JAIST Repository
https://dspace.jaist.ac.jp/

Title	Parallel TRAMのごみ集めの並列化に関する研究
Author(s)	斉藤，嗣治
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1170
Rights	
Description	Supervisor:二木 厚吉，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

Parallel TRAM のごみ集めの並列化に関する研究

指導教官　二木厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

斉藤 嗣治

1998 年 2 月 13 日

Copyright © 1998 by Tsuguharu Saitou

要 旨

本稿では，項書換えシステム TRAM を並列簡約を行うように拡張された Parallel TRAM を対象にし，ごみ集めに関して並列化することを試みる．

Parallel TRAM ではごみ集めの際にグローバルに同期をとり，かつ，ごみ集めを行う collector は唯一つに限られるというオーバヘッドがあることが指摘されている．そこで本研究では，Parallel TRAM にローカルごみ集めを導入することを考える．この際処理ユニットをまたぐ参照に対しては外部参照テーブルを用いることで，これに対処する．また，非常に重い処理である排他制御のためのロックに関して，これを減らすための工夫も行い効率の向上を目指した．

目次

1 はじめに **1**

1.1 本研究の目的 1

1.2 本研究の背景 1

1.3 本論文の構成 2

2 項書換えシステムについて **3**

2.1 項書換えシステム 3

2.2 関連研究 5

3 ごみ集めについて **7**

3.1 ごみ集め 7

3.2 ごみ集めの方式 8

3.2.1 リファレンスカウントによるごみ集め 9

3.2.2 トレースによるごみ集め 11

3.2.3 その他のごみ集め 13

3.3 並列ごみ集め 14

3.3.1 グローバルごみ集めとローカルごみ集め 14

3.3.2 外部参照の実現 15

3.3.3 リファレンスカウントによる並列ごみ集め 15

3.3.4 トレースによる並列ごみ集め 16

4 項書換え抽象機械 **18**

4.1 TRAM 18

4.1.1 TRAM の概要 18

4.1.2 E-戦略 18
4.1.3 書換え規則と入力項のコンパイル 20
4.1.4 マッチングプログラムの構成 20
4.1.5 抽象命令のインタプリタ 22
4.2 Parallel TRAM 23
4.2.1 Parallel TRAM の構成 23
4.2.2 Parallel TRAM のメモリ管理 23
4.2.3 並列 E-戦略 24
4.2.4 Parallel TRAM の戦略リスト 24
4.2.5 Parallel TRAM の同期処理とロック機構 25
4.2.6 プロセス処理ユニットの管理 27

5 ごみ集めの並列化 34
5.1 Parallel TRAM におけるごみ集め 34
5.2 並列ごみ集めの設計 35
5.2.1 メモリ管理 35
5.2.2 並列ごみ集め 35
5.2.3 コピー方式の並列ごみ集め 36
5.2.4 外部参照テーブル 36
5.2.5 外部参照テーブルの排他制御 37

6 実験と評価 42
6.1 実験 42
6.2 評価 43
6.3 考察 43
6.3.1 Parallel TRAM の性能 43
6.3.2 ごみ集めのよるオーバヘッドの計測 44
6.3.3 並列ごみ集めの性能 44

7 まとめ 47
7.1 並列ごみ集め 47
7.2 今後の課題 48

謝辞 **50**

参考文献 **51**

付録 **53**

付録 1.　フィボナッチ数列 53
付録 2.　ソート 54
付録 3.　100×100 の計算 56

図 目 次

3.1 リファレンスカウントと循環リスト 10
3.2 世代別ごみ集めと殿堂入り 12

4.1 TRAM の構成 19
4.2 マッチングプログラムの構成 21
4.3 マッチングプログラムのフラグメンテーション 22
4.4 並列 E-戦略の構文 24
4.5 ロックの仕組み 26
4.6 Parallel TRAM の構成 28
4.7 CODE 領域の構成 29
4.8 マッチングプログラムにおけるメモリフラグメンテーション 30
4.9 FORK の定義 31
4.10 WAIT の定義 32
4.11 EXIT の定義 33

5.1 メモリ構成 40
5.2 外部参照テーブル 41

表 目 次

6.1 計測結果 46

第 1 章

はじめに

1.1　本研究の目的

項書換えシステムは代数仕様言語や定理証明などの実装で非常に多く用いられている．さらに計算機へ比較的容易に実装することが可能である．しかし，計算機への実装の容易さに比べ実行効率は一般に悪く，実用になる程の項書換えシステムを得るためにさまざまな工夫や最適化が必要となる．そこで本稿ではすでに弁別ネットをはじめ，さまざまな要素技術が盛り込まれている TRAM を並列簡約を行えるように拡張された Parallel TRAM のごみ集めについて焦点をあて，これを並列化することを試みる．

1.2　本研究の背景

項書換えシステムは，代数仕様言語，関数型言語，等式論理の証明などに幅広く応用できる計算モデルである．等式を左辺から右辺への書換え規則とみなすことによって，元来計算するという意味を持たないはずの等式を計算に用いるという考え方が基本になっている．このような考え方によって論理の世界と計算の世界を結びつけることができ，プログラムの検証や変換というような論理と計算の両方を用いる問題に対し有効に働くことができるモデルとなっている．また，実装という観点から見ても計算機との相性は非常によく，比較的容易に行うことができる．しかしながら，このような利点とは対照的に実際に実行するとその効率はあまりよいものではない．そのため，実用にするためには数々の工夫が必要となる．本稿では，簡略化戦略に E-戦略を採用し，パターンマッチ用に弁別ネッ

トを用いるといった特徴をもつ抽象機械 TRAM をさらに並列簡約が出来るように拡張し共有メモリ型マルチプロセッサに実装した Parallel TRAM を対象とする．また Parallel TRAM の大きなボトルネックとしてごみ集めがあげられている．これは，Parallel TRAM がごみ集め時にグローバルな同期を必要とするということとごみ集めでは唯一つのプロセッサのみしか働かないということが大きな要因となっている．そこで，本稿では並列ごみ集めの考えを採り入れ，Parallel TRAM に並列ごみ集めを導入することを試みる．

1.3 本論文の構成

本論文の構成は以下の通りである．まず 2 章で，項書換えシステムについての基本概念や関連研究についての説明を行う．3 章は，ごみ集めについて，その種類をいくつかに分け，それぞれについて特徴や関連研究についての説明を行っていく．4 章は，項書換え抽象機械である TRAM とそれを並列拡張した Parallel TRAM についての説明を行っていく．5 章で，ごみ集めの並列化に関して設計を行っていく．6 章は，実装した新しい Parallel TRAM についてその性能の評価を行っていく．

第 2 章

項書換えシステムについて

2.1　項書換えシステム

項書換えシステムとは項の書換えを計算の基本とした計算モデルで，項の集合と項を書換えるための書換え規則の集合の対で定義される．

項の定義は以下のとおりである．

1. 定数記号および変数記号は項である．
2. $t_1, t_2, \ldots, t_n$が項で，階数を n とする関数を fとすると $f(t_1, t_2, \ldots, t_n)$ も項である．

定数： 定数とは，階数が 0 である変数記号である．また，変数を含まない項を定数項と呼ぶ．

書換え規則： 以下の条件を満たすとき s と t の対 ($s \to t$ と書く) を書換え規則という．

1. s は変数記号ではない．
2. t に出現している変数記号は s にも出現していなければならない．

ここで，s は左辺，t は右辺と呼ばれる．

項書換えシステムでは，与えられた項を書換え規則に基づいて書換えていき，それ以上書換えられなくなった項をもとの項に対する計算結果をして得ることができる．

ここで，＋の定義として以下の等式が与えられたとする．

x+0=x

x+s(y)=s(x+y)

これを書換え規則とみなして 2+1 を計算すると，s(s(0))+s(0)→s(s(s(0))+0)→s(s(s(0))) となり結果 3 が得られる．

項の照合： 項 t, t' について，t にある適当な項を代入すると t' に一致するようなとき，t は t' に照合するという．この際，代入 σ は，変数記号の集合から項の集合への写像として定められる．また，パターンマッチとも呼ばれる．

項の簡約： 項 u が書換え規則 $l \to r$ の左辺 l と代入 σ によって照合するような項 t を部分項として含むとき u の部分項 t を σr へ置き換えて得られる項を v とする．このとき，u は v に**簡約**または**書換えられる**という．また，定数項およびその部分項が書換え可能なとき，一回以上の書換えを行うことを簡約化という．

リデックス： 項書換え系 R において，書換え規則の左辺に対し照合するような項を R のリデックスという．

正規形： 部分項に一つもリデックスを含まない項を正規形または既約という．つまりそれ以上書換えることの出来ない項である．

正規形を元の項に対する計算結果とみると，項書換え系は項を受け取ってその正規形を返すような計算機構と考えることができる．計算結果が存在することを保証するのは停止性と呼ばれる性質で，求まった計算結果が一意であることを保証するのは合流性と呼ばれる性質である．

停止性： 停止性とは停止して正規形に至る簡約系列つまり書換えの列が必ず存在することである．

合流性： 合流性とは，与えられた項に対し異なる簡約系列が存在してもその正規形は

必ず一致することである．

停止性と合流性は項書換え系システムの基本的な性質であるといえる．

重なり： 適当な定数項 c が存在してある書換え規則の簡約項が他の書換え規則の可簡約項の部分項となるとき，この二つの書換え規則は重なりを持つという．

最外演算子： 項の一番外側の演算子を最外演算子とよぶ．

書換え戦略： 項の簡約を行う際に，簡約すべき順序を書換え戦略という．代表的な書換え戦略としては，最も左側で最も外側に出現するリデックスから書換えていく戦略である**最左最外戦略**や，最も左側で最も内側に出現するリデックスから書換えていく戦略である**最左最内戦略**などがあげられる．また，TRAM で用いられる戦略は E-戦略と呼ばれるもので，戦略をユーザが指定できるというものである．

2.2 関連研究

項書換えシステムは，色々な分野への応用が可能であるので，理論・実装の両方で盛んに研究がなされている分野の一つである．特に項書換えの処理系を抽象機械という形で設計するという研究は，Kamperman らによる ARM(Abstract Rewriting Machine) を代表に数多くの研究がなされている．この ARM はわずか四種類の命令と一つのヒープ領域，二つのスタック領域からなる項書換え抽象機械である．この ARM の抽象命令は C 言語に変換されて実行されるため非常に効率の良い処理系となっているという特徴がある．

また，もともとシングルプロセッサを対象にしたソフトウェアを並列処理できるように拡張するといった研究も数多くなされている．R. H. Halstead による Multilisp[7] などは非常に有名で，このなかで並列処理を実現するために考えられた「future」という概念はその後の並列・並行システムなどで普通に使われるようになった重要な概念である．Multilisp ではこの future という概念によって驚く程の多くの並列性の抽出が可能であることを示した．

また，項書換えシステムを並列に処理するという研究も今までに多く行われた研究分野である．その例としては Peyton J. の実装した GRIP などがあげられる．この論文では

並列簡約を効率良く行わせるためにはマシンの負荷分散が重要であると述べている．このGRIPというシステムは関数型言語の処理系として実装されたグラフリダクションシステムであり，このような高い並列性を示すグラフリダクションのような処理では並列の粒度をあらくし，うまく負荷分散させることが重要であると述べている．最も理想的なのは簡約処理の分配を行うスケジューラを動的に切替えることであるが，この処理ではスケジューラの切替え自体がオーバヘッドとなり単純なスケジューラでも十分な効果が得られると結論付けている．

第 3 章

ごみ集めについて

3.1 ごみ集め

動的記憶管理を行っているなどで，明示的に領域を解放できないような場合，使用されなくなったセルは未使用なまま放置されることになる．このように放置されているセル(ごみ) を再利用するために回収することをごみ集めと呼ぶ．このような処理を行うことにより，メモリといった限りある資源を再び有効に活用することが出来るようになる．

