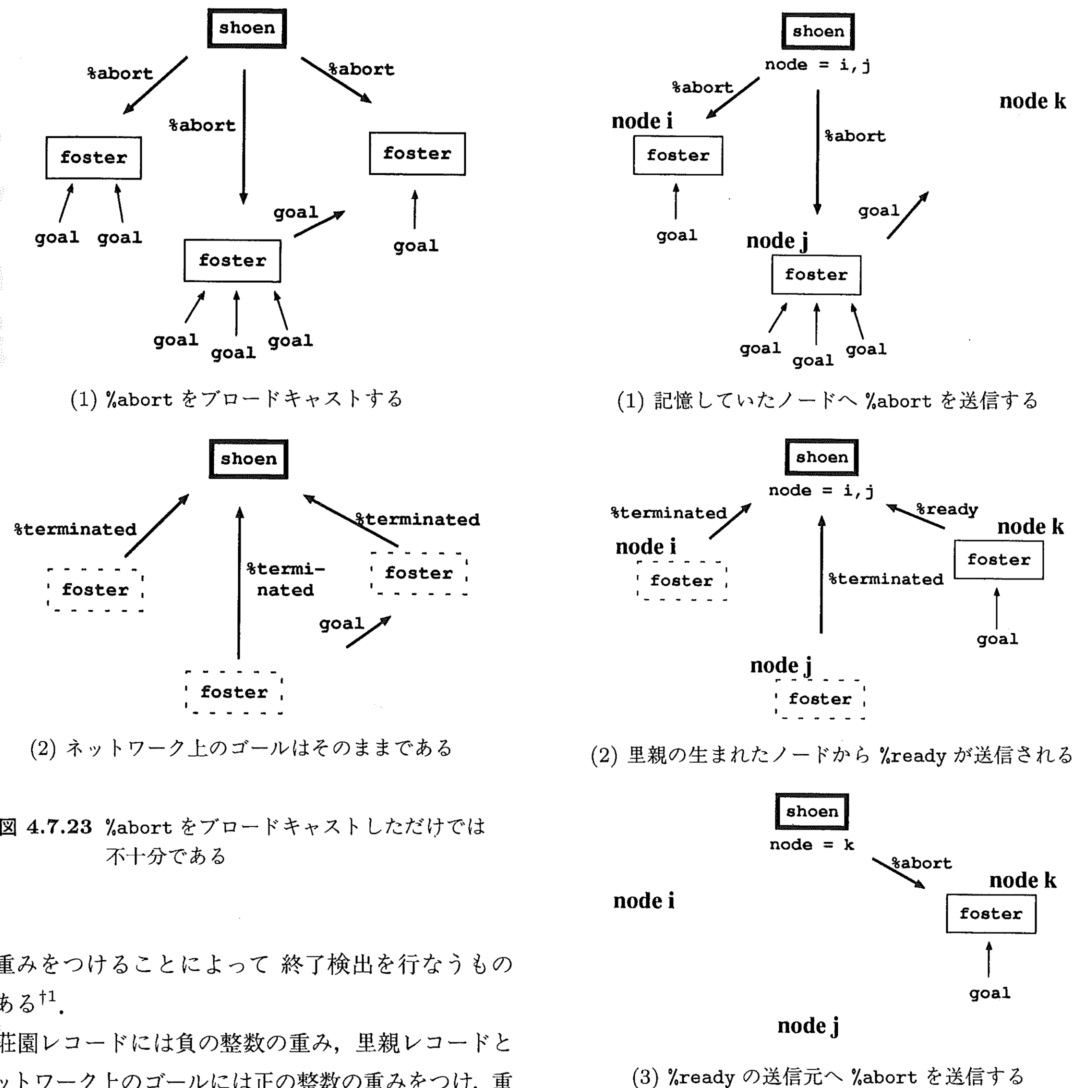
荘園がネストしている場合，子荘園の荘園レコードはゴールと同様に親荘園の里親レコードを指す．たとえば，図 4.7.21の荘園 B は荘園 A の子荘園であるので，ノード j 内の荘園 B の荘園レコードは荘園 A の里親レコードを指している．

A. 終了検出

ある荘園に属するゴール群の終了[†1]を検出することは荘園の状態監視機能の最も基本的なものであるが，この機能の実現は容易ではない．ある里親レコードに属するゴール群の終了は簡単に検出できるが，送信はされたがまだ受信されていないゴールがネットワーク上に存在するかもしれないからである．たとえば，すべての里親レコードが自分に属するゴールの終了をメッセージ (%terminated) によって荘園レコードへ報告しても，まだネットワーク上にゴールが残っている可能性がある．

PIM では，“重みづけ送出カウント (Weighted Throw Counting) 方式” [19, 32] によってゴール群の終了を検出している．この方式は，並列 GC の方式である “重みづけ参照カウント (Weighted Reference Counting) 方式” [33, 34] を終了検出に応用したものであり[†2]，荘園レコードと里親レコード，ネットワーク上のゴール

†1 その荘園の子荘園もゴールと同様に扱うことができる．
†2 GC 方式から終了検出方式の導出は [35] が述べている．



(1) %abort をブロードキャストする

(2) ネットワーク上のゴールはそのままである

図 4.7.23 %abort をブロードキャストしただけでは不十分である

(1) 記憶していたノードへ %abort を送信する

(2) 里親の生まれたノードから %ready が送信される

(3) %ready の送信元へ %abort を送信する

図 4.7.24 強制終了処理

に重みをつけることによって 終了検出を行なうものである[†1]．

荘園レコードには負の整数の重み，里親レコードとネットワーク上のゴールには正の整数の重みをつけ，重みの合計をゼロに保つように制御する．この結果，すべてのゴールが終了したときのみ荘園レコードの重みがゼロになる．ゴールを投げるときは，里親レコードから重みを適当な分だけ投げるゴールに移す (図 4.7.22 (1))．ゴールを受け取ると，ついていた重みを里親レコードの重みに加える (図 4.7.22 (2))．里親レコードに属するすべてのゴールが終了したら，%terminated を送信してもっていた重みを荘園レコードへすべて返却する (図 4.7.22 (3))．重みの返却を受けたら荘園レコードは自分の重みに加える (図 4.7.22 (4))．加えた結果ゼロになったならば，すべてのゴールの終了が検出される．

†1 表現形式は異なるが本質的には同じ方式を [36] が述べている．

B. 強制終了

荘園の実行制御の最も基本的なものの一つに，特定の荘園に属するゴール群の強制終了がある．「あるゴールが無限ループに陥った場合」や「ある荘園下のゴールの実行が無意味であるとわかったとき」など，ゴール群を強制的に終了させることができなくてはならない．

しかし，強制終了の実現は容易ではない．強制終了を起こすメッセージ (%abort) をすべてのノードへ送信しただけでは，ゴール群をすべて終了させることはできな

い．メッセージが到着したときに里親レコードの下にあったゴール群を終了させることはできるが，まだネットワーク上に存在したゴールを終了させることはできないからである (図 4.7.23).

終了は前述した重みづけ送出カウント方式によって正しく検出されるので，里親レコードの存在するノードへ確実に %abort が届きさえすればよい．PIM では，里親レコードの存在を荘園レコードへ伝達することによってこれを実現した．

里親レコードが存在しないノードへゴールが到着し里親レコードを生成した場合，生成を伝えるメッセージ (%ready) を荘園レコードへ送信する．荘園レコードは %ready を受信すると送信元ノードを記憶する．

強制終了は以下の手順で行なう．

1) 記憶しているノードへ %abort を送信する．
2) 強制終了中に %ready を受信したらその送信元へ %abort を送信する．

荘園レコードが把握していた里親レコード下のゴール群は 1) によって終了する．一方，強制終了が始まってから検出した里親レコード下のゴール群は 2) が対処する．図 4.7.24に強制終了の例を示す．

第 5 章 並列オペレーティングシステム PIMOS

5.1 概説

5.1.1 設計・開発の方針

PIMOS は並列推論マシン PIM のために開発されたオペレーティングシステム (OS) である．第五世代プロジェクトでは，第五世代コンピュータのプロトタイプの構築を目標においてきた．プロトタイプ・システムを作り上げる目的は，机上だけの理論，実際の使いものにはならないオモチャのソフトウェア，ソフトウェアがろくに走らないハードウェアの実験機，といったものの寄せ集めではなく，高並列のハードウェアから，その上で動作する実用レベルの複雑さと規模の問題に対する並列応用ソフトウェアにいたるまで，知識情報処理のためのコンピュータの基礎技術を，体系的に，しかも実践的に作り上げることにあった．

このような背景のなかで，PIMOS の研究開発には，二つのやや矛盾する目的が課せられていた．

1) 並列応用ソフトウェアの研究開発環境を提供する．
2) 高並列システムの OS のあるべき姿を研究する．

目的が 2) だけなら，研究開発の成果はプロジェクトの最後に得られればよい．失敗に終わる可能性がある野心的な方式を追求することも考えられる．また，方式がわかりきっているような機能を開発する必要はない．一方，1) を目的とするならば，できるだけ早期に応用ソフトウェア研究に使えるシステムを提供する必要があるし，失敗は許されない．また，応用の研究に必要な機能は，OS 自身の研究テーマになり得ないものでも，一通り提供する必要がある．

PIMOS の研究開発にあたっては，まず 1) の応用ソフトウェアへの環境の提供を第一に，2) は従と考え，できるだけ早期に必要最低限の機能を提供し，順次改良していく方針をとった．これはプロジェクト全体のための要請もあったが，実際に使われるシステムを作らなければ，提供した機能や性能が本当に適切かを判断できないだろうと考えたからでもある．

その結果 PIMOS の設計が保守的なものになってしまったかというと，決してそうはならなかった．また，そうはできなかった．応用ソフトウェア研究のプラットフォームとして十分役立つように，1,000 台規模の高並列システムを十分な効率で制御し，適切なソフトウェア開発環境を提供するためには，逐次処理システム用の OS を拡張したような従来のシステムと同じ方式を踏襲するわけにはいかなかったのである．

5.1.2 百万台を想定した設計

同じ並列システムでも，10 台程度の規模の並列システムと 1,000 台規模のシステムとではまったく事情が違う．たとえば，OS による管理のオーバヘッドが，全体の処理の 1%を占めていたとしよう．十台程度の規模なら，すべての OS 機能を 1 台のプロセッサに集中しても，そのプロセッサに 10% のオーバヘッドがかかるだけで済む．これが 1,000 台になると，1,000% のオーバヘッドがかかることになる．100% 以上の処理ができるはずはないので，常に 900 台のプロセッサは OS の処理の終了を待っており，動いているのは 100 台だけになってしまう．1,000 台程度の規模の並列システムでは，OS の管理オーバヘッドが 1% でも，少なくともその 90% を並列分散処理することが必須な

のである．

どうせ新たに設計するのなら，1,000台規模までしか拡張できない方針はとりたくない．そこで，100万台規模まで適用できる方式をと，あらゆるOS機能をできる限り並列分散処理する方針をとった．[†1]

徹底した並列分散管理となると，さまざまな新しい方式の導入が必要になってくる．あらゆる管理を分権的にし，中央に連絡せずにその場で判断しなければならない．システム中のタスクに一連番号をつけて一つの表で管理する，といった方式は許されない．大規模並列システムでは通信遅延が無視できないので，分散した管理プロセス間の通信にはかなりの遅延があるものとして，処理手順を設計しなければならない．たとえば，一つのメッセージを送ったあと，その返答を待たずに次の処理に入れるようにする必要がある．

PIMOSでは，タスクは木構造をしており，親子は互いを知っているが，孫の代までは知らないようになっている．システム全体の状況は誰も把握していないのだが，各所で適切に管理していれば全体として整合的に動作する．PIMの場合はガーベジコレクション（ゴミ集め）によるかなり長い（秒単位の）実行中断も起きるので，任意の遅延があってもよいように処理手順を設計した．

ソフトウェア開発環境についても，従来の逐次処理用の開発環境の延長ではとてもやっていけない．10台規模なら応用ソフトウェアもその程度の並列性でよく，プロセスごとにウィンドウを作ってトレースすることもできよう．1,000台規模のシステムで同じことをしても，まったく役に立たない．もっと抽象的な視点に立ったツールが必要である．プログラムの振舞いを把握するにも，統計的な扱いが不可欠になる．