まず，ごみ集めで用いられる用語の解説を行う．

- mutator：プログラムはセルを生成したり他のセルへの参照を捨てたり移動したりする．このような変化を mutation と呼びこの変化を引き起こすプログラムの通常計算のプロセスを mutator と呼ぶ．この用語は on the fly ごみ集め [3] から使われ始めた．

- collector：mutator に対しごみを集める計算のプロセスを collector と呼ぶ．これは，ごみとそうでないものとを識別するという動作と，ごみセルの回収という 2 つの動作を含む．

- コピー方式ごみ集め：記憶領域を 2 分しリスト処理には一方のみを用いる．空きセルが無くなった時点でリスト処理の実行が中断され使用しているセルを他方へ複写する．もともと使用していた領域をすべて破棄し次は新たな領域として使用する．

- 並列ごみ集め：一般に並列ごみ集めと言った場合，collector を複数にして，ごみ集

めの処理を並列に実行させるものと，collector と mutator が同時に存在し，それらが並列に実行されるようなごみ集めの２つがある．

- ルート：プログラムはオブジェクトを利用する際，ルートと呼ばれる参照ポインタの集合からたどることにより利用する．ルートから参照していくことによりたどり着くことの出来るオブジェクトは reachable と呼ばれ，たどることの出来ないオブジェクトは unreachable と呼ばれる．

3.2 ごみ集めの方式

ごみ集めは OS や多くの言語処理系，例えば Lisp に代表される記号処理言語の処理系など非常に多くで用いられる重要な技術である．さらに，効率の良いシステムを実装するのはあまり容易では無いため，古くから研究されている分野でもある．しかし，並列ごみ集めなどマルチプロセッサを前提にしている研究は比較的最近になってからの研究で，現在もさまざまな研究が行われている．

ごみ集めについての研究には一般に以下のようなものが対象となっている．

- ごみ集めの全所要時間の短縮
- ごみ集めによる mutation の中断の短縮
- セルの寿命や参照関係の統計的解析の利用
- プログラムの静的解析によりコンパイル時にごみの回収を行う
- ページングによる仮想記憶の性質の利用やキャッシュによる効率の向上

また，ごみ集めのアルゴリズムはすでに多くのものが考えられており，それを利用するためには以下のような条件を考慮して選択，利用する必要がある．

- セルの長さが固定か可変か
- ユーザ定義のセルの有無
- 仮想記憶の利用の有無

- ごみ集めによる停止時間の許容の有無
- ポインタ使用の有無
- セルの寿命や参照関係に傾向や規則が見出せるか
- マルチプロセッサにおいて共有メモリか分散メモリか

では，ごみ集めのアルゴリズムについて述べていく．ごみ集めのアルゴリズムとしては，大きく以下の２つに大別できる．

- リファレンスカウントによる方法
- トレースによる方法

それぞれについて説明を行っていく．

3.2.1　リファレンスカウントによるごみ集め

一般にリスト構造は，単純な「木」や木の集まりである「林」であることは少ない．木のより葉に近いところからより根に近いセルを参照しているようなことも多々起きる．このような状態では一つのセルが複数から参照されているという多重参照が起こる．ごみ集めでは，この多重参照の存在のため回収してよいセルとまだ使われているセルとの区別が難しくなるのである．そこでリファレンスカウンタと呼ばれる自身のセルの参照回数を数える機構を用意し，この値が 0 になることによってごみであると判別するのがリファレンスカウントによるごみ集めと呼ばれる方法である．この方法はごみになった時点で即座に回収出来るという利点を持つ．また，ごみの回収に必要な処理をプログラム全体に分散出来るという点から実時間処理などに向いていると考えられる．しかし，この方法では本質的に循環リストの回収ができないという欠点を持つ (図 3.1)．さらに，ごみ集めのプログラムが分散してしまい見通しが悪くなり，また特に実時間処理などにおいてセルへの参照の度にカウンタの増減が必要であるという大きなオーバヘッドがあるため，プログラム全体の処理時間が長くなるという欠点もある．また，セルの大きさが不定な場合や仮想記憶空間内でセルの配置の局所性が問題になる場合などは，さらに詰め替え (compaction) といった作業が必要になる．さらには，各セル内にリファレンスカウントのための領域が

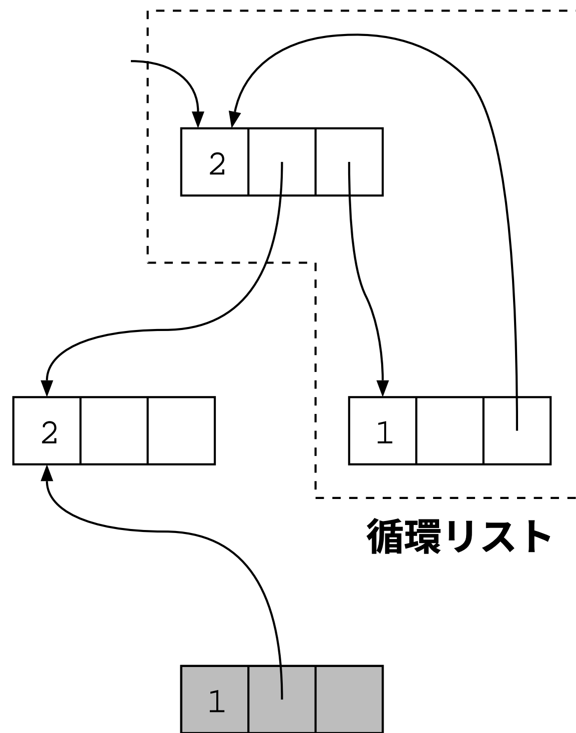

図 3.1: リファレンスカウントと循環リスト

必要となる．以上の多くの欠点のためこのままでは実用的で無くなっているという事が言える．

そこで，これを改良したいくつかの方法が考えられている．

Bobrow のテクニック

通常のリファレンスカウントでは循環リストを回収することが出来ないと述べたが，この方法を用いることによってリファレンスカウントを用いても循環リストが回収できるようになる．これはプログラムによって分けられたグループ内にセルを配置し，参照回数をカ

ウントする際にグループ内か外かで区別するという方法である．しかしながら，この方法では，セルの参照の度にグループの判別を行わなければならず，オーバーヘッドの増加は否めない．

3.2.2　トレースによるごみ集め

トレースによるごみ集めとは，ルートからセルをたどっていき，いきついたセルを生存セルとみなしごみかどうかを判別してごみを集める方法である．これには大きく分けて２つの方法がある．また，このどちらの方法もリファレンスカウントによる方法と異なり循環構造のごみも回収することができる．

マーク・スイープ方式

これは，ルートからセルをたどっていく際に，たどり着いたセルに印をつけ，すべての生存セルに印をつけた後にセル領域全体を走査して印の無いセル，すなわちごみを回収するという方法である．回収する方法として生存セルを移動させずにリストでつないでいく方法と，生存セルをセル領域の端に詰めていくという方法などがある．マーク・スイープ方式では基本的にマークとスイープという２つの動作で２度同じセル領域を走査するという手間がかかる．後者は，マーク・コピー方式と呼ばれる方式で，セル領域のフラグメンテーションを回避できるという利点があるが，セルの位置が変化するという問題や，そもそもコピーをするために作業量が増えるという問題がある．
マーク・スイープに関する研究としては，ごみ集め時にマーク動作のみを行いスイープ動作は新しいセルの割り当て時に行うといった研究やセルの生成順序が保存されるという性質を用いてセル領域の走査回数を減らすといった研究もある．

コピー方式

コピー方式のごみ集めでは，セルの生成領域全体を連続する二つの部分空間に分けて使用する．通常，mutator はそのうちの片方しか使用しない．セルはこの片方 (from space と呼ぶ) を先頭から割り当てられていく．from 空間から十分な空きがなくなったときに mutator の実行が中止されごみ集めが行われる．このごみ集めは，使用中のセルを先ほど使われなかったもう片方の部分空間 (to space と呼ぶ) にコピーを行うことでなされる．この際，ルートからセルをたどっていくという動作自体はマーク・スイープと同じであるが，

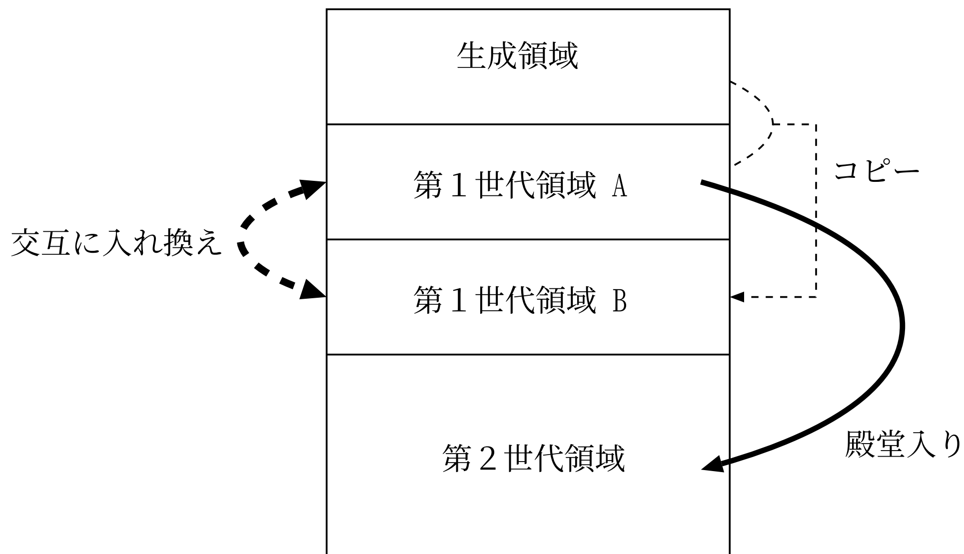

図 3.2: 世代別ごみ集めと殿堂入り

たどったセルに印をつけるのではなく別の領域に生存セルをコピーしていく．これを，すべてのたどれる (生存) セルに対して行えば，いままで使っていた from space 全体が再び利用できるという仕組みである．コピー方式の特徴としては，生存セルの量に比例した時間で処理が出来るという事と，ごみ集め後にセルが隣接するために局所性が上がるという特徴がある．また，欠点としては，メモリ領域を２つ以上に分割して使用するので一度に利用できるセル領域が減るということや，すべてのセルが移動するためにアドレスが変化するといった事があげられる．

世代別ごみ集め

コピー方式のごみ集めでは，生きているセルの数が多ければ多いほどそれらをコピーしなければならず大きな負担となるが，少なければ負荷が小さくて済むという性質を持つ．また，セルに注目すると以下のような性質をもつ事が分かる．

1. セルは新しくつくられたものほどごみになりやすく，生き残ったものほどさらに生き残りやすい．

2. ポインタの方向を考えるとできてまもないセルから長く生き残っているセルに向かうものが多い．

１番に関しては生き残ったセルほどコピーされる回数が多くなるのが分かる．このような性質に基づいて設計されたのが世代別ごみ集めと呼ばれるものである．この方式では，セルを生成されてからの時間 (年齢と呼ぶ) によっていくつかの世代に分ける．セルが生成される領域をいくつかに区切りそれぞれに対しひとつの世代を割り当てる．ごみ集めは最も若い世代に対しコピー方式のごみ集めが行われる．一度コピーされたセルは一度生き残ったと判定される．そして，何回か生き残ると一つ上の世代とみなされ領域を移動する．これを殿堂入りという (図 3.2)．この方式は有望なごみ集め方式とみられ，Smalltalk や SML/NJ など多くの言語処理系で採用されている．しかし，この方式にも，いつ殿堂入りをするかという殿堂入り問題や，長寿命領域におけるごみの回収の問題など解決しなければならない問題は多い．

3.2.3 その他のごみ集め

以上の他にもさまざまなごみ集めの方法が考案されている．

保守的ごみ集め

記憶領域におけるポインタの所在が正確に把握されていないような状況で行われるごみ集めは，保守的 (conservative) ごみ集めと呼ばれる．この保守的ごみ集めの導入により C や C++といった汎用言語においてのごみ集めの研究が盛んに行われるようになってきた．

これは，マーク・スイープやコピーなどによる方法ではポインタ追跡は重要不可欠な作業である．つまり，追跡を行う回収器はセルへのポインタの位置を正確に把握している必要がある．しかし，実際には正確なポインタの情報が無くてもある程度の識別は可能である．これは，ポインタであるかどうか曖昧な場合はポインタと同じビットパターンを持つかどうかで判定を行う．これにより，本当はごみであるものも回収してしまう場合もあるが，いわゆるダングリングポインタを生み出さないという点でこのごみ集めは安全に動作する．また，一般にポインタは，レジスタ，スタック，ヒープなどといった各領域に存在するこれら全領域で行う保守的ごみ集めを特に，全保守ごみ集めと呼び，一部でのみ行うものを半保守ごみ集めと呼ぶ．この方法の最大の欠点はごみの回収率で特に局所性の高いごみの回収率に難点がある．しかし，まだまだ新しい研究であり，今後の発展が期待できる．

ごみを出さない処理系

Linear Logic に基づく Lisp は，すべてのセルのリファレンスカウントを１に保ったまま Lisp の基本関数が実装できるという報告がある．このことによって，不要になったセルはただちに回収できることになり，まったくごみが発生しない処理系をつくることができるというものである．

新しいモデルを利用したもの

市場原理に基づき記憶領域を「賃貸」とするという考えがある．賃貸料を払うために，個々のセルは自分を参照しているポインタから「利用料」を徴収し，その一部を賃貸料にあてるとともに自分が参照するセルへの利用料として渡すというものである．これを資金の流れで見て行くとルートからのマークに相当し，これを単なるマークではなく金額という連続量とすることで記憶領域の「賃貸相場」とあわせて階層型記憶の有効利用を行っていくというものである．

他には，熱力学のアナロジーから記憶管理を説明しているものもある．熱・温度・エントロピーを導入しごみ集めは冷却であるという結論を導いている．

3.3 並列ごみ集め

3.3.1 グローバルごみ集めとローカルごみ集め

特に，分散メモリ型マルチプロセッサにおいてごみ集めを行う場合，各ノード内の参照関係のみならずノードをまたいだ参照関係も考慮しなければならない．そこで，このように，ノード内の参照関係のみを扱うごみ集めと，ノードをまたぐ参照関係を含めたごみ集めを分けて考えることがある．そして，前者はローカルごみ集め，後者はグローバルごみ集めと呼ばれる．ローカルごみ集めの場合ノードをまたいだ参照関係を扱わないので，シングルプロセッサにおける伝統的なごみ集めをほぼそのまま利用することができる．さらに，分散ごみ集めでは，一括型と即時型というように分けることもできる．一括型は，割り付けが可能なメモリが減って来た時点で mutator の処理を中断し，すべての mutation を停止した上でごみを回収するという方式である．即時型は，mutator がごみセルを積極的に検出しその場で回収するというものである．

3.3.2 外部参照の実現

分散環境ではプロセッサをまたがるポインタの表現は処理系設計の上でも基本的なことの一つにあげられる．メッセージ通信型並列計算機ではシステム内のアドレスは p をプロセッサ a をアドレスとした時に $< p, a >$ というように表現される．また，プロセッサをまたがるような場合には輸出表や間接参照表と呼ばれる外部参照のための表を用意し，e を間接参照表のエントリとすると $< p, a >$ というように表現する．また，外部参照と内部参照を区別しないという方法もあるが，分散共有メモリアーキテクチャなど，内部アドレスと外部アドレスに区別が無いような場合でないと内部参照の際のオーバーヘッドが大きくなるので，特別に扱う方がよい．また，間接参照表には，外部参照一つにつき一つのエントリを対応させるほか，同じ参照先ごとに一つのエントリを対応させる方法など複数の方法があり，ごみ集めの方法，メモリの種類などによって使い分ける必要がある．

3.3.3 リファレンスカウントによる並列ごみ集め

マルチプロセッサにおけるリファレンスカウントごみ集めは，mutation を中断すること無しに collection を行えるという長所は非常に重要なものとなる．さらに，一般にシングルプロセッサにおけるアルゴリズムをあまり変更せずに容易に用いることができる．そのためのリファレンスカウントによるごみ集めをマルチプロセッサで実装するためにいくつかのアルゴリズムが考え出されている．

ウエイトリファレンスカウント方式

分散環境におけるリファレンスカウントは，参照数の増減をメッセージによって他のノードに伝える．単純に考えるとこれは，以下の二つの問題が生じる．

1. 参照数の増減を伝えるメッセージの順序が不定.
2. そもそも増減によるメッセージ通信の量が多い.

このうち，ウエイトリファレンスカウント方式は２について改良の手段を与えている．これは，参照に重みを持たせて管理する方法である．

１ビットリファレンスカウント方式

１ビットリファレンスカウント方式 [4] は，主に分散ごみ集めで用いられる方式で，参照

側が単一参照かどうかを示す多重参照ビットを持つ．また，これは処理系によって，多重参照ビットが立っていないときはそれが単一参照であるということが保証している．このような場合，ポインタが捨てられた場合参照先はごみになる事が分かる．この際，多重参照ビットを操作しているのはすべて参照側であり局所的に行えるという利点がある．つまり，ごみ回収時以外では，参照先にメッセージを送る必要がないのである．

この方式では，セル生成時は多重参照ビットが落ちているが，一度立つと２度と落ちることはない．そのため通常のリファレンスカウント方式と比べると，回収されないごみの割合が高くなるという欠点がある．それゆえ，一括型のごみ集めと併用されるのが前提である．この方式の使い方としてはごみの溜る速度を抑えることによって一括のごみ集めの頻度をさげることであると言える．

3.3.4 トレースによる並列ごみ集め

トレースによるごみ集めでは，collection による mutation の中断がマルチプロセッサにおいてはより深刻な問題となる．さらに，ある時点での参照関係を得るといったことや，トレースの終了の検出も難しい問題となり，一般に多くのメッセージ通信を必要とする．

On-the-fly ごみ集め

On-the-fly ごみ集め [3] は，Dijkstra らによって考案された並列ごみ集めのアルゴリズムである．これは，mutator と collector の２種類を同時に並列に動かすことによって mutator の中断をなくそうというものである．基本となっているアルゴリズムはマーク・スイープとなっている．これが通常のマーク・スイープと異なるのは，セルのマークにそのセルが使用中か否かの２通りではなく，白，黒，灰色の３つを用いているという点があげられる．

インクリメンタルコピーごみ集め

Baker によるインクリメンタルコピーごみ集めは，いわゆるコピー方式のごみ集めを実時間化したものである．コピー方式のごみ集めは，ルートから直接指されているセルを全て to space に先頭からコピーを行い，新しいセルを指すようにルートのポインタを更新する．そして，to space を先頭から走査していき，セルを次々にコピーしていき，ポインタを更新してゆく．この際，ルートから複数の経路でたどることの出来るセルは二度以上コピーされてしまうので，セルのコピーを行う際に from space の古いところには，コ

ピー先へのポインタを書き込んでおく．これをリードバリアと呼ぶ．このようにすることによって複数回のコピーを防ぐことが出来る上に，毎回セルのリンク関係が保存されるのため途中で中断することが出来る．そこで，Baker によるアルゴリズムでは，このごみ集めを細かく分けてプログラムを実行する間に少しずつ実行するために，インクリメンタルと呼ばれる．しかし，このリードバリアはかなりのオーバヘッドを生ずることになり，またこのオーバヘッドを根本的に解決する方法はいまだに存在しない．

第 4 章

項書換え抽象機械

4.1　TRAM

項書換え抽象機械 TRAM は，特に実行速度の向上に重きをおいて設計・実装された抽象機械である．その特徴として，パターンマッチ処理部に弁別ネットを用いたパターンマッチの高速化や，その内部では項をすべて抽象的な機械命令列で表現しインタプリタで実行することができるといった事の他，E-戦略の採用によるユーザによる書換え順序の制御といったことがあげられる．

4.1.1　TRAM の概要

TRAM は規則のコンパイラ，入力項のコンパイラ，抽象命令のインタプリタの３つの処理ユニットと DNET，CR，CODE，STACK，SL，VAR，CANDS の７つの領域からなる (図 4.1)．そのうち CODE，STACK，SL，VAR，CANDS の各領域は書換えの際，動的に内容が変化する．

4.1.2　E-戦略

E-戦略とは，演算子ごとに簡約の順番をユーザが指定することの出来る戦略である．この指定は数列を用いて行われこの数列の各要素は 0 が全体項簡約をあらわし，数字 n により，n 番目の引数項の簡約をあらわしている (ただし，n は引数の数以下)．

次のような書換え規則があったとする

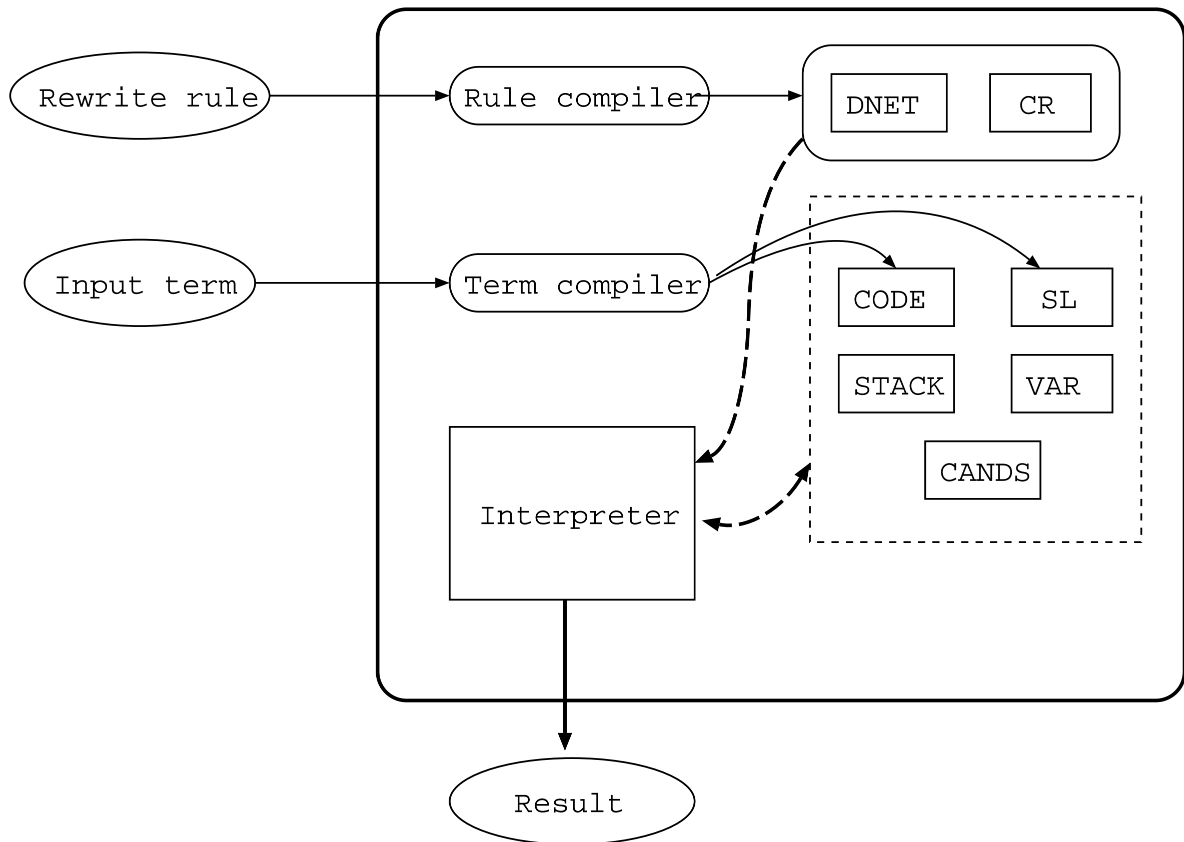

図 4.1: TRAM の構成

```
prev( X, Y ) -> X.    { strat: ( 1 0 ) }
back( X, Y ) -> Y.    { strat: ( 1 0 ) }
```