5.1.3 開発の経緯

PIMOSの設計を始めた1986年秋頃には，まだPIMはもちろん，その先駆となるマルチPSIシステムもなかったし，KL1の仕様もはっきりしていなかった．PIMOSの設計とKL1の設計は表裏一体で進められ，KL1の仕様が固まってくると，まずPIMOS開発用のUnixマシンで走るKL1の擬似並列処理系PDSSの開発が始まった．これと並行してPIMOSの設計と，KL1の並列分散処理系の実装方式の検討が続いた．

'87年にはPDSSが動き始め，その上でPIMOSの開発が始まった．この間にもKL1の言語仕様は何度も手直しされ，PDSSはそのつど改訂された．PIMOSの中核部分は，ほとんどこのPDSS上で開発された．

64台のプロセッサをもつマルチPSIの上でKL1処理系がなんとか動き始めた'88年の夏に入った頃，PDSSからその上にPIMOSを移植した．PDSSはあくまで擬似並列処理系なので，並列処理特有の再現性のない非決定的な動きはしない．一方，マルチPSIは本当の並列計算機であるから，タイミングしだいで同じプログラムでも動きが変わることもある．にもかかわらず，移植時にPIMOSのバグはほとんど出なかった．

マルチPSIとその上のPIMOSは，さまざまな実験的な並列応用システムの研究開発に利用されていった．この利用経験からのフィードバックを得て，PIMOS自身にもさまざまな面からの改良を重ねていった．

その後'91年から'92年にかけて，PIMの各モデルが順次完成し，PIMOSを移植して応用ソフトウェアの研究に利用していった．これらの各モデルはハードウェアアーキテクチャが異なり，同じプログラムでも並列処理時の実行順序はそれぞれ異なっている．しかし，512台のプロセッサをもつシステムに至るまで，どの移植の際にもタイミングの問題によるバグに悩まされることはまったくなかった．

当初の設計意図どおり，512台のプロセッサをもつシステムに至るまで，PIMOSによる管理はシステムのボトルネックになっていない．当初64台のプロセッサをもつマルチPSIだけを念頭において設計していたら，新たな大規模な並列マシンへの移植のたびに，大がかりな再設計が必要になっていたことだろう．

OSの開発で最も苦労するのは，バグに再現性のないことである．逐次計算機でもマルチタスクなら，並列計算機と基本的には変わりはない．普通の手続き的な言語では，プロセス間の同期はプログラム中に明白に記述しなければならない．KL1ではデータフローに従って自動的に同期が起き，同期のバグは出ない．このような言語を用いたのでなかったなら，PIMOSの開発ははるかに苦難に満ちたものになっていただろう．

†1 100万台というのは，対数をとれば定数に近いものとして無視できる範囲，という程度の意味である．プロセッサ台数の対数に比例するようなオーバヘッド増加は，問題にしないこととした．

5.2 既存のOSと違うところ

5.2.1 PIMOSの守備範囲

既存の「普通の」計算機システムでは，メモリ管理やプロセス管理がOSの重要な仕事となっている．たとえば，ユーザプログラムが自分以外のメモリ領域をアクセスしていないだろうかとか，複数のユーザプログラムを適当にスケジューリングする，といった仕事である．PIMOSの場合は，OS本体はKL1言語で記述されており，このKL1言語の処理系が，それらの基本的な資源の管理を行なってくれている．このためPIMOSは，基本的なメモリの割当てとかメモリ領域のアクセスチェックなど資源の低レベルな管理をする必要がない．

KL1言語は，LispやPrologなどの記号処理言語同様に，ガーベジコレクションを含む自動的なメモリ管理機能を有している．つまり，基本的なメモリ管理は言語処理系レベルで自動的に行なわれる．また，並列処理という観点から見れば，KL1言語処理系では，暗黙的な並行処理，データフロー的な同期処理，コンテキストスイッチング，スケジューリングなどをサポートしている．このように，PIMOSは低レベルで細かい単位のプロセス管理をする必要がない．一方で，もっと荒い粒度の仕事，たとえば複数のプロセスをグループ化して大きな単位で制御したりすることが必要となる．これには，やはりKL1言語で提供される優先度制御を利用している．

このように，メモリや細かい粒度のプロセスはKL1言語処理系で管理される．筆者らは，このような資源を，言語定義資源と呼んでいる．また，より高レベルの資源，たとえばウィンドウやファイルなどの仮想的な入出力デバイスはPIMOSによって制御される．筆者らは，この資源をOS定義資源と呼び，言語定義資源とは区別している．

PIMOSでは，ユーザプログラムの管理単位としてタスクを提供し，ユーザに提供する資源をすべて仮想的なデバイスとして，統一的に扱っている．入出力デバイスはいうまでもないが，タスクもまたデバイスの一つであり，タスクのなかで子供タスクを生成するような場合には，その子供タスクはデバイスの一つとして扱っている．

5.2.2 並列OSとしての気くばり

通常，OSでは管理するためのデータを表の形式でもっていて，管理＝「表のアクセス」となる．たとえば，既存のOSでは，あるユーザプログラムがどのファイルを開けているかとか，現在どのプロセスがI/O待ちにあるかといった情報を管理するために，ユーザプログラムをプロセス（もしくは，タスク[†1]）という単位で管理している．こういった管理を，通常のOSではOS内部の表に，プロセスがもっているファイルなどの資源を登録してある．

既存のOSの発想の延長線上として，並列計算機上にOSを作るとなると，これらの表を一元的に管理する方法をまず思いつく．しかし，この方法では，一つのプロセッサが表にアクセスしているときは，他のプロセッサがアクセスできないようにしたりする工夫が必要になる．このような方式でOSを実現すると，要求が集中して，並列処理のよさが失われてしまう．また，単純にこれらの表を分散してOS内部でもった場合，表の一貫性を保つ必要が生じる．このような方式は処理が複雑なばかりでなく，一貫性を保つために生じる通信のオーバヘッドがばかにならない．

PIMOSでは，この既存のOSの発想をやめ，ユーザプログラムを管理するための表を木構造にして分散させている．たとえば，あるユーザが仕事を始めると，そのユーザのトップレベルのためにPIMOSの側に管理用プロセスが作られる．このユーザが別のタスクを起動すると，そのタスクは元のタスクの子タスクとして登録される．この子タスクに対してもPIMOSは管理プロセスを作るが，この管理プロセスは親タスクの管理プロセスの下に局所的に作る．あるタスクが入出力装置を使おうとすると，それはそのタスクの子供として登録され，管理プロセスはやはり局所的に作る．このような管理プロセスの木構造が，集中管理テーブルの代わりをしている．だから，タスクや入出力装置の識別子は単なる整数ではなく，トップレベルから，誰の何番目の子供の，そのまた何番目かの子供，...，というようにたどっていくものになっている．

それぞれの管理プロセスは局所的に自分の担当している資源を管理しているだけであり，その全体は把握していない．というよりも，PIMOSでは全体を一元的

†1 これらの呼び方はOSによって異なる．

に把握している部分はない．もしも，全体の状態を知りたければ，木構造全体をトラバースする．トラバースしている最中にも，タスクや資源の状態は変わっていき，ある瞬間の全体の状態を知ることは不可能である．並列計算機の世界では，こういった微妙なタイミングのずれが現実問題として起こる．これを避けるために PIMOS では とりあえず局所的な管理をして，かつ それが他の部分と矛盾がないように設計している．

5.2.3 並列論理型言語で OS を書くとこうなる

OS 自身が並列に動作し，かつある程度規模の大きい PIMOS のようなプログラムを書く場合に，並列論理型言語で記述するのは非常に都合がよい．

論理型言語では，データのやりとりをするときに論理変数を用いるが，この論理変数は引数を通じてしかアクセスできない．アクセスできないと書くと否定的にとられてしまいそうであるが，逆にいえば，他のプログラムにデータを勝手に壊される心配がないということである．このことは，プログラミングをする上では非常に有利な性質である．たとえば，ある巨大なプログラムを複数の人間で作っているときに，「このデータはほかで書き換えられていない」ということを想定して書くのは，プログラマにとって非常に安心感がある．

さらに，並列計算機の環境でプログラムを作っていく場合，タイミングによってプログラムの動作が違うことを想定しないといけない．バグが出たときに，あるデータが壊されているのを発見しても，どこからでもデータを書き換えることができるのでは，それを発見するのは非常に困難な作業となってしまうからである．このような環境で言語が並列性とデータの一貫性をサポートしてくれると，プログラマの負担は激減する．

実際，PIMOS 自身は数万ステップで，プログラムを書く人も最大で 10 名という，研究システムの開発としては比較的規模の大きい開発を行なってきている．ところが，PIMOS を実際に並列マシンの上で動かし始めるとほとんどバグが出ずに動き出すということを経験してきており，筆者らが予想したよりもはるかによい結果が出て，驚いている．

一方，普通の OS では，あちこちで操作したいようなデータは大域変数に格納しておいて，アクセスするときにはそのデータをロックしたりして，ほかからアクセスできないようにする．ところが，論理型言語でプログラムを書くと，引数を通じてしかデータの受けわたしをすることができない．このことは，プログラムの設計を十分考慮した上でないとプログラムが作れないことを意味していて，開発チームのなかに「設計重視」という健全な風潮が当たり前のように存在していることも見逃せない事実となっている [†1]．

5.2.4 PIMOS の構成

PIMOS は，図 5.2.1 のような階層構成をもつ．

基本入出力部: 低レベルの入出力を仮想化し，PIMOS の移植性を向上させる役割をもつ．

資源管理: PIMOS の中核として機能する部分であり，ユーザプログラムの実行制御，デバイスの管理を行ない，クライアント–サーバ型の通信機能を提供する．

サーバ: PIMOS の入出力サブシステムは，サーバと呼ばれるタスクによって提供されている．サーバは PIMOS の資源管理部とは独立しており，システムのモジュラリティを向上させるのに役立っている．たとえば，ファイルシステムなどもこのサーバの形で提供している．

プログラム開発環境: PIMOS は，並列プログラムを効率的に開発するため，コンパイルやデバッグ，プログラムの性能チューニングを行なう種々のツールを提供している．

実行時ユーティリティ: PIMOS では，PIMOS 本体，応用プログラムの別を問わず，広く利用できる共通のユーティリティを提供している．これらのユーティリティは，実行時にユーザプログラムから述語呼出しによって提供される．

5.3 節では，PIMOS の基本的な機能である資源管理機能の実装方法について述べ，5.4節では，プログラム開発環境，ユーティリティの主な機能について述べる．

†1 実際，PIMOS の中心である資源管理部は，第 3 版で全面的な書き直しを行なった．

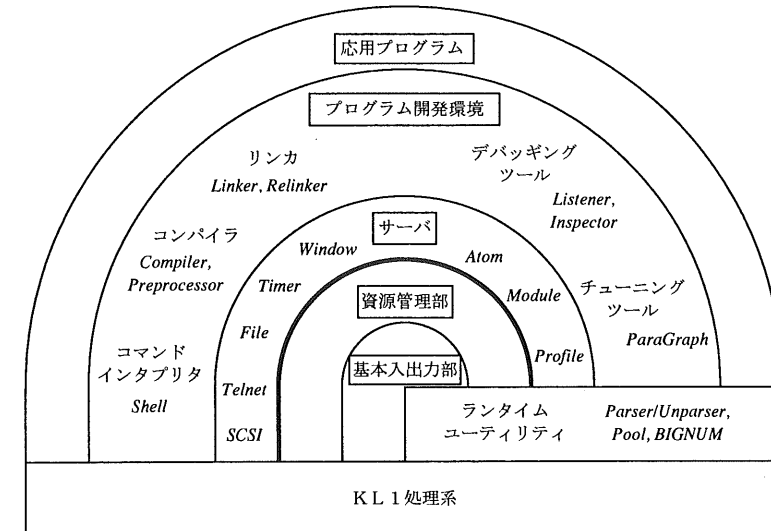


図 5.2.1 PIMOS の構成

5.3 特徴的な実装

5.3.1 タスクの実現方式

KL1 は他の並列論理型言語同様，バックトラック機構をもっていない．このため，KL1 で記述した OS とユーザプログラムを同じ実行環境で実行すると，ユーザプログラムの失敗によって OS も失敗することになる．ユーザプログラムが失敗するたびに，OS を立ち上げ直すようなシステムでは誰も使ってくれないから，ユーザプログラムの失敗から OS を保護するような仕組みが必要となる．この仕組みを実現するために，KL1 では「荘園」という機能を提供している (図 5.3.1 参照)．