この場合の戦略はどちらも第一引数を簡約しその後で，全体項の簡約をすることを指定している．この場合 prev では，第一引数を簡約したあとで全体の簡約を行うので効率の良い書換え順番となっている．それに対し back では第一引数を簡約した後で全体項の簡約を行うがこの時点では第二引数項の簡約が終ってないので，これを行う必要がでてくる．そこで

```
prev( X, Y ) -> X.    { strat: ( 1 0 ) }
back( X, Y ) -> Y.    { strat: ( 2 0 ) }
```

というような指定に変えるとどちらも無駄を省いた簡約が出来るようになる．

4.1.3 書換え規則と入力項のコンパイル

入力された書換え規則は，書換え規則のコンパイラにより規則の左辺を弁別ネットに，右辺を右辺のマッチングプログラムの雛型と戦略リストの雛型にコンパイルされ，それぞれ DNET および CR に納められる．弁別ネットはシンボルをキーとして分岐した木構造の事で，これを用いることによりマッチする規則を効率よく探すことが可能になる．入力項はマッチングプログラムと戦略リストにそれぞれコンパイルされ，CODE および SL に格納される．

弁別ネット

弁別ネットは最外項シンボルをキーにして分岐しているパターンマッチ用の木構造である．この弁別ネットを用いることによってマッチする規則を効率良く検索することが可能となる．

右辺の雛型

TRAM では項をマッチングプログラムによって表現している．このため書換えが行われ項の構造が変化するとこのマッチングプログラムも変化する．また，書換えとは書換え規則の左辺を書換え規則の右辺に置き換える事であるので，そのための雛型をつくっておく必要がある．また，それに応じて戦略リストも再構成する必要があるので同様な雛型を作る必要がある．

戦略リスト

戦略リストとは，簡約の順番を制御するために用いられるリスト構造である．TRAM では E-戦略を採用しているのでユーザによって指定された戦略をあらわしていることになる．抽象命令のインタプリタはこの戦略リストの順番にそってマッチングプログラムを解釈していき書換えを行っていくことになる．また，定数，構成子，すでに正規系になっている項など，事前に書換えの必要がないと判別出来る項があった場合，そのような項に対して戦略リストを生成しないという最適化も行っている．

4.1.4 マッチングプログラムの構成

マッチングプログラムとは，適用可能な書換え規則を検索し，弁別ネットを用いて変数束縛を利用して実際に書換えを行う抽象機械命令のことである．マッチングプログラムは，

1000:	match_symbol "plus"
1001:	1003
1002:	1006
1003:	match_symbol "s"
1004:	1005
1005:	match_symbol "0"
1006:	match_symbol "s"
1007:	1008
1008:	match_symbol "s"
1009:	1010
1010:	match_symbol "0"

図 4.2: マッチングプログラムの構成

ラベル，シンボル，インデックスからなる抽象命令列 (図 4.8に項 plus(s(0),s(s(0))) のマッチングプログラムを示す) であり，CODE 領域に格納される．また構造的に項と等価なものと考えることができ，入力項はすべてこのマッチングプログラムの形で蓄えられる．

TRAM には抽象命令を解釈・実行するインタプリタが実装されており，直接マッチングプログラムに適用することにより簡約を行うことができる．書換えが行われる度にマッチングプログラムは動的にその内容が変化する自己改変となっている．書換えでは，用いた規則の右辺のマッチングプログラムの雛型のインスタンスを CODE 領域に格納し，それを指すようにポインタを張り直す．その結果，張り直す前に指されていた部分はごみとなる (図 4.3の網掛け部がごみである)．これは，メモリのフラグメンテーションを引き起こすことになる．

アドレス	オペレータ	オペランド
100000:	match_symbol	"plus"
100001:	100003	
100002:	100006	
100003:	match_symbol	"s"
100004:	100005	
100005:	match_symbol	"0"
100006:	match_symbol	"fact"
100007:	100008	
100008:	match_symbol	"s"
100009:	10000A	
10000A:	match_symbol	"0"

→

アドレス	オペレータ	オペランド
100000:	match_symbol	"plus"
100001:	100003	
100002:	10000B	
100003:	match_symbol	"s"
100004:	100005	
100005:	match_symbol	"0"
100006:	match_symbol	"fact"
100007:	100008	
100008:	match_symbol	"s"
100009:	10000A	
10000A:	match_symbol	"0"
10000B:	match_symbol	"s"
10000C:	10000D	
10000D:	match_symbol	"0"

図 4.3: マッチングプログラムのフラグメンテーション

4.1.5 抽象命令のインタプリタ

書換え規則と入力項のコンパイルが終了すると抽象命令を解釈し書換えを行うインタプリタが動き出す．この抽象命令のインタプリタは以下のように動作し書換えを行っていく．

1. TRAM の初期化
2. 戦略リストの先頭から対応するマッチングプログラムを取り出し，そのマッチングプログラムに制御が移る．この際，書換えの終了を示すラベル `BINGO` が来た場合には終了の処理に移る．
3. 上でマッチングプログラムに制御が移ると，適用可能な規則の右辺を探しだし処理が戻って来る．適用可能な規則が見つかった場合は，バックトラックを引き起こし他の全ての適用可能な規則を探し出す．逆に一つも見つからなかった場合は初めに戻る．

4. 適用可能な規則のなかから実際に適用する規則を一つ選び出す.

5. 書換えを行う．つまりマッチングプログラムを選択された規則の右辺に置き換え，戦略リストの再構成を行う.

6. 以上を繰り返すため最初に戻る.

4.2 Parallel TRAM

Multilisp をはじめ単一プロセッサ向きに設計・実装された言語処理系を並列拡張する研究はいくつも行われており，Parallel TRAM は TRAM を並列拡張し簡約の効率向上を目指したものである．また，TRAM のような項書換えシステムでは潜在的に多くの並列性を持つことが報告されており [12]，同時に多すぎる並列性も指摘されている．そこで，Parallel TRAM では引数項の簡約のみを並列化の対象としている．さらに，TRAM が簡約の順番をユーザが決めることができるという E-戦略を用いているのを継ぎ，Parallel TRAM では E-戦略に引数項の並列簡約を明示的に指定できるように拡張した並列 E-戦略を用いている.

4.2.1 Parallel TRAM の構成

Parallel TRAM では，CPU，メモリといった資源を処理ユニットと呼ばれる単位に分けて管理する．TRAM と同じ構造をそのうちのひとつの処理ユニットに割り当て，その他の処理ユニットに抽象命令を解釈・実行するインタプリタと簡約の際，内容が書換えられるおそれのある CODE，CL，STACK，VAR，CANDS の五つの各領域をそれぞれ割り当てる (図 4.6)．また，これ以外にすべての処理ユニット間で共有する DNET，CR とごみ集めで用いられる参照テーブルを置く.

4.2.2 Parallel TRAM のメモリ管理

TRAM のメモリ構成は前に述べたとおりで，Parallel TRAM では SL，STACK，VAR，CODE，CANDS の五つの領域が処理ユニットごとに複製されて用いられる．ただし CODE 領域は，最も頻繁に書換えられるので他とは違う扱いをしている．CODE 領域はこの全体を比較的小さなブロックに分割し，必要に応じてブロックを確保するという方法をと

アリティ n の演算子 f に対し

<StrategyDefinition> ::= ε | ”{” ”strat:” <UserDifinedStrategy> ”}”
<UserFifinedStrategy> ::= ”(” ”)”| ”(” <ReductionSeq> <Whole> ”)”
<ReductionSeq> ::= ε | <ReduceElement> <ReductionSeq>
<ReduceElement> ::= <Whole> | <Arg> | <ParallelReduction>
<ParallelReduction> ::= ”{” <ArgReductions> ”}”
<ArgReductions> ::= ε | <Arg> <ArgReductions>
<Whole> ::= ”0”
<Arg> ::= ”1” | ”2” | $\cdots$ | ”n”

図 4.4: 並列 E-戦略の構文

る．これは，処理ユニットによって領域の使用量が異なるためで，少しでも CODE 領域を効率よく消費していくための工夫である．

4.2.3 並列 E-戦略

引数項の並列簡約をサポートするために E-戦略の指定構文を図 4.4のように拡張する．以下の例でみてみると，

```
f( X, Y, Z, W ) -> ...    { strat: ( 1 { 2 3 } 4 0 ) }
```

この場合，まず第一引数を簡約し，続いて第二，第三引数を並列に簡約し，この並列簡約が終了するのを待って第四引数を簡約し，最後に全体項の簡約を行う．

4.2.4 Parallel TRAM の戦略リスト

Parallel TRAM では並列簡約を実現するためにあらたにいくつかの抽象命令が加えられた．インタプリタはこの新たな命令を実行することによって並列簡約を行っていく．追

加されたのは以下の五つの命令である．

FORK：

アイドル状態の処理ユニットがある場合には，その処理ユニットに戦略リストに続く１ブロックの簡約をEXIT 命令を付加して割り当てる．アイドル状態の処理ユニットが無い場合，その部分の簡約は自ユニットで行われることになるので，この命令はなにもしないことになる．

WAIT：

FORK した簡約がすべて終了するまで待機する．待機中は自ユニットの状態をアイドルにして他の簡約を受け付ける．

EXIT：

FORK 命令によって割り付け先の処理ユニットにまず割り当てられ，FORK 元の処理ユニットに簡約が終了した事を伝える．

SLEEP：

サブプロセスユニットにおいて簡約すべき項が無くなった時に実行される．この命令が実行されると自ユニットの状態をアイドルにし，他からの簡約を受け付けられる状態にする．

NOP：

なにもしない空命令である．4.1.3 節の戦略リストで説明したように実際の戦略リストの要素には最適化の対象となり取り除かれるものがでてくることがある．このような場合この NOP 命令を埋めることにより補正する役割を持つ．

上記のうち FORK，WAIT，EXIT について，後述の図 4.9〜図 4.11に次のロック機構をまじえて図示する．

4.2.5 Parallel TRAM の同期処理とロック機構

一般にマルチプロセッサ上では，プロセスやスレッド間で同期をとる必要がある場合や排他的に処理しなければならないクリティカルセクションが存在するような場合がある．このような場合，スレッドの join や resume，suspend を用いるといった方法や mutex 変

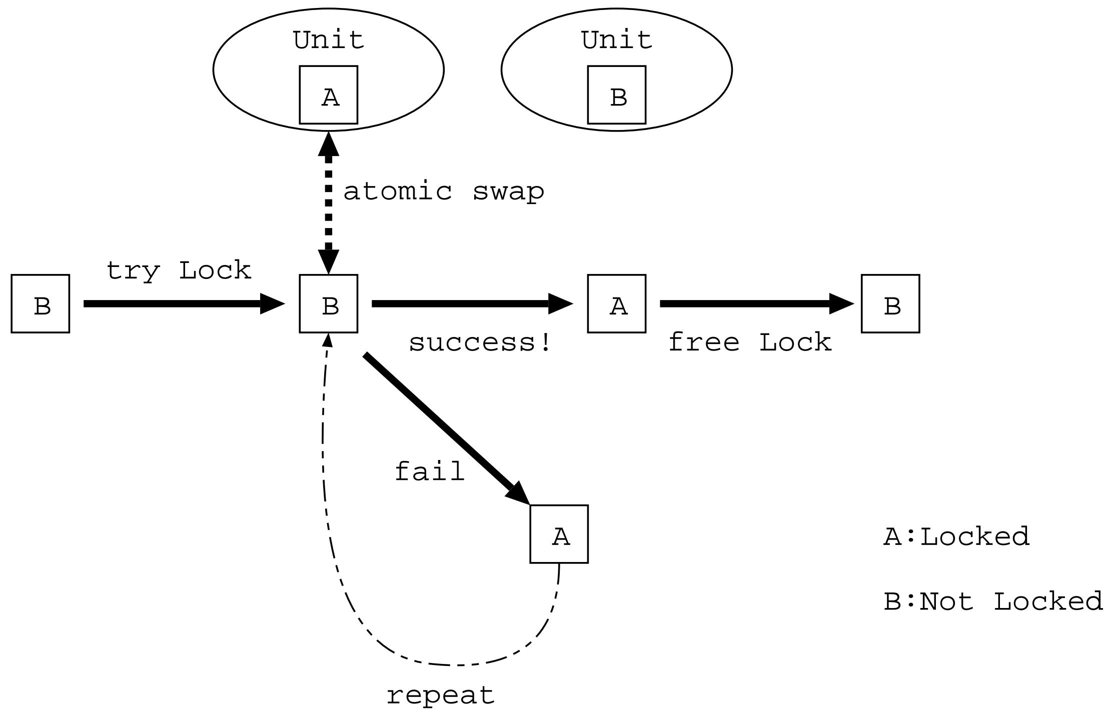

図 4.5: ロックの仕組み

数を利用するなどで対処することができる．しかし，これらの処理は一般に非常に重い処理である．そこで Parallel TRAM では，スピンロックによる busy wait を採用している．たとえ，busy wait を用いたところで処理が重いというのは変わりがないのであるが，待ち状態に入る，または待ち状態から復帰するといった動作はスレッドの resume，suspend を用いた場合よりもかなり少ない事が分かる．つまり中断している時間が短いような場合，スレッドの resume，suspend を利用するよりも busy wait を利用したほうが，オーバーヘッドを軽減することができるのである．さらに，Parallel TRAM の場合，待機状態になるのは簡約すべき項がない時におこるので busy wait とは相性がよいということが言える．

このロック機構を詳しく説明していく (図 4.5)．まず，準備として，ロック状態を表す値 A とアンロック状態を表す値 B の二種類の値を用意する．まず，はじめセルには値 B が入っておりロックがかかっていないという状態を示している．このとき，あるユニットがそのセルの参照を行う際にはまずこのロックを獲得する必要がある．これは，atomic である swap といった命令を用いてセルと値 A を入れ換えるという動作を行う．このような

動作を行うと他にロックが確保されていない状態ならは，値 B が得ることができ，これはすなわちロックを獲得したという状態になる．また，この際セルには値 A が入ることが分かる．もし，値 A が帰ってきたなら他のユニットによりロックされているという事になり，この動作を繰り返してロックを獲得することになる．

また，単純な test-and-set 方式のスピンロックを用いてしまうとロック中は重いという事実は変わらない．そこで実際には遅延を導入して実装されている．遅延を用いることによってスピンロックの性能が向上するというのは Anderson[1] により報告されている．

4.2.6 プロセス処理ユニットの管理

Parallel TRAM には FORK という命令があり，待機状態にあるアイドルプロセッサが存在する場合には引数項の簡約をまかせるというのは前に説明した通りである．そこで，アイドルプロセッサを管理する機構が必要となるのだが，Parallel TRAM ではアイドルキューというものでこれを管理している．これは，アイドルプロセッサを FIFO キュー構造で連結させただけの簡単な構造になっている．また，このキューの操作はそれぞれの処理ユニットごとに追加，削除といった動作を行っており，この操作には排他制御が必要となっている．