「荘園」は KL1 のゴール列をひとまとめに管理する機能であり，まとめられたゴール列の失敗を他のゴールに波及させない機能をもつ．具体的には，荘園は次の組込み述語によって呼び出すことができる．

```
execute(Program,
        MinPrio, MaxPrio, ExcepMask,
        Control, ^Report)
```

引数 Program は，荘園のなかで実行される最初のゴールである．この最初のゴールの子孫ゴールは，すべてこの初期ゴールが与えられた荘園のなかで実行される．

荘園には，制御ストリームと報告ストリームと呼ぶ二つのストリームがついている．これらのストリームはそれぞれ，execute/7 の引数 Control，Report で表現される．制御ストリームは荘園に属するゴール群全体の実行を制御するために用いられる．制御の種類としては，実行の開始/停止/再開/強制終了などがある．これらの制御は，いずれも制御ストリームに対して，対応する (start, stop, resume, abort) メッセージを流すことによって制御する．また，荘園内部で起きたゴールの失敗，デッドロック，数値計算でのオーバフロー，ゴール群の実行終了などは例外事象として扱われ，報告ストリームからのメッセージで報告される．

引数 MinPrio，MaxPrio は荘園に属するゴール群の優先度範囲を示す．PIMOS は，KL1 で記述された細かい粒度で動く並列プロセスのスケジューリングは行なわないが，制御ストリームとこの優先度のメカニズムを使うことにより，荘園に属するゴール群全体の制御を行なう．

荘園のなかでさらに荘園を作ることもできるので，荘園は任意のレベルの入れ子構造を構成することができる．ある荘園を停止させると，その子供の荘園，孫の荘園も止めることができる．引数 ExcepMask は，この execute/7 を実行したレベルで，どの種類の例外の報

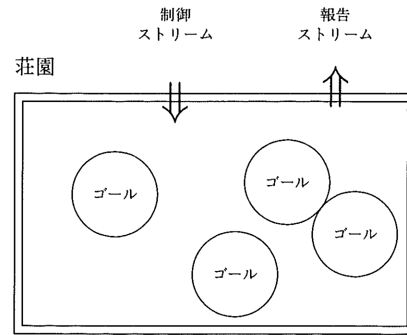


図 5.3.1 荘園

告を受け取るかを指定するためのマスクである．もし，報告を受けたくないようなときは，このマスクを設定することによって例外が発生しても，その報告をより上位の荘園に報告するようにできる．

PIMOS ではこの荘園の機構を利用して，ユーザプログラムを管理している．ユーザプログラムと PIMOS の間の通信路を設定するためには，荘園の例外報告機構を用いている．例外事象をユーザプログラム中で明示的に発生させるために，次の組込み述語をプリミティブとして用いている．

```
raise(Tag, Data, Info)
```

引数 `Tag` はこの組込み述語によって生成する事象の種類を示す．前述のように，荘園の生成時には例外マスクが設定できるので，この例外を荘園の階層のどのレベルで扱うかを決めることができる．

残りの二つの引数 `Data` と `Info` は，例外事象の詳細な情報をわたすために用いられる．Data は，常に具体化されていることが保証されているデータであり，`Info` は変数でも構わないことになっている．この組込み述語は，引数 `Data` が論理変数を含まない項に具体化されるまで，中断される．

報告ストリームを監視することにより，PIMOS はユーザからの要求を次の形式のメッセージとして受け取る．

```
exception(EventInfo,^NewProgram)
```

引数 `EventInfo` は例外事象の種類とその情報を示す．この `EventInfo` を見ることによって，たとえば，例外が組込み述語 raise によって発生したものであるかどうかを判別することができる．

また，引数 `NewProgram` は，例外を発生させたゴールの代わりに実行させるゴールである．たとえば，荘園のなかで整数の除算が 0 除算のため失敗したような場合には，荘園の報告ストリームから

```
exception(
    zero_division(
        divide(0, 100000, Result)),
    NewProgram)
```

のように報告される．ここで，プログラムの実行を継続させるために，NewProgram に対して

```
NewProgram =
    (unify(Result,16#FFFFFFFF))
```

のように置き換えれば，さらに実行を進めるようなことができる．

raise を使って設定される PIMOS とユーザの間の通信路にも，この機構を使って保護フィルタと呼ばれるプログラムを挿入している．なお，この保護フィルタについては，あとで詳細に述べる．

5.3.2 通信方式

A. ストリーム通信

並列システムにおいては，さまざまな資源を効率的に管理することは，逐次型のシステムよりも難しくなる．たとえば，OS がメモリ中のある特定のデータを読み書きする間，ユーザプログラムがそのデータを読み書きさせてはいけない場合がある．逐次システムの OS ではそのような場合，ユーザプログラムの実行を単純に中断させればよい．そのような方法を，並列システムにそのままもち込むと，ユーザプログラムが際どい領域のデータをアクセスするか否かにかかわらず，並列に動作している全ユーザプログラムを止める必要性が生じてしまう．これでは並列，特に大規模な並列システムでは，細粒度の並列性を生かすことができない．

PIMOS とユーザプログラムとの通信は，並列論理型言語でよく用いられているプロセス指向的なプログラミングスタイルで記述している．このプログラミングスタイルでは，プロセスが外部からアクセスできな


```
?- pimos(Req), user(Req).

   user(Req) :- true |
        Req = [get(String)|ReqT],
         ......  .

   pimos([get(String)|ReqT]):- true |
        readFromKbd(KBDString),
        KBDString=String,
        pimos(ReqT).
```

図 5.3.2 プロセス間通信の例

いような内部データをもつことができ，プロセス間通信のための共有変数をもっている．プロセス間通信はプロセス間で共有しているデータを，順番に具体化することによって実現している．

変数の具体化はデータの送信を意味し，同じ共有変数を待つプロセスがその具体化されたデータを受信することになる．このメッセージの送受信はリストの要素を順番に送受信する形で表現され，このリストが通信路としての役割をもつ．データの順番が保証されるので，並列プログラムを記述する際には，このプログラミングスタイルは非常に都合がよい．

ここで，PIMOS とユーザプログラムとの通信を，簡単な例を使って考えてみよう．キーボードから文字列を読み出すような場合である (図 5.3.2 参照)．ユーザは，PIMOS との間のストリーム `Req` にメッセージ `get/1` を送ることによって文字列を受け取る．このメッセージの引数 `String` は，変数のまま PIMOS 側に送られ，この変数に PIMOS からの応答が返る．pimos プロセスは `ReqT` をストリームとして用い，自分自身を再帰的に呼び出し，ユーザの次の要求のための処理を行なう．

B. 非同期通信

ストリーム通信は簡単な応用に対しては単純かつ十分強力なものであるが，いろいろな入出力デバイスを同時に制御する場合には，あまり柔軟なものではない．並列処理においては，通信の遅れが重要な問題となるが，そこでストリーム通信のようにデータを次から次にパイプラインで送信するとスループットがよくなる．

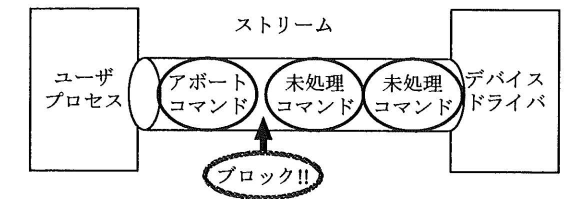


図 5.3.3 ストリーム通信のブロック

しかし，このようなパイプライン処理で，レスポンスもよくしようとすると，次の機能が必要となる．

- 本当に必要かどうかを確認する前にメッセージを送ってしまう機能．
- メッセージを送ったあと，そのメッセージが不要になったらそのデータをキャンセルしてしまう機能．

OS とユーザの間の通信をストリームで実現した場合，前者の機能は満足できる．しかし，後者の機能を実現するために，キャンセルのためのメッセージを同じストリームで送ろうとしても，そのメッセージは前に送った未処理のメッセージに邪魔されてしまうので，キャンセルはできなくなってしまう (図 5.3.3参照)．

この問題を解決するため，PIMOS はメッセージのやりとりをする通信路であるストリームとは別に，緊急用の通信路を設けている．この通信路はアボートラインと呼ばれ，OS とユーザ間の共有変数で実現されている．もし，先に送ったメッセージをキャンセルしたい場合には，この変数をユーザ側で具体化する．

ユーザと OS 間でただ一つの通信用のストリームだけしかもたない場合，デバイスから非同期的な情報をユーザにわたす方法がないことがあげられる．この問題を解決するため，アボートラインと同じような考え方で，逆向きに，デバイス側からユーザ側への通信路を設けることで対処している．この通信路のことをアテンションライン と呼んでおり，PIMOS の通信はストリーム，アボートライン，アテンションラインの三つの通信路によって実現している．(図 5.3.4 参照)．

この三つの通信路を設けることによって，スループットが高く，応答性もよい通信が可能となっている．

これらの二つのラインは，その性質上，一回具体化してしまうと次には使えない．したがって，これらア

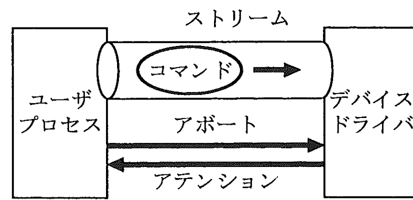


図 5.3.4 デバイスとの非同期通信

ボートライン，アテンションラインが使われてしまったあと，通信用のストリームから reset メッセージを送って，新しいアボートライン，アテンションライン用の変数を新たにユーザ側に送る．

C. 通信の多重化

複数のプログラムから資源を共有したいという要求は，通常の計算機システムでも当然ある．たとえば，あるシェルの下で複数のプログラム (子タスク) を実行する場合，子タスクが親のタスクであるシェルと端末を共有する．このような場合，ある一つのタスクだけが資源を排他的にアクセスできるとうれしい．また，端末からの割込みによって実行中のタスクを止めるような場合には，応答性がよくないと使い勝手が悪い．その一方で，入出力のスループットを向上させるためには，パイプラインなどの先送りするメカニズムが必須となる．このような相反する要求を，先に述べたアボート，アテンションラインのメカニズムを使うだけでは実現できない．

この問題を解決するため，PIMOS はさらに次の入出力メッセージを提供している．

reset(^Result): ユーザ側からこのメッセージを送ると，変数 Result は normal(^Abort,Attention, ID) に具体化される．引数 Abort と Attention は，それぞれ新しいアボートラインとアテンションラインに対応する．最後の引数 ID は，このメッセージを流したストリームに送ったメッセージ列の識別子を示す．このメッセージ列とは，reset メッセージを送ってから，次にアボートされるまでに送った一連のメッセージ列のことである．

resend(ID,^Status): アボートラインを使って

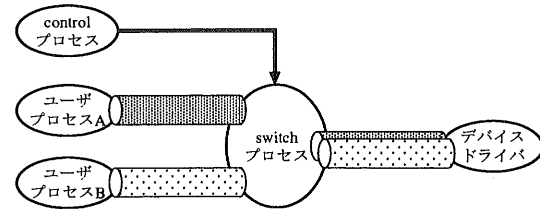


図 5.3.5 ストリームの多重化 (1)

入出力の要求メッセージがアボートされた場合，デバイスドライバは，メッセージ列とそのメッセージ列に対応した識別子を記憶しておく．この reset メッセージは，デバイスドライバで記憶している識別子 ID に対応するメッセージ列を再送するためのメッセージである．先に述べたように，このメッセージ列は，アボートされて未処理のままデバイスドライバに蓄えられたメッセージ列である．

cancel(ID,^Status): この cancel メッセージはデバイスドライバに対して，識別子 ID に対応するアボートされたメッセージ列を破棄するためのものである．