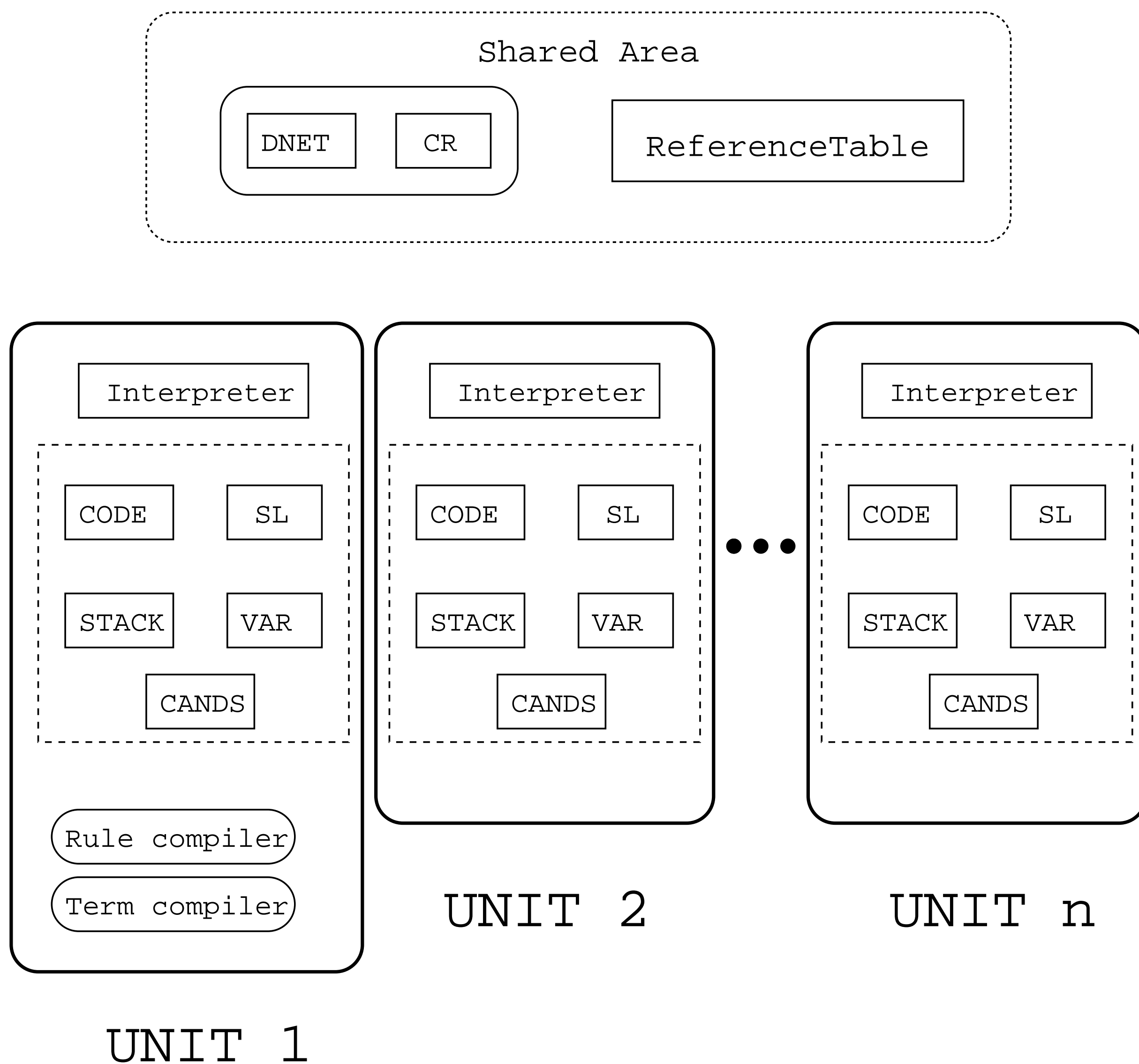

図 4.6: Parallel TRAM の構成

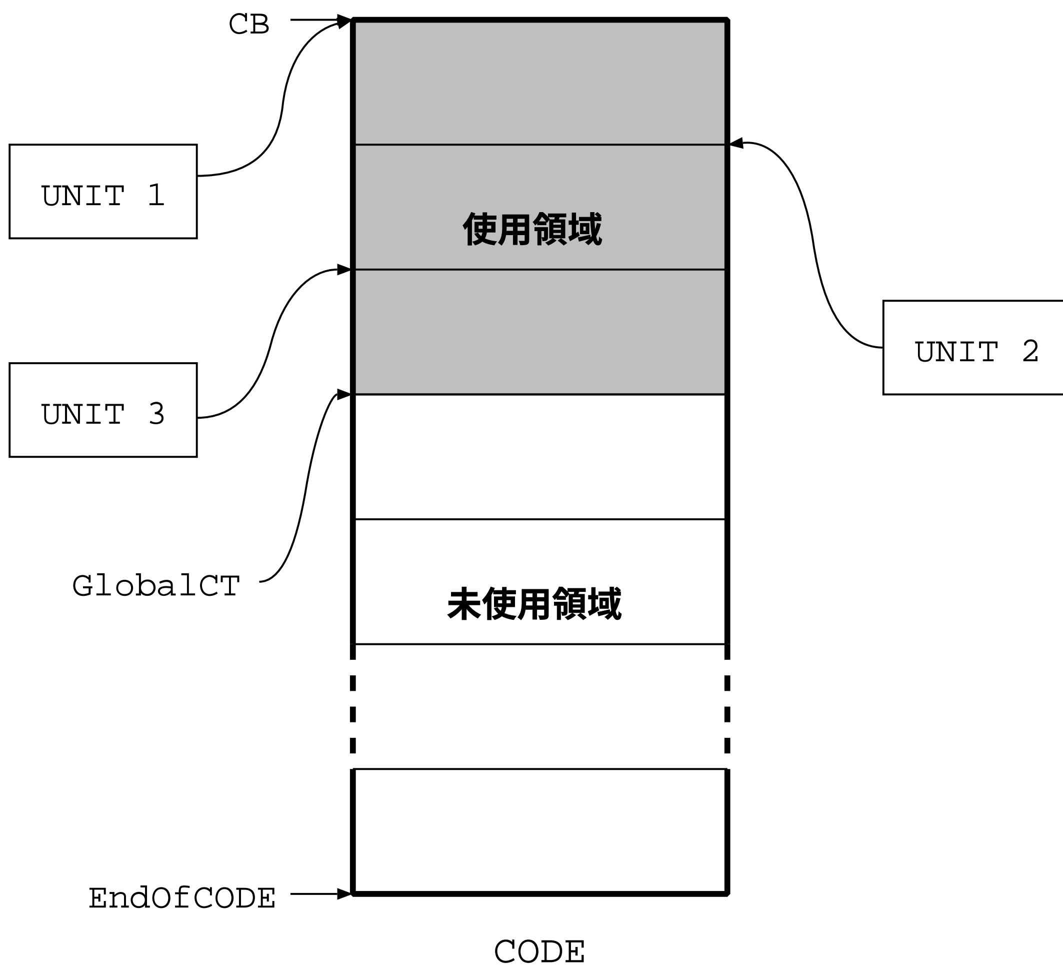

図 4.7: CODE 領域の構成

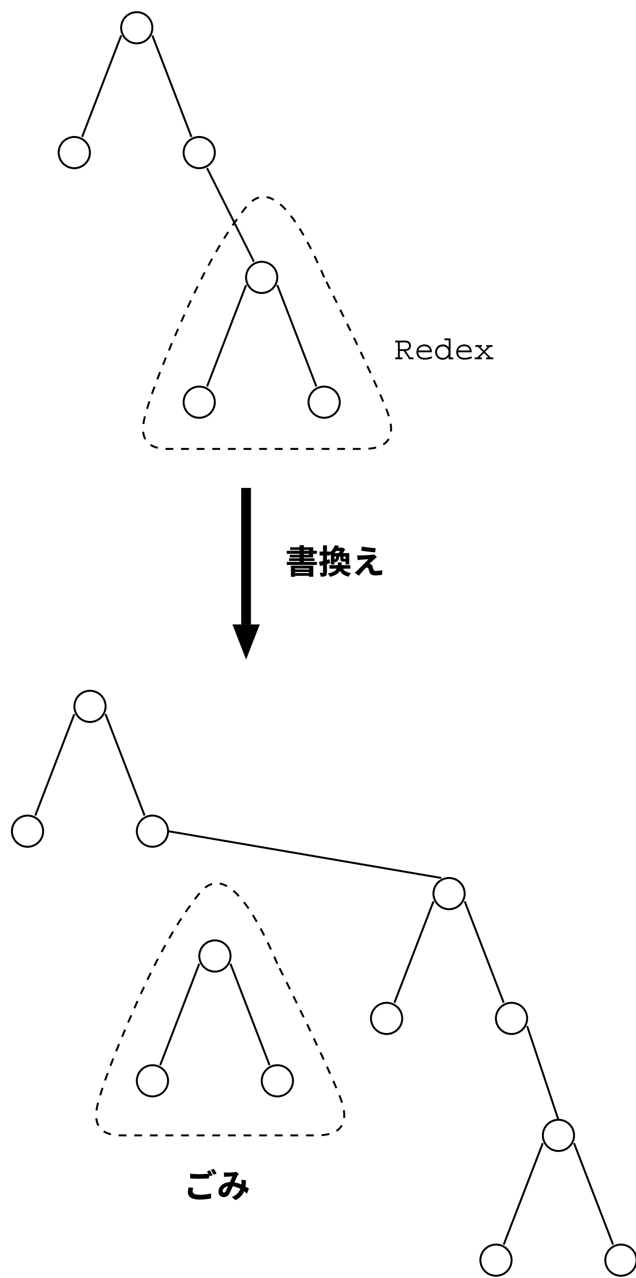

図 4.8: マッチングプログラムにおけるメモリフラグメンテーション

< FORK , fork する範囲 , 対応する WAIT の位置 >

空いている処理ユニットを探す

- 空き処理ユニットがない場合
  - WAIT の数を減らして終了
- 空き処理ユニットがある場合
  - fork 先から戻るための EXIT 命令の埋め込み
  - 戦略リストとマッチングプログラムを空き処理ユニットに送る
    この際，参照テーブルを通すようにする
  - fork 先のプログラムカウンタや state をセット
  - fork 先を起こす
  - 終了

図 4.9: FORK の定義

< WAIT , WAIT する数 , − >

WAIT する数に lock をかけて数を調べる.

- 0 の場合
  - lock を解放して終了.
- 1 以上の場合
  - 自身を suspend 状態にセット
  - プロセスキューを操作するための lock を得る
  - 自身を state を idle にしてキューの先頭に加える.
  - プロセスキューを操作するための lock を解放
  - WAIT 数の lock を解放
  - 起こされるのを待つ

図 4.10: WAIT の定義

< EXIT , WAIT する数 , 親ユニット >

WAIT する数に lock をかけて数を調べる.

- 1 より大きい場合
  - WAIT 数を 1 減らして lock を解放して終了.
- 1 の場合
  - プロセスキューを操作するための lock を得る
  - 自身を idle キューから外す.
  - 親ユニットを起こす.
  - プロセスキューを操作するための lock を解放
  - WAIT 数を減らし，lock を解放

図 4.11: EXIT の定義

第 5 章

ごみ集めの並列化

5.1　Parallel TRAM におけるごみ集め

TRAM においてごみ集めの対象になるのは，CODE 領域に蓄えられるマッチングプログラムのみである．書換えにより変更されるのは部分的なマッチングプログラムであり，変更後のマッチングプログラムを空き領域に格納しそれを指すようにポインタが変更される．この操作によって変更前の部分的なマッチングプログラムはどこからも参照されないセルとなることが分かる (図 4.8)．これがごみとなるのである．TRAM でのごみ集めの方法は，単純なコピー方式を用いていた．

Parallel TRAM でのごみ集めの基本的な方法は TRAM の場合と同様にコピー方式のごみ集めとなっている．これは，

- セル (マッチングプログラムの各要素) の長さが不定なため詰め替えの動作が必要
- ごみに比べ必要なセルの量が少ない

という大きな２つの理由からである．

Parallel TRAM では次のような流れでごみ集めが行われている．

1. ごみ集めの必要性を感じた処理ユニットが `GCFLAG` を立てる．
2. 各処理ユニットは一回の書換えが終わるごとに `GCFLAG` をチェックし，このフラグが立っていた際には，自ユニットの状態を GC にして待機する．

3. 1. でごみ集めの必要性を感知した処理ユニットは，他の全てのプロセスユニットが待機状態になるのを確認してごみ集めを行う．

4. コピー方式のグローバルごみ集めを行う．

5. ごみ集めが完了したら `GCFLAG` を元に戻し，各処理ユニットを再開させる．

ここで，`GCFLAG` はごみ集めの必要性を表しているフラグで各プロセスで共有している．

また，マルチプロセッサ上でごみ集めを行うには何かしらの同期をとる必要がある．Parallel TRAM の場合，プロセッサ間で大域的に同期をとる方法を採用していた．また，ごみ集めを行うプロセッサは１つに限られるため，ごみ集め処理中は完全に他のプロセッサの処理が止まる．つまり，ごみ集めを行う際に collector は mutator の処理に区切りがつくまでまたされることになり，またごみ集めを行うのは１つの collector に限られるという大きなボトルネックを持っている．このボトルネックは計算と並列してごみ集めを行う並列ごみ集めを採り入れることにより軽減できるものと思われる．

5.2 並列ごみ集めの設計

5.2.1 メモリ管理

基本的なメモリ構成は Parallel TRAM の場合と同じ方法を用いる．ただし，CODE の扱いは並列ごみ集めの導入により変わることになる．Parallel TRAM での扱いは前に述べた通りであり CODE とごみ集めに用いられる PAST と FUTURE の各領域をそれぞれユニットごとに持たせるように変える．この領域はそれぞれ 1M バイトごとを割り当てる．図 5.1にその構成をしめす．

5.2.2 並列ごみ集め

並列ごみ集めのアルゴリズムとしては，前に挙げた On-the-fly ごみ集めが代表的なものとしてあげられる．

しかし，基本的なアルゴリズムがマーク・スイープ方式であるため，TRAM で用いているマッチングプログラムのようにサイズの異なるセルを扱う場合，詰め替えが必要になると言った事や，セルを二度スキャンする必要があるといった欠点がある．また，TRAM

では Lisp や Smalltalk といった言語にみられる特徴である生成されるセルの多くがごみになるという性質を持つので，コピー方式を基本とするごみ集めが適すると考えられる．

また，一般に分散環境においては，ごみ集めも一つの分散アルゴリズムなので，終了判定や同期取りといった分散アルゴリズムにおける主要な問題点を引き継いでいる．例としては，メッセージの到着不順の問題やフォールトトレラントの考慮などがあげられる．しかし，ここで対象となるのは共有メモリ型マルチプロセッサであり，プロセッサの数も少ないので，このような危惧はほぼ無いものと考えられる．

5.2.3 コピー方式の並列ごみ集め

コピー方式を基本としたごみ集めには Baker[2] によるコピー方式インクリメンタルごみ集めなどがある．しかし，この方法ではそもそも実時間ごみ集めであり，リード・バリアと呼ばれる同期をとるためのオーバーヘッドが大きく，処理時間の短縮という目的には向かない．そこで，Parallel TRAM の各処理ユニットの独立性に注目し，分散メモリ型マルチプロセッサで用いられている分散ごみ集めにコピー方式を採用したごみ集めを考える．Parallel TRAM の CODE 領域は処理ユニットごとに個別に持ち，それぞれは各処理ユニットで独立しているという実装にもよく合うと考えられる．

これは，各処理ユニットは通常は mutator となり計算を行うが，ごみ集めが必要と判断すると collector となり処理ユニットに割り当てられている CODE 領域のごみ集めを行うものである．また，実際のごみ集めは，TRAM や Parallel TRAM と同じくコピー方式のごみ集めを行う．これを各処理ユニットごとに非同期に行う方法を採用する．しかし，このままでは処理ユニット外からの参照があった場合，参照先のセルの内容が保証されないという問題が起きる．Parallel TRAM では他の処理ユニット内のセルを参照する可能性があるので，このような外部参照が起きる場合がある．これを解決するために外部参照のテーブルをつくる方法 (図 5.2上) で対処する．

5.2.4 外部参照テーブル

外部参照テーブルは，分散ごみ集めの分野では輸出表と呼ばれるものと役割的には同じ物と考える事が出来る．しかし，構造などを考えてあえてこのように呼ぶことにする．この外部参照テーブルを実現するために，

- GCForward
- GCLocked

の２つの抽象機械命令を新たに導入する．外部参照テーブルは構造的にはマッチングプログラムと同じような構造をしており (図 5.2下)，そのオペレータ部に GCForward，その引数部に参照先のアドレスを表すポインタを持つ．

このような構造を持つマッチングプログラムの集まりを共有される領域上にテーブルとして置く．これが外部参照テーブルと呼ばれるものである．このように一ヶ所に集めることによって同期をとる際に lock をかけたりするのが容易になる．

このような外部参照テーブルを用いることによって collector と mutator が並列に動作することが可能になる．collector は，まず参照テーブルのうち自身を参照しているテーブルに対して lock をかける．この動作は実際には

参照している処理ユニット番号と参照先アドレスからなる．処理ユニットを越える参照を行う時には，必ずこの外部参照テーブルを通して参照を行うこととする．また，ごみ集めではこの外部参照テーブルより，各処理ユニットは自分の中のどのセルが参照されているかが分かるので，ごみ集めによりそのセルの場所が変わるようならば更新する．このようにして，処理ユニット外からの参照が問題なく行えるようになる．Parallel TRAM は共有メモリのマルチプロセッサ上に実装されているので，分散環境と比べ比較的少ないオーバーヘッドでこれを実現できると思われる．しかし外部参照テーブルの参照や書換えをおこなう際には，排他制御のためのロックを掛ける必要があるというオーバーヘッドが生じる．

5.2.5 外部参照テーブルの排他制御

一般にマルチプロセッサにおけるプログラミングでは，共有されるオブジェクトに触れる際は排他制御を行わなければならない．これは，同時に複数のプロセッサによって例えば変数といったものが書換えられるとその結果は不定となるという問題が発生するのである．本研究の場合も外部参照テーブルは共有されているので，排他制御を行わなくてはならない．しかし，排他制御では，mutex 変数，セマフォまたはハードウェアによって atomic(この命令の実行中にはいっさい他からの割り込みが許されないような命令) であることが保証されている機構を用いる必要がある．そして，この処理は非常に重いこと

が知られている．

そこでなるべく排他制御を行わずに外部参照テーブルの参照を行うようにすれば効率が向上するものと考えられる．しかし，単にこのようなことを行えば問題が発生する．そこで，この問題が起こるメカニズムを見ていく．まず基本的なことがらとして，参照だけが複数存在するような場合これは，どの処理ユニットがどのようなタイミングで参照したとしても問題は起きない．逆に，変更が行われるときはたとえ他に一つの処理ユニットからでも参照されていると問題が起こる可能性が出てくる．これは，参照側が走査したのちにセルの変更が加えられたり，走査する前にセルの変更が起こるのことになり，参照側が得られる結果が不定となる．つまり，ここで排他制御を行う必要性がある．では，このParallel TRAM ではいつ共有領域を書換える可能性があるのかをみていく．

Parallel TRAM の並列ごみ集めにおいて，外部参照テーブルに対し変更を加えるのは，fork する時と，書換えた時に書換え元の項が他のプロセッサ上にある時，それに，ごみ集めにおける排他制御といった場合が考えられる．ここでは，最後のごみ集めにおける排他制御について注目してみる．

ごみ集めではまず，ごみ集めの必要を検出した処理ユニットが参照テーブルを走査し自ユニットへの参照に対しロックを掛ける．全ての collector の参照に対しロックが得られた時は，collector が完全に独立する時である．この場合に collector 内を書換えてもあとでテーブルの参照を直しておけば問題は無い．しかし，この処理のためには mutator はテーブル通して参照している間はそのテーブルを他から参照されないように排他制御を行う必要がある．そこでこの重い排他制御を行わずに参照する方法を考える．

排他制御を行わないといってもすべてのロックを取り除いたりしてしまうと，動作は保証されないので，ロックを掛ける場所を減らすことを考える．そうすると，mutator は書換え後のマッチングプログラムを自ユニット内で構築して書換え前の項からポインタのつけかえという動作で書換えていることが分かる．これはすなわち，書換えるのはテーブルまでであることがわかる．テーブル先 (つまり，他処理ユニット内にあるマッチングプログラム) は参照するにとどまるのである．そこで，テーブルから先の参照ではロック動作は行わないものとする．これによって，ロックでかかるオーバヘッドを減らすことが可能となる．

しかし，単にロックを行わないというだけでは，参照中にごみ集めが起こり，そのセルの位置が変わるといった事が起こりうる．そこで，ごみ集めでは，まず参照テーブルに

ロックを掛けてこれ以降に新たな参照が起こらないように排他制御を行う．その上で，他の処理ユニットが最低１回以上の書換えが済むのを待つことにする．このようにすればロックを掛けた後参照される処理ユニットは存在しないことが保証される．ただし，実際にはロックを掛けた後に参照テーブルを見に行くことになりロックの獲得を待っているという状態や，書換えが終り SLEEP 命令で待機状態にあるなどといった場合書換えは起こらないが参照も起こらないのでこの場合もごみ集めを始めてよいことになる．

このアルゴリズムを書き下してみると次のようになる．

1. mutator がごみ集めの必要性を検出
2. mutator がごみ集めの動作をはじめる (以降 collector となる)
3. まず，参照テーブルを走査していき自ユニット内の参照に対し排他制御のためロックを掛ける
4. 他の処理ユニットの状態および書換え回数を見ていくことにより，ごみ集めを開始してよい状態になるのを待つ．
5. 他からの参照が無くなったのでコピー方式のごみ集めを行っていく
6. すべての生存セルのコピーが終了したら，参照テーブルを書換え，コピー先のセルを指すように更新する．
7. ごみ集めの処理が終ったのでテーブルのロックを開放する
8. 再び mutator となり処理を続ける

このようなアルゴリズムにより参照する側のロックの保持している期間を減らすことが可能となる．

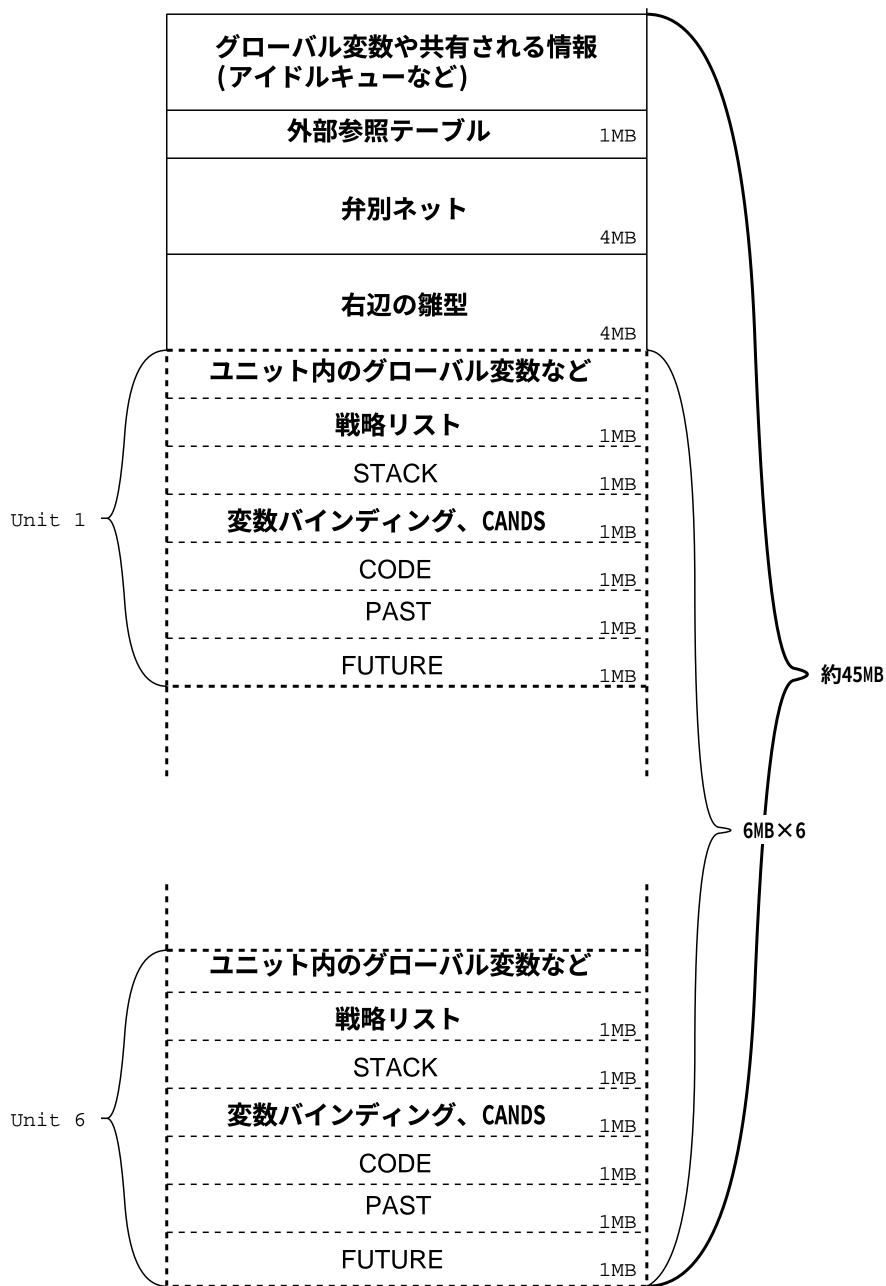

図 5.1: メモリ構成

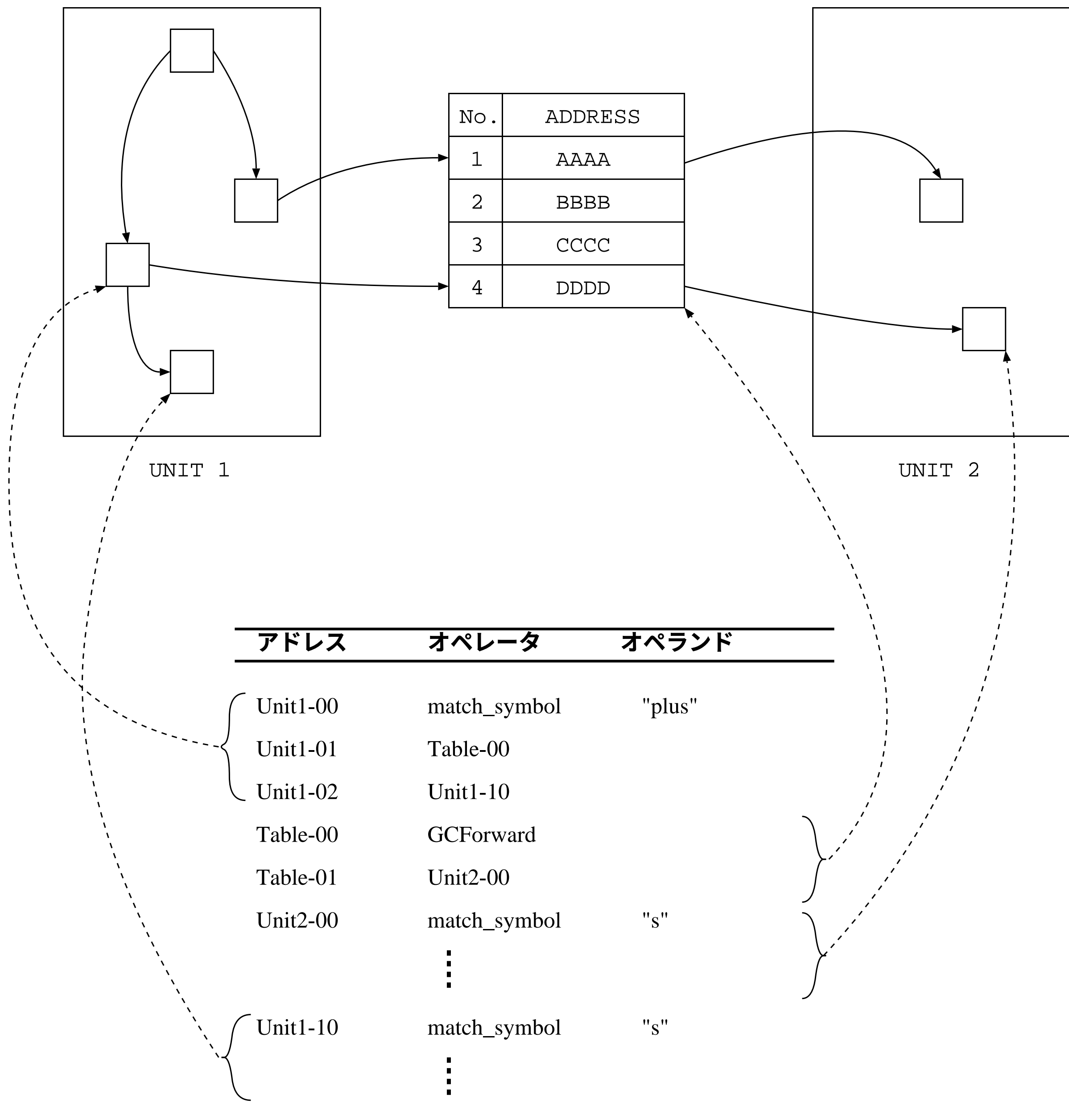

図 5.2: 外部参照テーブル

第 6 章

実験と評価

6.1 実験

本研究で Parallel TRAM のごみ集めをグローバル方式からローカル方式へと変更し実装した．これによってどの程度，効率の改善が実現できたのかをベンチマークにより評価をおこなう．

評価に使用した環境は以下の通りである．

機種 Fujitsu S7-7000U (Ultra Enterprise 4000)

CPU Ultra SPARC

CPU 数 6

クロック 167MHz

メモリ 256M

OS Solaris 2.5.1

使用コンパイラ gcc Version 2.7.2.3

評価には実行時間を用いる．計測には C のライブラリ関数 gettimeofday を用いた．

6.2 評価

今回実験を行ったベンチマークは次の通りである．

1. fib(20) の TRAM，Parallel TRAM の６プロセッサ時，並列ごみ集め版の Parallel TRAM の各プロセッサ時における計算
2. fib(23) の TRAM，Parallel TRAM の６プロセッサ時，並列ごみ集め版の Parallel TRAM の各プロセッサ時における計算
3. 200 個の項の sort の TRAM，Parallel TRAM の６プロセッサ時，並列ごみ集め版の Parallel TRAM の６プロセッサ時における計算
4. 100×100 の TRAM，Parallel TRAM の６プロセッサ時，並列ごみ集め版の Parallel TRAM の６プロセッサ時における計算