端末デバイスが二つのプロセス A,B に共有されている場合を例にとって考えてみよう．各ユーザプロセスは，端末デバイス一つの通信路を共有している．ユーザプロセスから出た各通信路は switch プロセスにマージされており，この switch プロセスには control プロセスと呼ばれるプロセスに接続されるストリームで継がれている (図 5.3.5 参照)．

control プロセスは通常，シェルのようなプログラムでよく使われるプログラムである．シェルなどのプログラムは，複数のプログラムで一つの端末デバイスを共有させるようなメカニズムをもっている．この control プロセスで，複数のユーザプログラムから送られるメッセージを制御する．

たとえば，シェルの下で動いているプログラムがキーボードからの割込みで実行を中断させられるような場合には，次のようなことが考えられる．

実行を中断させられたプログラムが文字列を表示するなどのために，すでに端末に送ってしまっているが，割込み用のキーをユーザが打った時点では，まだそのコマンドは処理されていないような場合がある．この

ような場合，control プロセスはアボートラインを具体化し，switch プロセスが resend メッセージを発行する．アボートラインを具体化することによって，中断させられたメッセージは未処理のまま，デバイスドライバ内に識別子 ID とともに蓄えられる．もし，このプログラムの通信を再開させる場合には，コントロールメッセージは識別子 ID を引数として，resend メッセージを発行する．このときの識別子 ID は，中断させた入出力の要求を再開させるために用いる．

図 5.3.6 は，上述の仕組みを使って，ユーザプロセス A, B の通信路が切り替わる様子を示したものである．control プロセスから送られる abort, reset, resend メッセージの組によって，通信路が切り替わっていく様子を示している．

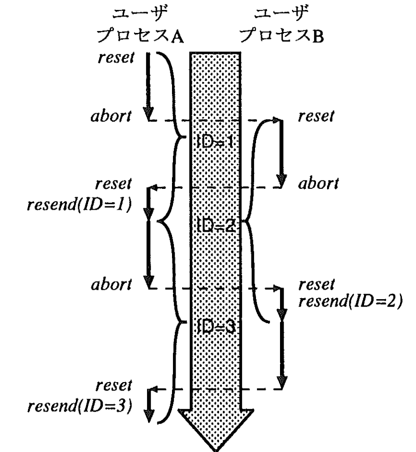


図 5.3.6 ストリームの多重化 (2)

5.3.3 保護フィルタ

並列論理型言語をベースにしたシステムでは，通常の OS で問題となるような保護の問題はあまり表面にでてこない．これは，ユーザプログラムと OS との間の通信がすべて論理変数を共有することで実現されているためである．このため，ユーザプログラムが OS によって使われているメモリ領域をアクセスできない．また，ユーザプログラムの失敗は，先に述べた荘園というメタレベルの制御機能を使うことによって保護できる．

しかし，この論理変数を共有するという単純な機構だけでは，ユーザプログラムで起きるエラーが，次のようなシステムのエラーとなってしまう場合がある．

多重書込みによって発生する問題: OS とユーザプログラムで共有している同じ論理変数に対して，それぞれが違う値を書き込むと unification が失敗する．並列論理型言語である KL1 は，PIMOS とユーザプログラムとの間の unification は並行して実行される．このため，PIMOS の具体化が遅ければ，PIMOS が失敗するという事態が発生し得る．

永久中断によって起きる問題: ユーザプログラムは PIMOS に送られるメッセージの引数を具体化し忘れるような場合，PIMOS はその中断を永遠に待つことになってしまう．

これらの問題を解決するために，PIMOS では保護フィルタと呼ばれるフィルタプロセスを導入している．この保護フィルタは PIMOS とユーザプロセスとの間のフィルタプログラムとして機能する．先にも述べたように，PIMOS とユーザプログラムの間の通信路は，荘園の例外報告機能 (raise) によって実現している．具体的には

```
exception(EventInfo,^NewProgram)
```

という形式のメッセージが，荘園の報告ストリームから報告される．この exception メッセージの引数 NewProgram に保護フィルタプログラムを unify すると，ユーザの荘園のなかでこのプログラムが実行される．このフィルタプログラムは，OS から返される結果については，必ずユーザの荘園のなかで unify するので，もし unification の失敗が起きても，失敗するのはユーザプログラムだけであり，OS 側にエラーが波及するようなことはない．また，PIMOS に送るはずのメッセージがすべて具体化されるまで，PIMOS に送らない．このため，もし PIMOS に送るはずのメッセージが具体化されなければ，永久中断するのはユーザの荘園だけで，OS 側に問題は起きない．

図 5.3.7 は，保護フィルタのプログラム例である．この例のように getc/1 というメッセージの引数の値を OS 側から受け取るまで，wait_and_unify/2 というゴールで待つようになっており，wait_and_unify/2

```
filter([get(C)|User],OS):-
    true |
        OS = [get(C)|OS1],
        wait_and_unify(C1,C),
        filter(User,OS1).
wait_and_unify(OSV,UserV) :-
    wait(OSV) |
        UserV = OSV.
```

図 5.3.7 保護フィルタの例

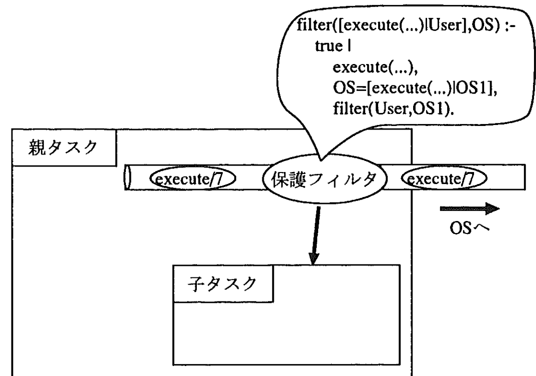


図 5.3.8 タスクの生成方法

のボディゴールとして unify が実行される．このため，OS からの値とユーザからの値が違う値で unify されても，unification の失敗はこのプログラム，すなわちユーザの荘園内で起きるだけで済む．

この保護フィルタは，メッセージをチェックする役割をはたしているわけであるが，逆にいえば，各メッセージごとにプログラムを書く必要がある．この保護フィルタのプログラムを人手で記述するのは面倒なので，メッセージのプロトコルを記述するだけで保護フィルタのプログラムを自動的に生成するツールを用意している．

また，この保護フィルタのメカニズムは，メッセージのチェックだけでなく，ほかにも流用されている．たとえば，子供タスクを生成する場合である．タスクは荘園を用いて実現されているが，この荘園を生成するプリミティブは組込み述語によって実行される．一方，ユーザから OS に対しての要求はすべてメッセージの形式で実現される．OS が荘園を生成する組込み述語

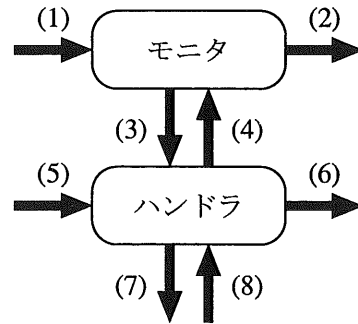


図 5.3.9 局所資源管理モデル

を実行してしまうと，すべて OS の直下に荘園ができてしまい，多重レベルの管理や資源の消費量などが階層的に実現できない．

このため，タスク生成のメッセージがユーザ側から届くと，NewCode としてユーザの荘園のなかに，荘園を生成するための組込み述語のプログラムを埋め込む．NewCode はユーザの荘園のなかでできるので，子供タスクの荘園は，子供の荘園としてできる仕組みとなっている (図 5.3.8 参照)．

5.3.4 資源管理

A. 階層的な資源管理

前述したように PIMOS では，タスクは KL1 で提供されている荘園を用いて実現している．PIMOS 資源管理部は，この荘園を監視する KL1 プロセス群で構成されている．PIMOS では，この監視プロセス群自身の並列性を保証しつつ，他のプログラムの邪魔をしないようにする工夫をしている．

PIMOS 資源管理部の監視プロセスは，割り当てた資源ごとにモニタプロセスとハンドラプロセスの二つのプロセスで監視を行なう．モニタは，他の監視プログラムから送られてくるメッセージを解釈して，自分が監視している資源にするメッセージかどうかを判断する．一方，ハンドラプロセスは，自分の監視している資源との通信路をもっていて，資源の解放など具体的な管理の仕事をする役目となっている．

これらの監視プロセスは，図 5.3.9 のような構成となっている．図中の矢印は，プロセス間の通信路であり，次のような役目がある．

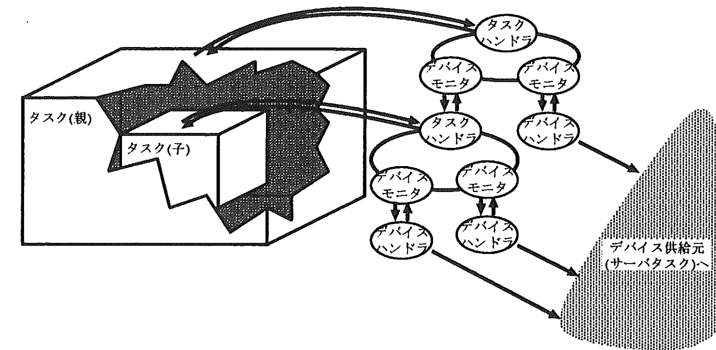


図 5.3.10 資源の階層的管理

(1), (2): 親タスクのハンドラ，または兄弟資源のモニタから/への通信路

(3), (4): 他資源から/へのメッセージの入出力用通信路

(5), (6): 自分の資源から/への通信路

(7), (8): 子供資源をもつ場合 (現在，タスクのみ) から/への通信路

PIMOS では，タスクも仮想的なデバイスとして扱っており，他のデバイス同様，これらの監視プロセスが監視している．タスクのハンドラプロセスは荘園の制御ストリームと報告ストリームを監視していて，たとえばユーザがタスクを強制終了したような場合には，図中の (1)，(3) のパスでメッセージを受け取り，(6) のパスで終了メッセージを荘園に送り出す．

タスクが子タスクをもつ場合には，図 5.3.9 中のハンドラの (7) とモニタの (1) が接続され，(2) が次の資源のモニタの (1) に接続される．これを繰り返し，最後に (8) に接続される．最終的には，図 5.3.10 のような階層構造を構築していく．PIMOS では，この階層的な資源管理機構が木のように構築されていることから，「資源木」と呼んでいる．

これらの監視プロセスは，局所的に，自分の与えられた資源を監視しながら，かつ外界とメッセージのやりとりをする．たとえば，PIMOS 全体の終了はトップレベルの監視プロセスからメッセージを流し込むことで，資源木をトラバースしながら木の葉の資源から順番に終了していく．木の葉のほうで資源が終了した場合には，影響のある部分木にしかメッセージが流れないので，他の部分には影響を与える心配がない．

また PIMOS では，これらの監視プロセスの負荷分散も行なう．といっても，負荷の状態を見て，積極的に移動するというわけではなく，要求されたところに出張するというポリシーで負荷分散を行なっている．負荷分散を行なうタイミングは，タスクを含む仮想的なデバイスを生成するとき，そのデバイスに対応するハンドラプロセスが要求された場所に出張するようになっている．モニタプロセスはタスクハンドラと同じ場所に作られる (図 5.3.11参照)．

B. サービスの提供方法

PIMOS の中核である資源管理機構を最小限にとどめるために，PIMOS 資源管理部は基本的な機構だけを提供している．たとえば，ファイルやウィンドウのようなデバイスのサービスに関しては，サーバと呼ぶタスクによって提供している．

先にも述べたように，ユーザプログラムと PIMOS の間の通信路は KL1 の組込み述語である raise/3 によって確保される．しかしながら，このメカニズムは資源管理部との通信路を確保するだけであり，サーバタスクとの通信路は確保できない．

クライアントタスクとサーバタスクの間の通信路は次のようにして確保される (図 5.3.12 参照)．

1) サービスを開始する際にはまず，サーバは資源管理部の内部にあるサービス表と呼ばれる表にサー

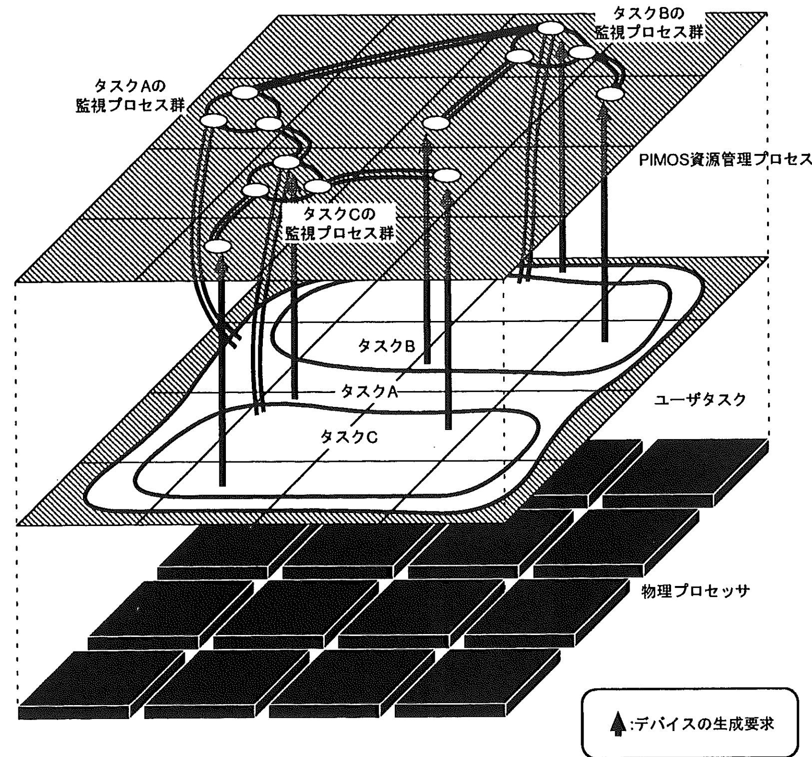


図 **5.3.11** 資源管理プロセスの負荷分散

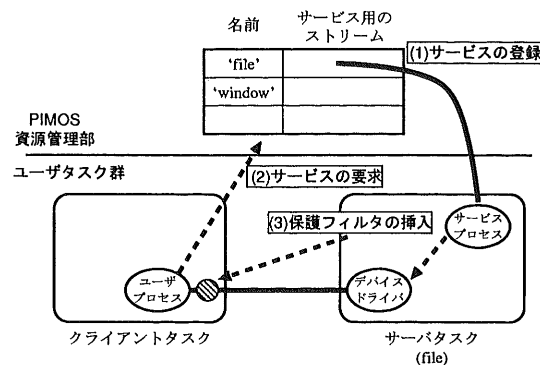


図 **5.3.12** クライアント–サーバ間の通信 (1)

ビスを登録する．サービス表のエントリには サービス名とサーバへのストリーム，およびそのサーバに対するメッセージをチェックするための保護フィルタから構成される．

2) クライアントタスクは，raise/3 組込み述語を用いて，PIMOS 資源管理部へのストリームを張り，名前によってサービスを要求する．

3) 資源管理部はサービス表にある名前を探し，もし同じ名前のサービスが見つかれば，クライアントとサーバを結びつける．

上述の順番は典型的な例であり，実際の 1) と 2) の順番はどちらが先でもよい．もし，クライアント側の要求がサービスの登録より先に資源管理部に到着した場合には，登録が完了するまで要求を中断させることができる．

また 3) で，PIMOS の資源管理部はデバイスモニタとデバイスハンドラをクライアントタスクのデバイスとして生成する．デバイスハンドラは，クライアントタスクの終了を監視する．もし，デバイスハンドラが終了を検知したら，デバイスハンドラはデバイスの終了をサーバ側に通知する (図 5.3.13 参照)．

PIMOS では，このように資源管理部と資源を提供するサーバが分離されているため，システムを柔軟に構成することができる．また，サーバは一般のタスクとして実現されているので，サーバのエラーが資源管理部にまで直接波及することがないので，堅牢なシステムを構築することが可能となっている．

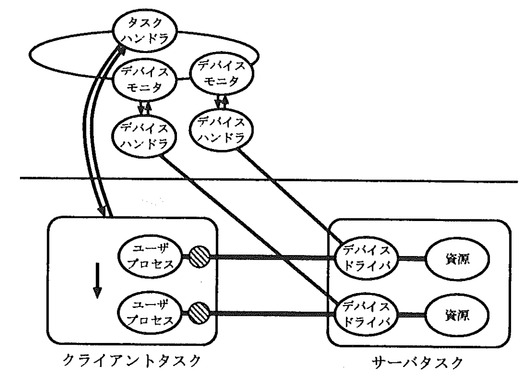


図 **5.3.13** クライアント–サーバ間の通信 (2)

5.4 プログラム開発環境

ここでは PIMOS のプログラム開発環境について紹介する．開発環境を支える主なツールは，コンパイラ，デバッガ，そしてプログラムチューニングツールである．チューニングツールが開発環境を支える重要なツールの一つに数えられるのは，並列プログラム開発環境ならではの大きな特徴の一つであろう．プログラムが一通り正しく動作するようになると，最終段階として性能改善が問題になる．逐次プログラムだったら，適当にプロファイル情報を取って主処理を高速化すればある程度納得できるものが得られるので，プログラムチューニングツールなどと大げさな名のつくものは必要ではないかもしれない．しかし，並列プログラムの場合はそう簡単にはいかない．プログラムを構成するたくさんのプロセスが，時間や場所という異なる次元で複雑に絡み合って動作するためである．プログラマはプログラムの振舞いや性質をよく理解して，優しく手なづけるようにチューニングしなければならないのである．そのためには，そうした振舞いを解析してわかりやすく表示してくれる便利なツールが必要になる．

コンパイラやデバッガも並列環境ならではの特徴を備えている．実行コードの管理方法が普通のシステムとだいぶ異なるので，コンパイラやその周辺ツールもそれに合わせた処理をしている．並列プログラムのデバッグは，逐次のものより難しい．複数の計算が同時に進行したり，それらが互いに干渉し合ったりするために，全体的にプログラムの振舞いが複雑になってしまうためである．プログラムにバグがあるとデッドロックという現象が起こったりすることもあるし，計算のタイミングによって，ときには再現性のないバグにぶつかってしまうこともある．デバッガは，そういった並列プログラム独特のデバッグに対応できるよう，さまざまな機能を搭載している．

ところで，ユーザが PIMOS を利用する場合の周辺環境について一言．PSI を端末とすれば，すべての機能を遠隔利用することができる（図 5.4.1はユーザによる利用例）．端末が PSI でなくても，telnet でアクセスすればほとんどの機能を利用できるし，その際 X ウィンドウシステムが動作する環境があればプログラムチューニングツールも使える．PIMOS は独自のプログラム編集用ツールをもっていないが，emacs などユーザ各自のローカルシステム上で提供されているものを適当に用いればよいだろう．

5.4.1 実行コードの生成

KL1 プログラムを実行するには，まずコンパイルして実行可能なコードを生成する．KL1 処理系のコード管理方式は並列処理向けに設計されていて，普通のシステムとはだいぶ異なる．たとえば，処理系はコードの名前とコード自身を対応づけるというところまでは管理しないので，PIMOS が管理している．ここでは，PIMOS のコード管理のやり方と，コード生成に関するツールについて紹介する．

A. PIMOS のコード管理

KL1 プログラムにはモジュールという単位があり，なかに書かれている述語の実行コードはこのモジュール単位でまとめて，一つの実行の実体として生成される．述語単位でないのは述語呼出しや GC を高速化するためである．このモジュールごとにまとめられたプログラムの実行の実体は，モジュールという型のデータである．モジュールの先頭にはモジュールヘッダという部分があって，そのモジュールに関するさまざまな

図 5.4.1 PIMOS のプログラム開発環境

情報が定義されており，そのあとに複数の述語のコードが続くような構造になっている．

KL1 処理系では，あるモジュール名のプログラムの実体であるモジュールが複数版同時に存在してもよいことになっている．これはいうなれば，並列処理だからである．たとえば，あるモジュールを更新しようとしたまさにそのときに，そのモジュールは他のプロセッサで実行中かもしれない．これを上書きしてしまうことにすると，すでに実行されているものの動作内容がいつの間にか変わってしまうことになり，混乱の元である．一般に並列処理と副作用は相性が悪いから，コード管理についても副作用なしという方針を貫くことですっきりと整理されている．しかし，これではプログラムと実体との対応が容易につかない．現在実行中のモジュールはプログラムのどのバージョンだろう，新しいプログラムをコンパイルするにも，モジュール間のリンクを張るには一体どのモジュールを参照するのがいいのだろう，ということになってしまう．そこで，PIMOS がモジュール名とモジュールとの対応を管理している[†1]．具体的には，PIMOS はモジュールデータベースをもっていて，常にモジュール名から最新版のモジュールを参照できるようになっている．PIMOS のコンパイラが新しいモジュールをコンパイルしたときにはここに登録し，またモジュール外呼出しによるリンクを張る場合にはこれを参照する．

さて，プログラム実行の実体であるモジュールは，そのプログラムがコンパイルされたプロセッサのメモリ上に生成される．では，そのプログラムが他のプロセッサから呼び出された場合はどうなるのだろう．KL1 処理系では，モジュールも普通のデータと同様に扱うことになっている．たとえば，あるプロセッサで他のプロセッサのデータが必要になると，そのときに読みに

†1　複数のユーザが同時にシステムを利用すると，まれに異なるユーザが予期せず同じ名前のモジュールを登録（または変更）しようとして混乱することがあり，具合が悪い．そこで，よくほかのシステムにあるのと同様，PIMOS もモジュール名空間を多重にするためのパッケージ機能をもっている．

いく．モジュールの場合も同様である．あるモジュールを実行しようとしてそのプロセッサにないことがわかると，そのときに読みにいく．つまり，モジュールはプロセッサごとに要求に合わせて配布されるようになっている．このようにしてプロセッサに配布されたモジュールは，実行が終わって誰からも参照されなくなると，普通のデータと同様に GC によって消去される[†1]．

B. プログラムをコンパイルする

コード管理についておおまかにわかったところで，具体的に PIMOS でプログラムからモジュールを生成する方法を紹介しよう．つまりコンパイラの話である．

コンパイラは，まずマクロ展開[†2] などの前処理を行なってから，コンパイル，アセンブル，リンクという三つの作業を経てモジュールを生成する．コンパイル段階では，ソースプログラムを解析していったん KL1-B と呼ばれる抽象機械語命令列を生成する．その際，実行時に高速に節を選択するための手法であるクローズインデキシングのための解析も行なう．KL1 では変数の入出力関係を決定しやすいので，最終的に実行すべき節を絞り込めるところまで徹底的に絞り込むような索引を作成するようになっている．また，変数の参照数を解析して MRB-GC サポートのための命令語を生成し，各変数にレジスタ割りつけをする．複雑で大きなプログラムをコンパイルするとレジスタが不足してしまうこともある．そのような場合には，一時的にレジスタの内容をメモリに退避して，見かけ上はレジスタ数の制限がないようにしている[†3]．また，述語の存在判定もここで行なう．普通の言語でコンパイラを書いたりすると，述語の飛び先判定は合計 2 回の走査で行なう必要がある[†4]のだが，PIMOS のコンパイラは KL1 自身で書かれており，差分リストを用いたストリーム通信を巧みに利用することによって，1 回の走査で完了することができるようになっている．このようにして KL1-B の命令列を生成し，アセンブル段階で対応する機械語命令列に変換してモジュール内の参照関係を解決し，最後にリンク段階でモジュール外の参照関係を解決する．このような一連の処理が成功してモジュールが生成されると，コンパイラは PIMOS のモジュールデータベースにこのモジュールを登録して，作業が完了する．

コンパイラは，ソースファイル単位で負荷分散することによって処理を高速化している．具体的には，最初に各プロセッサにそれぞれ一定数のファイルをコンパイルするように割り振っておき，処理が終了したプロセッサに順次，未処理のファイルを割り当てていくという方式である．

また，モジュールにはモジュール生成日時などのタイムスタンプ情報やソースファイル名なども書き込まれている．これらの情報を基に，プログラムを構成するモジュールのそれぞれについて，更新されたソースファイルを自動的に探索してコンパイルすることもできる．これを Unix にある同様の目的の機能にならって make 機能という．

C. モジュールを再リンクする

さて，あるモジュールを再コンパイルして更新したとしよう．こんなとき，もし更新したモジュールを参照しているモジュールがあれば，古いモジュールを参照するのをやめて新しいほうを参照するように変更してほしいものである．当然，そうやってあるモジュールを変更したら，それを参照しているモジュールも，それをまた参照しているモジュールも，というように．ところがコード管理の説明で述べたとおり，すでに存在するモジュールを上書きする（書き換える）ような処理は行なわないことになっている．そこで，再リンクという操作を行なうことによって，モジュール間の参照関係をメンテナンスする．再リンクはまず，参照関係を変更したいモジュールの内容をコピーして，参照関係だけを変更したモジュールを新しく生成する．モジュールをコピーするので副作用がない．つまり，動作中のモジュールを更新しても，その実行には影響がないというのが利点である．参照関係を変更するだけのために関連するモジュールを全部再コンパイルするという方法もあるが，KL1 のコンパイラはいろいろと複雑な解析をしながらコードを生成するので，たいていは再リンクするほうが手っとり早い．

†1　PIMOS のモジュールデータベースにだけは登録されたモジュールが最後まで残る．定められた操作をすれば不要なモジュールを消去することができる．
†2　プログラムを書きやすく，読みやすくするために，KL1 もマクロ機能をもっている．
†3　普通の（C のような）言語では当然のことかもしれないが．
†4　述語定義があるかどうかの判定は，問い合わせた時点ですでに登録済みならば問題ないが，その時点で未登録だった場合，ファイルに吐き出すなどして後に再度チェックする必要がある．

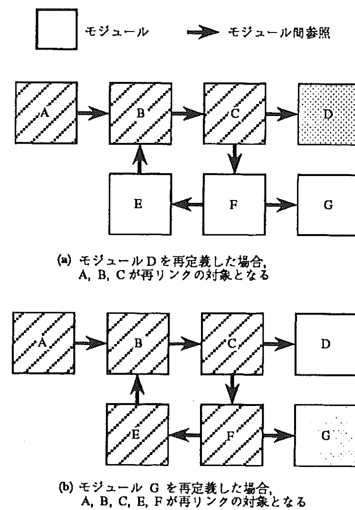


図 **5.4.2** モジュールの再リンク

D.　モジュールをアンロード/ロードする

コンパイルした結果, 生成されるモジュールはメモリ上に存在するだけなので, 電源を落とせば消えてしまう. そこで, ユーザが開発したプログラムを再び実行するには再度コンパイルする必要があるのだが, これでは効率が悪い. そこで, メモリ上のモジュールをファイルに保存することをアンロードという. こうしてモジュールをアンロードしておけば, ロードという機能によってアンロードされたファイルから再びモジュールを生成することができる.

モジュールをアンロードするときには, バイナリとしてそのままファイルに書き出すことはできない. モジュールには機械語命令のほかにアトムや外部モジュール呼出しなどが含まれていて, たとえばこれらのアトム番号は, 次の機会にロードするときには変わっている可能性があるからである. そこでモジュールをアンロードする場合は, モジュール内に現われたすべてのアトムについてアトム番号をそのモジュール内での相対番号に変換し, アンロードファイル内に相対アトム番号とアトム印字名の対応を記述するテーブルを置く. 外部モジュール呼出しは, モジュール名, 述語名, 引数個数の対で指定されるが, これらの名前情報はすべて相対アトム番号で表わすように変換する. アンロードされたファイルから再びモジュールをロードする場合には, これらの逆変換を行なう.

E.　コンパイルインタフェース

このように書いてくると, プログラムを実行可能な状態までもっていくにはコンパイルしてから再リンクするという一連の操作が必要で, 結構面倒そうである. ところが, 実はそうではない. PIMOS のコンパイラは, これらの一連の操作を自動的にこなす自動リンクサービスを行なっており, 普通 ユーザはコンパイルしたいソースファイル名を指定するだけで十分である.

さらに PIMOS のコンパイラは, 単に KL1 専用のコンパイラではなく, KL1 をベースとする上位言語の開発にも対応できるような枠組みをもっている. これは, ある言語からある言語への変換を行なうプリプロセッサの列をユーザが任意にコンパイラに追加することができるというもので, 変換の中間コードをファイルに出力することなく, 上位言語から KL1 プログラムを経て自動的に実行可能なコードまで変換することができる.

たとえば, コンパイラが前処理として行なうマクロ展開は, マクロによる記述を含む標準の KL1 言語からマクロを含まない KL1 言語へ変換する一つのプリプロセッサと考えることができる. また, コンパイル前にプログラムの静的解析や検証を行なうベリファイという機能も, プリプロセッサの一つとして組み込まれている. ベリファイ機能は, 将来はもっと複雑な静的解析を行なうかもしれないが, いまのところはプログラムの節単位でそれぞれの変数の出現回数をカウントし, 出現が一度きりの変数を検出するというものである. プログラムを書いたときに変数名の綴り間違いによってデッドロックが起こってしまうことも多いものだが, そのようなうっかりした間違いは, この機能を使えば簡単に検出できる.

このようなプリプロセッサの追加機能での難しさは, デバッグ時にエラーメッセージとソースプログラムの対応をつけにくいことである. PIMOS では, 実行時のエラーやデバッガによるトレースにはまだ対応できていないが, コンパイル時に生じるエラーについては対応しており, エラー報告時にエラー内容とともにソースプログラムに関する情報を表示することができる.

また, 現在のコンパイラではすでに述べたように, ソースファイル単位での負荷分散しか行なっていないが, プリプロセッサがパイプラインとして動作していることから, プリプロセッサ単位で負荷分散を行なう

ことも考えられる. これは今後の課題である.

5.4.2　プログラムのデバッグ環境

いよいよリスナを使ってのデバッグである. リスナは, 述語のトレースやスパイといったデバッグのための基本機能をはじめ, KL1 プログラムのデバッグ中に陥りやすいデッドロック現象についても有益なデバッグ情報をユーザに報告するなど, さまざまなデバッグ機能をもっている. 並列プログラムは逐次プログラムに比べて動作が複雑な分, ユーザもその状況を把握しにくい. そこでユーザがデバッグ対象に集中できるよう, デバッグ情報は極力少ない出力量で役立つ情報を得られるのが望ましい. たとえばデバッグ時には, 動作の詳細を見るというよりむしろ, その述語の実行結果として得られるデータの内容のみに興味があることもしばしばである. このような場合, 述語トレースでは出力量が多いだけに, 読み取るのに時間がかかり不便である. そこで, リスナには変数モニタという機能があって, 必要な変数に焦点を当ててその変数の具体化状況だけを報告する.

A.　トレースとスパイ

普通の言語のデバッグでトレースというと, プログラムをステップ実行しながら実行過程を逐一表示する機能をいう. スパイは, プログラム中にブレイクポイントを設定して実行すると, そこまで実行が進んだ時点で一時中断してデバッガのトップレベルに戻る機能である. その状態で, その時点でのプログラムの進行状況をいろいろ調べることもできるし, そこからトレースを開始することもできる. スパイを用いると, デバッグ対象にたどり着くまでの不必要なトレースの手間を省くことができるのだ. KL1 のデバッグにおいてもこれらの機能はある意味では同じである. しかし, 逐次に実行される言語だと, プログラムのある部分が実行を終えるまで他の部分の実行は始まらない. したがって, プログラム中のある箇所にブレイクポイントを設定しておいてそこから適当な期間だけ実行内容をすべてトレースしてしまえば, 自然にプログラム中のデバッグしたい部分だけをトレースすることができた. しかし, KL1 ではゴールはすべて並列に, 複数が同時進行しているので, そのように期間を区切るだけでは同時に実行中の他の不必要なゴールまで一緒にトレー

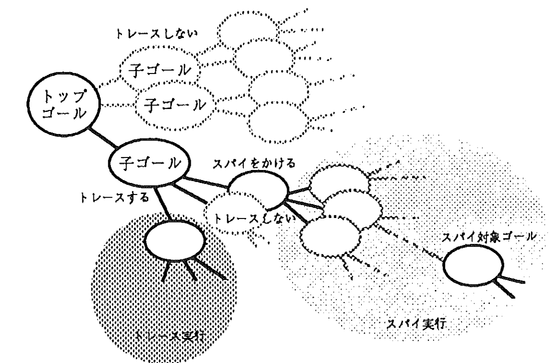


図 **5.4.3** トレース木

スされてしまう.

そこで PIMOS のトレーサは, ゴールの親子関係だけに注目してトレースする. つまり, あるゴールがリダクションされるたびにその結果生じた子ゴールを報告し, 必要ならばそれらの子ゴールのトレースを継続するというような具合である (図 5.4.3). このようにすると, たとえ複数のゴールが同時に動いていても必要なものだけをトレースすることができるので, デバッグ対象に集中することができる. スパイでも同様である. また, リスナにはトレース中にゴールの実行を一時的に保留しておいて好きな時点でその保留を解除する機能がある. これを用いるとユーザが計算の実行順序を制御することができるようになり, デバッグしやすくなることもある.

B.　トレースやスパイの仕組み

ここで, トレースやスパイの実現の仕組みについて簡単に触れておこう. リスナがユーザのゴールを実行するときには, それらの実行を監視したり操作したりしなければならないので, PIMOS のタスクの機能を使っている. タスクは KL1 の荘園機能を基にしたものである. トレースやスパイをする場合には, 荘園がもっているトレースやスパイ用の特別な機能を使う. 処理系がこれらの機能を提供していることで, PIMOS のデバッガでは高速にトレースやスパイを行なうことができるのである.

荘園内で, あるゴールをトレースしながら実行するには, トレース実行のために処理系で用意された特別なメタ組込み述語を実行する. この述語の引数には, ゴール自身と, トレーサがゴールを識別するために適

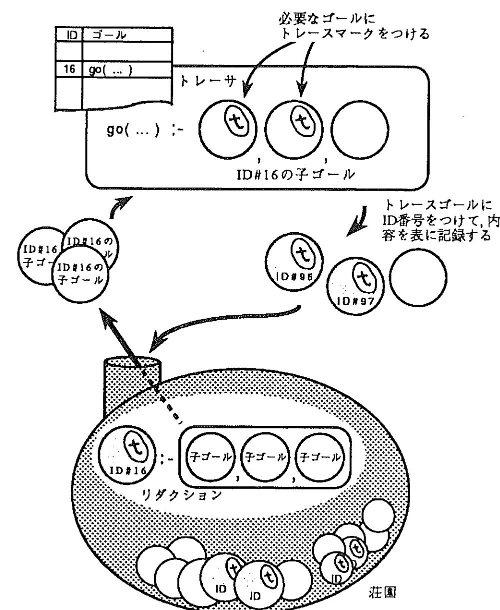


図 5.4.4 トレースの仕組み

当に決めたゴール識別子を与えるようになっていて，この述語を実行することで，指定したゴールにトレースマークがつく．

トレースマークのついたゴールがリダクションされると，そのゴールの識別子と子ゴールがトレース例外として荘園の報告ストリームを通じてトレーサに報告される．例外の報告形式には，例外処理後に実行すべき代替ゴールを指定するようになっている．トレーサは，子ゴールにも続けてトレースマークをつけるかどうかユーザに聞き，代替ゴールとして子ゴールをトレースマークをつけて (あるいはつけないで) 実行するよう指定する．ゴールをトレースなしで実行するには，一般のゴール実行用の組込み述語を用いればよい．途中でトレースをやめた計算の部分木については，もともとトレースなしで実行した場合とまったく同じ効率で実行することができるというのが利点である．

スパイについても同様である．ある述語にスパイをかけてゴールを実行するには，スパイ実行のためのメタ組込み述語を用いる．この述語の引数には，ゴール自身とそのゴール識別子，さらにスパイ対象を設定するようになっている．実行すると，ゴールとそのすべての子孫ゴールが，スパイ中というマークつきで実行される．そして，実行中にスパイ対象となっているゴールが生成された時点で，スパイ例外としてトレーサに報告されるのである．

C. デッドロックの検出

KL1 でゴールが実行を中断してしまうのは，実行を継続するのに必要なデータがまだ計算できていないときである．デッドロックとは，あるゴールの実行が中断し KL1 における通常の処理の範囲では決して再開されることがあり得なくなってしまった状態をいう．具体的には，どこからも参照されていない (参照パスがない) 変数の具体化をゴールが待っているとか，複数のプロセスが互いに相手の計算が進むのを待っている，といった状態である．永久中断状態ともいう．

ゴールが一つデッドロックすると，たいていそのゴールが生成するはずだったデータを待っているゴールが全部デッドロックしてしまうから，デッドロックの連鎖が起こってしまい，結局，大量のゴールがデッドロックしてしまう．このような場合，ユーザがデバッグ時に知りたいのは，これらの大量のゴール全部ではなくて，最初にデッドロックしたゴールである．このようなゴールを極大ゴールと呼ぶ．

処理系には，デッドロックしているゴールを発見し，それらのなかから極大ゴールのみを報告する機能がある．ほとんどの場合は局所 GC 時に発見するが，データの MRB 管理を用いることによってゴールがデッドロック状態となる瞬間に検知できる場合もある．リスナは，処理系のこれらの機能を用いてユーザにデッドロックを報告する．

D. デッドロック検出の仕組み

処理系がデッドロックに陥っているゴールを発見し，報告する仕組みについて簡単に紹介しよう．まず局所 GC 時に報告する場合の手順は，1) デッドロック状態になっていることを検出し，2) デッドロックしているゴールをすべて探し出し，3) それらのゴール群のうち，極大ゴールを抽出して報告する，という 3 段階に分けて行なう．

なぜ処理系が局所 GC 時にデッドロックの検出をするようにしているのかというと，通常の実行時にデッドロック状態に陥っていることに気づくのはとても難しいからである．処理系のなかでは，実行を中断しているゴールは中断原因となった変数からのみポインタで参照されるような構造になっている．つまり，ゴールが実行を中断するのは必要なデータがまだ具体化されず未定義変数となっているためだから，処理系では，この中断原因となった変数が具体化されたときにはじめて実行を中断しているゴールを発見し，実行可能になったことがわかるようになっているのである．

デッドロックの検出方法はこの実行中断メカニズムをうまく利用したもので，やり方はこうである．ゴールがデッドロックしているということは，すべての実行可能ゴールのどの引数データからも，そのゴールはまったく参照されていない，つまり到達不可能なはずである．局所 GC はコピーイング方式で行なっており，GC 時には，実行可能なゴールのみをルートとして必要なデータを旧領域から新領域へコピーしていくので，デッドロックしているゴールは最後までコピーされずに旧領域に残る．そこで，局所 GC の前後でゴール数が減っていればデッドロックゴールが存在することになる．

このようにしてデッドロックしているゴールがあることがわかると，次に旧領域を走査してそれらのゴールをすべて探し出す．ゴールには特別なマーク (タグ) がついていて，メモリ上に残っているとすぐわかるようになっている．そして，発見したデッドロックゴール群のなかから極大ゴールだけを抽出して報告する．

ところで，局所 GC 時にデッドロックゴールを検出する方法だと検出がかなり遅れてしまうこともある．それに，たとえ極大ゴールを発見できるとしても，本当の原因 (変数への具体化パスを捨ててしまったゴール) を特定するのが困難な場合もある．そこで，ゴールがデッドロック状態になる瞬間を処理系が検知した場合には，それを即刻報告することにしている．これにはデータの MRB 管理を用いる．つまり，ある変数への最後の参照パスだと判断できるものを消費してしまうような命令を実行すれば，関係するゴールがデッドロックすることは明白である．このように，事が発生した瞬間にユーザに報告することで，より詳細な情報を一緒に報告することができるので，プログラムのデバッグに大いに役立つ．

E. 変数モニタなど

並列プログラムでは，複数プロセスが同時に計算を進めることが多い．それらのいくつものプロセスに対してむやみに述語のトレースやスパイを行なうのでは，これらの出力情報が大量になってしまって本当のデバッグ対象が見にくくなり，結局デバッグの効率が悪くなってしまう．これから紹介する機能は，述語トレースやスパイによる大量の出力情報を大幅に減らし，ユーザが不必要な情報をなるべく見ないで済ませられるために大変よい手助けとなるものである．これらの機能は，トレースやスパイなどの機能とうまく組み合わせて用いることによって，さらに効率よくデバッグすることができるようになる．

まず，変数のモニタ機能．KL1 データを監視し，その具体化を待って報告する機能である．プログラムの実行が進むに従って未定義変数がどのように具体化されていくのか，状況を知りたい場合に利用する．KL1 プログラムの基本的なプログラミングスタイルの一つとして，プロセス間のストリーム通信によってメッセージをやり取りしながら計算を進めるような手法があるが，この場合，ストリームはプロセス間の共有変数である．したがって，ストリームを流れるメッセージを観察したければ，この共有変数をモニタすればよい．普通は，プロセスになっている述語をスパイしたりトレースするなどしておき，リダクションが報告された時点でストリーム変数のモニタを開始するというように使用する．その後，特に必要がなければトレースやスパイは中止してしまってモニタの報告だけを観察するようにすれば，余分なトレース出力に目をやる必要がなく，デバッグの対象に集中できるので便利である．

インスペクタは，複雑な構造体になっているような KL1 データの内容を調べるものである．ユーザがなるべくデータの不必要な部分を見なくて済むように，会話的にデータの一部分を調べて表示するようになっている．また，リスナには変数を管理する機構がある．ここにはゴールの引数に用いた変数やトレース中に現われた変数を保存しておき，それらのデータをあとの実行に利用することができる．

5.4.3　さらに，並列プログラムをチューニングするために

さて，プログラムが一通り正しく動くようになると，次の課題はチューニングである．プログラムの遅いところを速く効率よく動くようにするのである．各プロセッサが均等に忙しく仕事をし (高プロセッサ稼働率)，その結果，N 台のプロセッサで実行したら1台で実行するより N 倍速く処理を実行できるようにする (高プロセッサ台数効果) のがチューニングの大きな目標 (理想?) である．具体的には，プロセッサの忙しさに偏りがないか，ゴールが適切に各プロセッサに分散しているか，などを調べてプログラムを改良する．PIMは疎結合並列マシンだから，プロセッサ間通信のオーバヘッドが非常に大きい．効率のよいプログラムを作るには，プロセッサ間通信を最小限に抑えることも重要なポイントである．そこで具体的には，プロセッサ間にわたる通信量を調べる．

しかし，このようなことは当然，プログラムを単に実行するだけではわからない．そこで，プログラムの動作状況を詳しく調べてわかりやすく表示するツールが必要となる．PIMOS にはこのようなツールが2種類ある．一つはリアルタイムにプロセッサの稼働状況を表示するもの，もう一つはプログラムの実行時に計測した実行ログ情報を基に，プログラム実行後にプログラムの動作をさまざまな観点から解析してグラフに表示するものである．どちらの場合も処理系の機能を用いてデータを計測しているので，比較的小さなオーバヘッドで重要な情報を得ることができる．

A.　リアルタイムに表示する

パフォーマンスメータは，リアルタイムにプロセッサの稼働状況を表示するツールである．一定時間間隔ごとに各プロセッサの稼働状況を測定し，そこからプロセッサ稼働率などを算出してカラー (または白黒の濃淡) のグラフで表示する．測定項目は，各時間間隔 (T) において各プロセッサがプロセッサ間通信用メッセージの受信に要した時間 (A)，送信に要した時間 (B)，GC 時間 (C)，アイドル時間 (D) の4項目である[†1]．これらの計測値の和 $(A)+(B)+(C)+(D)$ を時間間隔 (T) から差し引くとプロセッサが実際に計算に要した時間 (E) が求められる．プロセッサ稼働率は，時間間隔 (T) のうち $(A)+(B)+(E)$ が占める割合をいう．

グラフには，プロセッサ稼働率をはじめとして，(A) から (E) の値 (のうち任意の物の合計) が時間間隔 (T) に占める割合を表示することができる．また，さらにそれらの値の平均を取って全プロセッサの平均値として表示することも可能だ．口絵4は，アミノ酸配列を解析するプログラムをプロセッサ総数256台の PIM 上で動作させたときのプロセッサ稼働率を表示した例である．縦軸がプロセッサ番号，横軸が時間を表わしており，時間経過とともに表示画面は左へスクロールしていく．桝目の色が赤いほどプロセッサ稼働率が高いことを示す．あるプロセッサで GC が発生すると三角形 (△) で表示されるようになっている．このプログラムは，全プロセッサにゴールを分割して[†2]並列に計算させ，全部の計算結果を比較して最良のものを次のサイクルの計算の初期値として用いる，という処理を反復して解を求めていく．グラフからは，ゴールが分散された時点で全プロセッサがほぼ同時に赤く (稼働率100%) なり，その後，計算結果が求まったプロセッサから順に紺色 (稼働率0%) になっていく様子がわかる．全プロセッサでの計算結果が出揃った時点で最良解が選択されると，再び全プロセッサで計算が再開され，次のサイクルが始まったことがわかる．

B.　グラフにまとめる

パフォーマンスメータは，プログラムの実行時に直観的に動作状況を見るのには便利だが，実際のチューニング段階ではもっと詳細で正確な情報が必要である．たとえば，ある時点であるプロセッサだけが稼働率が高かったなら，それが何の処理を (どのゴールを実行) しているためなのか，などである．そこでプログラムの実行時に，実行に関する詳細なログ情報を取っておき，実行終了後にこれをさまざまな観点から解析してグラフにまとめて表示するツールも用意されている．Parallel + Graphic という意味で ParaGraph という名前である．

ParaGraph で取れるログ情報は3種類あって，リスナで実行を開始するときにどの情報を取るかを指定する．(a) プログラムのゴール実行状況，(b) プロセッサ間通信の発生状況，(c) プロセッサの稼働状況，の三つ

†1　これらの値を計測することによる実行時のオーバヘッドは5%以下である．

†2　この例では256番目のプロセッサだけは使用していない．

である．ParaGraph はこれらの情報を「何が」，「いつ」，「どこで」，「どれくらい」処理されたかに着目して表示する．逐次プログラムの実行だったら「どこで」は常に一定だから必要ないし，「いつ」は実行順序がプログラム中に明記してあるのだからほとんど重要ではないのだが，並列プログラムの動作状況を知るためにはこれらの次元の情報を欠かすことはできない．ここでは，「どこで」はその事象が起こったプロセッサをプロセッサ番号で表わしたものであり，「いつ」は一定の時間間隔で区切ったうちのどの期間であるかで表わす．時間間隔の長さはユーザが自由に設定できる．「何が」は計測項目を示している．たとえば，(a) プログラムのゴール実行状況を指定して実行すると，いつどのプロセッサでどのような述語がどれだけリダクションしたか，サスペンションしたかが計測される．つまり，この場合の「何が」は述語ごとのリダクション数，サスペンション数である．(b) プロセッサ間通信の発生状況を指定して実行した場合の「何が」は，プロセッサ間通信用メッセージの種類ごとの送受信回数であり，(c) プロセッサの稼働状況を指定して実行した場合の「何が」は，計算時間，プロセッサ間通信のためのメッセージの送受信処理時間，GC 時間，アイドル時間が各時間間隔に占める割合である．

グラフには，何がいつどれくらい処理されたか，または，何がどこで，いつどこで，などの表示形式があり，ユーザは，取ったログ情報をさまざまな方向から眺めることができるようになっている．また，「何が」についてはすべてを表示せず，ユーザが必要なものだけに絞って表示するといった機能や，ログ情報をファイルに保存したり，ファイルから再度ロードするといった機能もある．口絵5は，さきほどのアミノ酸配列を解析するプログラムを計測実行した場合のグラフの表示例である．

ところで，計測の実現方法について簡単に触れると，(a) プログラムの実行状況を計測する場合には，プロファイル用の特別仕立ての荘園を用いる[†1]．この荘園内で述語が実行されると，実行されたプロセッサ番号，各述語が実行を開始 (再開) した回数，実行を中断した回数が，述語コードごとに適当なタイミングで処理系から報告される．ParaGraph は，各述語が実行を中断した回数をサスペンション数，実行を開始 (再開) した回数からサスペンション数を差し引いたものを実際のリダクション数として算出し，これらのデータを集計してログ情報を生成する．(b) プロセッサ間通信の発生状況や (c) プロセッサの稼働状況を計測する場合は，パフォーマンスメータが用いているものと同様の処理系の機能を使っている．

5.4.4　ユーティリティの提供

以上が PIMOS のプログラム開発環境の全容である．最後にちょっとつけ加えると，PIMOS には，ユーザが共通に気軽にプログラムの一部として使用できるユーティリティがいくつか用意されている．たとえば，任意の KL1 データを一意に比較するものとか，ハッシュ関数とか，ソータとか，データを一時的に格納しておいて必要に応じて探索のキーを用いて取り出したり，参照したりするようなプールなどである．また，負荷分散のためのライブラリは，ユーザがプログラムの実行中に分散させたいゴールを，適当に暇だと思われるプロセッサに割りつけるもので，その割りつけ方にはいくつかの手法がある．実際，非常に多くのユーザはプールや負荷分散ライブラリに頼っており，これらもまた，プログラム開発に一役買っているものといえるだろう．

†1　なお，プロファイル用荘園を用いて計測した場合，5割増し程度の実行時間を要すると見られている．

第6章

むすび

第五世代コンピュータの並列処理技術のなかから，言語，ハードウェア，言語処理系，OSといった計算機システムを実現するための技術について解説してきた．

第五世代コンピュータは，近未来の知識情報処理を指向することから，そこで研究開発される並列処理技術については，従来は難しさのためにあまり真剣に取り組まれることのなかった並列処理の新しいジャンルを扱うものとなった．すなわち，動的に変化する計算や均質さの低い計算負荷を発生する大規模問題の並列処理である．

この種の問題を扱うには，問題を並列実行可能なプログラムとして記述することや，負荷バランスをうまく取って並列処理性能を引き出すことなど，難しい問題が山積している．それをある程度可能とするのに最も貢献したのは，何といってもKL1言語であろう．そしてKL1プログラムの効率のよい実行と良好な実行環境，開発環境を実現したのが，KL1言語処理系と並列OSのPIMOSだったといえる．

一方ハードウェアについては，プロセッサのなかにKL1の高速実行に必要な記号処理向き機能を取り込んでおり，並列処理にかかわる部分では，最近になって標準方式として認められだした各種の機能を先取りして実現したといえる．すなわち，ハードウェアに関しては，第五世代コンピュータはもはや特殊なシステムではなく，これからの並列処理ハードウェアの標準となってくる技術を数多く試したといえる．

ICOTの渕所長（現東京大学工学部教授）がかつて何度も繰り返し示したFGCSプロジェクトの作業仮説がある．「論理型言語を研究開発の基本において，計算機技術のあらゆるものをその上に再構築することにより，従来技術の多くの問題が解消されるとともにまったく新しい可能性が見えてくるに違いない．」FGCSプロジェクトで，言語，ハードウェア，言語処理系，OSといった非常に多方面の研究開発を総合的に，しかもすべてゼロから進めた理由は，この仮説を証明したかったからだともいえる．

それがなし得たかどうかについて，まだ客観的な評価が定まっているわけではないが，少なくとも仮説の何割かは証明されたのではないかと考えている．すなわち，従来扱っていなかったような並列処理の新しいジャンルについて，プログラム記述と効率のよい並列実行が可能となってきたわけであるが，その記述を可能にしたのがKL1言語であり，効率のよい実行と使い勝手を実現したのがKL1処理系とPIM，そしてPIMOSだったといえる．

それではKL1言語やその実行環境によって，先に述べた新しい並列処理のジャンルが十分開拓されたかというと，これはまだまだ不十分であって，多様な問題における並列プログラミングの技術や，負荷分散の技術など，それを問題対応で書いていろいろ試すための道具だてがようやくそろった段階ということができるだろう．

以前に，「並列ソフトウェア技術の現状は，過去の汎用マシンの歴史でたとえるならば，1950年代の終わりから1960年代初頭の技術レベルといえる．言語も未熟であり，OSも整備されておらず，利用者が問題を解くためのプログラムだけでなく，メモリや入出力

の管理やスケジューリングまでこと細かく心配しなければならない」といったことを書いたことがある[37]．確かにまだそれに近い技術レベルではある．それでも，KL1言語とその処理系とPIMOSのおかげで，並列プログラム作りと実験がずいぶん容易になり，未熟な並列ソフトウェアの技術は着実に前進を始めたように感じられる．

並列ソフトウェア技術のなかでも，最終的にはOSにやってもらいたいのが負荷分散である．けれども現状では，どのようにOSに制御させるのがよいかわからないので，仕方なくユーザがケースバイケースで，プログラムの工夫を重ねている．並列プログラミングと応用開発のなかで行なわれているそのような努力に関しては次編で紹介する．

第 II 編　参考文献

[1] Ueda, K. : Guarded Horn Clauses, ICOT TR-103, ICOT (1985)

[2] Chikayama, T., Sato, H. and Miyazaki, T. : Overview of the parallel inference machine operating system (PIMOS), *Proc. FGCS'88 : International Conference on Fifth Generation Computer Systems*, pp.230–251, Tokyo, Japan (Nov.28 – Dec.2, 1988)

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

[4] Shapiro, E. : A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003, ICOT (1983)

[5] Clark, K. L. and Gregory, S. : Parlog: A parallel logic programming language, *ACM Transaction on Programming Languages and Systems*, 8(1) (1986)

[6] Shapiro, E. and Takeuchi, A. : Object oriented programming in Concurrent Prolog, ICOT TR-004, ICOT (1983), also in *New Generation Computing*, Vol.1, No.1, Springer-Verlag (1983)

[7] Kumon, K., Asato, A., Arai, S., Shinogi, T., Hattori, A., Hatazawa, H. and Hirano, K. : Architecture and Implementation of PIM/p, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems*, (1992)

[8] Nakashima, H., Takeda, Y., Nakajima, K., Andou, H. and Furutani, K. : A Pipelined Microprocessor for Logic Programming Languages, *Proc. 1990 Intl. Conf. on Computer Design*, pp.355–359 (Sept.1990)

[9] Nakashima, H., Nakajima, K., Kondo, S., Takeda, Y., Inamura, Y., Onishi, S. and Masuda, K. : Architecture and Implementation of PIM/m, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems* (1992)

[10] Nakagawa, T., Ido, N., Tarui, T., Asaie, M. and Sugie, M. : Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c, *Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems* (1992)

[11] Sakai, H., Nakase, A., Takewaki, T. and Asano, S. : Applying Inter-cluster Shared Memory Architecture to a Parallel Inference Machines, *Parallel and Distributed Computing in Engineering Systems*, Elsevier Science Publishers (1991)

[12] 佐藤 正俊, 大原 輝彦, 武田 浩一 : 並列推論マシン PIM/i プロセッサの設計, 情報処理学会論文誌, Vol.33, No.3, pp.278–287 (1992)

[13] Nakashima, H. and Nakajima, K. : Hardware Architecture of the Sequential Inference Machine : PSI-II, *Proc. SLP'87 : Symposium on Logic Programming*, pp.104–113 (Sept.1987)

[14] Shinogi, T., Kumon, K., Hattori, A., Goto, A., Kimura, Y. and Chikayama, T. : Macro-call Instruction for the Efficient KL1 Implementation on PIM, *Proc. FGCS'88 : International Conference on Fifth Generation Computer Systems*, pp.953–961 (1988)

[15] Chikayama, T. and Kimura, Y. : Multiple Reference Management in Flat GHC, *Proc. ICLP'87 : International Conference on Logic Programming*, pp.276–293 (1987)

[16] Goto, A., Matsumoto, A. and Tick, E. : Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures, *Proc. ISCA'89 : 16th Annual International Symposium on Computer Architecture*, pp.25–33, Jerusalem, Israel (1989)

[17] Matsumoto, A., Nakagawa, T., Sato, M., Nishida, K. and Goto, A. : Locally Parallel Cache Design Based on KL1 Memory Access Characteristics, ICOT TR-327, ICOT (1987)

[18] Nishida, K., Kimura, Y., Matsumoto, A. and Goto, A. : Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures, *Proc. ICLP'90 : Seventh International Conference on Logic Programming*, pp.83–95 (1990)

[19] Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H. : An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems, *Proc. ICPP'88 : International Conference on Parallel Processing*, Vol.I, pp.18–22, (August 1988)

[20] 大西 諭, 稲村 雄 : KL1 における永久中断ゴールの検出と報告, 情報処理学会 計算機アーキテクチャ研究会, 82-1 (1990-4)

[21] Inamura, Y. and Onishi, S. : A Detection Algorithm of Perpetual Suspension in KL1, *Proc. ICLP'90 : Seventh International Conference on Logic Programming* (1990)

[22] Warren, D. H. D. : An Abstract Prolog Instruction Set, Technical Report 309, Artificial Intelligence Center, SRI International (1983)

[23] 近山 隆, 木村 康則 : ポインタタグ (MRB) による多重参照方式, 情報処理学会 第 35 回全国大会, 2Q-5 (1987)

[24] 木村 康則, 近山 隆 : MRB による多重参照管理方式 — KL1 処理系における実時間ガーベジコレクション方式 —, 情報処理学会 第 35 回全国大会, 2Q-6 (1987)

[25] 宮内 信仁, 木村 康則, 近山 隆, 久門 耕一 : MRB による多重参照管理方式 — KL1 処理系における一括型 GC の特性評価 —, 情報処理学会 第 35 回全国大会, 2Q-7 (1987)

[26] 西田 健次, 松本 明, 木村 康則, 後藤 厚宏 : MRB による多重参照管理方式 — KL1 処理系におけるキャッシュ特性の評価 —, 情報処理学会 第 35 回全国大会, 2Q-8 (1987)

[27] Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Nakajima, K. : Optimization Techiniques Using the MRB and Their Evaluation on the Multi-PSI/V2, *Proc. NACLP'89 : North American Conference on Logic Programming*, pp.907–921 (1989)

[28] 市吉 伸行, 六沢 一昭, 近山 隆, 中島 克人, 宮崎 敏彦, 杉野 栄二 : 並列処理における重みつき参照カウントを用いた実時間 GC, 情報処理学会 第 36 回全国大会, 7H-3 (1988)

[29] Ichiyoshi, N., Rokusawa, K., Nakajima, K. and Inamura, Y. : A New External Reference Management and Distributed Unification for KL1, *Proc. FGCS'88 : International Conference on Fifth Generation Computer Systems*, pp.904–913, (1988)

[30] Ichiyoshi, N., Rokusawa, K., Nakajima, K. and Inamura, Y. : A New External Reference Management and Distributed Unification for KL1, *New Generation Computing*, pp.159–177 (1990)

[31] 六沢 一昭, 市吉 伸行, 近山 隆, 中島 浩 : 分散環境におけるユニフィケーションの実現, 情報処理学会 第 37 回全国大会, 7Y-4 (1988)

[32] 六沢 一昭, 市吉 伸行, 瀧 和男, 吉田 かおる, 稲村 雄, 中島 浩 : 並列処理における PE 間に渡るゴールの重みつき参照カウントを用いた管理方式, 情報処理学会 第 36 回全国大会, 7H-2 (1988)

[33] Watson, P. and Watson, I. : An efficient garbage collection scheme for parallel computer architectures, *Proc. PARLE : Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, pp.432–443 (1987)

[34] Bevan, D. I. : Distributed garbage collection using reference counting, *Parallel Computing*, Vol.9, No.2, pp.179–192 (1989)

[35] Tel, G. and Mattern, F. : The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes, *Proc. PARLE : Parallel Architectures and Languages Europe*, LNCS 505, Vol.I, pp.137–149 (1991)

[36] Mattern, F. : Global quiescence detection based on credit distribution and recovery, *Inf. Proc. Lett.*, Vol.30, No.4, pp.195–200 (1989)

[37] 瀧 和男 : 大規模汎用並列処理の実現に向けて — ICOT における研究より, *bit*, Vol.23, No.2, pp.157–171 (1991)