これらの計測結果を表 6.1に示す．

6.3 考察

6.3.1 Parallel TRAM の性能

結果から述べると，従来の Parallel TRAM と比較して最大約 20%程の向上がみられる．また，平均的にも約 1 割ほどの改善がみられた．このことは，従来の Parallel TRAM のごみ集めに大きなオーバヘッドがあるということを示している．これは，同期をとるというオーバヘッドとごみ集め中に動作しているプロセッサが collector の一つであるという二点が大きな要因であると言える．さらに，並列動作させるようにしたための排他制御や外部参照テーブルの処理などのオーバヘッドがかかっていることを考えれば実際のオーバヘッドはかなりのものであることが分かる．

また，基本的な性能としては，従来の Parallel TRAM の場合と同様にプロセッサの数に応じた速度の向上が見られた．

結果に注目してみると sort での速度の向上が大きいことが分かる．この場合についてさらに詳しく見てみると，書換え回数のわりに全 fork の数が少ないことが分かる．これは，マルチプロセッサではよく問題になる並列の粒度がその原因ではないかと推測でき

る．粒度とは，プロセスを並列に実行させた時に各プロセッサに与えられる仕事の大きさを表す言葉で，一般にこれが大きいと負荷分散がうまくいき，小さいとあまり好ましい負荷分散が行えないことが知られている．この sort 場合，他の場合を比べてこの粒度が大きかったと言える．逆にかけ算では粒度が小さくあまり適切な負荷分散が行えず，効率が悪かったと言える．

6.3.2 ごみ集めのよるオーバヘッド の計測

前のベンチマークとは別に，ごみ集めの前後に時間を測る関数を挿入して，ごみ集めによるオーバヘッド を計測してみた．ただし，これはごみ集めの方法がグローバルとローカルであるという違いやごみ集めの方法の違いにより回数が異なるなど単純に時間を計測したところでは正確な比較は行えないのであくまで参考的な数字である．fib(23) における結果は以下の通りである．

	全時間	GC の時間	比率
TRAM	16.725	3.010	18%
P-TRAM	7.214	1.462	20%
CP-T	6.502	1.138	17%

これによると，Parallel TRAM が TRAM よりもごみ集めに時間がかかっている事がわかる．これは，ごみ集めをはじめる際の同期をとるといった動作と Parallel 化による走査領域の増大がその原因と考えられる．また，ローカルごみ集め版 Parallel TRAM においては，ごみ集め時間が最小となっている．これは，ローカル化によるオーバヘッド の軽減とローカル化により集められるごみの量が減ったためと考えられる．このことは，ごみ集めの回数の増加にみることができる．

6.3.3 並列ごみ集めの性能

並列ごみ集めを用いることによって書換えプロセスがごみ集めによって完全に停止することは無くなった．しかし，参照テーブルを扱う際には同期をとる必要があり，これは処理系の性能を低下させる．

ごみ集めによる処理系全体の性能は，処理ユニットをまたがる参照が増えれば増えるほど落ちることが分かる．これは，処理ユニットをまたがる参照は外部参照テーブルの参照

を行うことになり，これがオーバーヘッドとなるからである．また，ごみ集め時においても collector への参照が増えるにつれて排他制御のためのロックを掛ける参照テーブルの数が増え，オーバーヘッドの増加を招く．さらに，いくら並列ごみ集めとはいえ，ごみ集めの最中の処理ユニットへの参照はロックにより妨げられており，処理系全体の性能の低下につながる．このように並列ごみ集めの性能は処理ユニットをまたがる参照に左右される．このような処理ユニット外への参照は，処理ユニットにどのように項を割り当てるかということに関係があり，これは Parallel TRAM における処理ユニットのスケジューリングに帰結することになる．Parallel TRAM では特別なスケジューリングを行っているわけではなく，順に空いている処理ユニットに仕事を割り当てているにすぎない．このため書換えが進むにつれ外部参照が増加することが予想される．これは外部参照をなるべく抑えるようなスケジューリングを行うことによって，性能の向上が期待できる．しかしこれには項の参照関係などの解析を行わなければならず，これは単純な作業ではない．

また，collector と mutator が独立に動作するというのは，mutator の参照が collector に及ばない場合のみである．つまり，mutator が collector 内のセルの参照を持つような場合，今回の実装では，ごみ集めの処理が終るまで待たされることになる．しかしこれは，同期をとるためにテーブルにかけるロックを改良することで軽減することができると考えられる．つまり，ごみ集めの最中に実質的にロックを掛けなければならないのは，生存セルの移動中のみであって移動する前と移動した後ではたとえ mutator からの参照があっても影響はでないはずである．

	書換え回数	時間	秒間書換え数	GC	fork 成功	fork 失敗	比率
fib(20) TRAM	185837	3.383	54948	4	-	-	1
fib(20) CP-T-1	185837	3.408	54545	6	-	-	0.99
fib(20) CP-T-2	185837	3.028	61385	7	11	317	1.12
fib(20) CP-T-3	185837	1.931	96242	10	98	230	1.75
fib(20) CP-T-4	185837	1.769	105052	10	110	218	1.91
fib(20) P-TRAM	185837	1.708	108797	6	102	226	1.98
fib(20) CP-T-6	185837	1.502	121011	11	122	206	2.20
fib(23) TRAM	887877	16.275	54562	23	-	-	1
fib(23) CP-T-1	887877	16.278	54545	25	-	-	1.00
fib(23) CP-T-2	887877	14.467	61385	31	1014	45353	1.13
fib(23) CP-T-3	887877	7.101	125032	43	3375	42992	2.29
fib(23) CP-T-4	887877	7.142	124350	45	5288	41079	2.27
fib(23) P-TRAM	887877	7.431	119490	25	1264	45103	2.19
fib(23) CP-T-6	887877	6.328	140302	53	8329	38038	2.57
sort TRAM	2726906	46.870	58180	42	-	-	1
sort P-TRAM	2726906	20.831	130905	48	3205	36595	2.25
sort CP-T-6	2726906	17.499	158831	85	7103	32697	2.73
100 × 100 TRAM	495203	12.969	38180	13	-	-	1
100 × 100 P-TRAM	495203	7.165	69114	13	38	62	1.81
100 × 100 CP-T-6	495203	7.126	69488	31	51	49	1.82

TRAM：従来の TRAM

CP-T-n：並列ごみ集め版 Parallel TRAM の n プロセッサ時

P-TRAM：従来の Parallel TRAM

表 6.1: 計測結果

第 7 章

まとめ

本研究では，Parallel TRAM において分散ごみ集めの考えを採り入れ，ローカルな同期を行う並列ごみ集め Parallel TRAM の設計・実装を行った．また，この際ごみ集めの基本となるアルゴリズムとしては，コピー方式のごみ集めを採用し，これを並列に実行するようにした．

7.1 並列ごみ集め

Parallel TRAM に並列ごみ集めを導入することにより，大きなオーバヘッドであると考えられるごみ集めの時間を短縮することが出来た．これには以下の二点で有利に働くものと考えられる．

- 同期をとるオーバヘッドを削減
- collector と mutator が同時に働く

しかし，これと同時にテーブルの操作という新たなオーバヘッドも生じることがわかる．そこでこのテーブルの操作でごみ集めに関して排他制御のためのロックを減らす工夫を行った．これは，書換え時の参照ではロックを用いずに参照を許し，そのかわりにごみ集めの時点で参照が終るのを待つという工夫であった．

7.2 今後の課題

本研究で実装した並列ごみ集めのさらなる評価と改良のためいくつかの課題を以下にあげる．

外部参照テーブルにおけるロックの扱い

現在の実装では collector は自ユニットへの参照をすべてロックしてからごみ集めを行っている．しかし，実際にロックを掛けなければならない部分は，対象となっているごみが，コピー中である，もしくはそのセルのテーブルを更新している間だけである．ロックを必要な間のみに掛けるならばロックによる mutator の待ち時間を減らす事が出来る．これは，lock granularity と呼ばれ一般に小さいほうがよいとされる．しかし，たとえ一回のロックを掛けている時間を減らしてもロックの回数が多ければ効果は得られないので，このバランスが重要な問題となる．

ごみ集めのオーバヘッド を計測する

本研究で，ごみ集めのオーバヘッド は実際にごみ集めに掛けた時間を測定した．しかしこの方法では，特に並列のごみ集めにおいて mutator が collector によって妨げられた時間やテーブルの参照や更新などにかかる時間といったオーバヘッド は出て来ない．そこでこれらをなるべく本来の処理を妨げずに測定する方法が必要である．

グローバルごみ集めとの併用

並列ごみ集めの導入によって一回で集められるごみの量は減っている．これはごみ集めの回数の増加などに見ることが出来る．そこでグローバルごみ集めと併用を行う．また，グローバルごみ集め時に回収出来るごみの量を計測する．このことによってどの程度ローカルごみ集めで未回収になっているのかを測定することができローカルごみ集めの効率を計測することができる．これによってグローバルごみ集めの頻度などに利用できるものと考えられる．

Parallel TRAM の並列化の効率の改善

現在の Parallel TRAM の実装では if 文などに代表される投機的な簡約をサポートしていない．これは，if 文では一般に条件式，then 節，else 節の三つの項があるがこれを一度に

この三つの項を簡約することができれば条件節の簡約が終った時点で then 節と else 節のどちらかを選ぶようなことができ，効率の向上が期待できる．このためには，簡約を途中で破棄するようなメカニズムが実現できれば可能であると考えられる．

Parallel TRAM のスケジューリングの改善

現在の Parallel TRAM ではとくにスケジューリングといった事は行っていない．管理という意味で行っているのは fork などにおいてアイドルプロセッサを簡単に得る事が出来るように単純な FIFO キューを用いているにすぎない．この管理だけでは，負荷分散がうまくいかず，ある特定のプロセッサに対し負荷がかかる可能性がある．これを負荷分散が行えるようなスケジューリングが行うことができれば効率の向上が期待できる．またごみ集めの観点からみても，あるプロセッサがアイドル状態でかつごみがある程度たまっているような場合，もしすぐに fork が行われないということが分かれっていればあらかじめごみ集めをしておくといったようなことも可能であると考えられる．しかし，これを実現するにはある程度項の意味を解釈せねばならずこの実現はそう単純なものではない．そのかわり，これはコンパイル時に行う必要があるので実行時の効率はあがるものと考えられる．

設計仕様の形式化

今回設計した並列ごみ集めや Parallel TRAM 自身についても仕様記述言語などを用いて形式化し，動作の正当性を検証することが必要である．

謝辞

本研究を終始御指導して下さった二木厚吉先生に感謝いたします．有益な助言を下さった渡部卓雄先生，緒方和博先生に感謝いたします．また，研究に関する議論につき合って頂いた言語設計学講座の皆様にお礼申しあげます．

参考文献

[1] Anderson, T. E.:The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Parall. Dist. Syst.*, Vol. 1, No.1 (1990), pp. 6–16.

[2] Baker, H. G.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No.4 (1978), pp. 66–70.

[3] Dijkstra, E. W., Lamport, L., Martin A. J., Scholten, C. S. and Steffens, E. F. M.: On-the-Fly Garbage Collection, An Exercise in Cooperation, *Comm. ACM*, Vol. 12, No.11 (1978), pp. 966–975.

[4] Friedman, D. P. and Wise, D. S.:The OneBit Reference Count, *BIT*, Vol. 17 (1990), pp. 351–359.

[5] 二木 厚吉, 外山 芳人：項書き換え型計算モデルとその応用. 情報処理, Vol. 24, No.2, 情報処理学会, (1983), pp. 133–146.

[6] Goguen, J., Kirchner C. and Meseguer J.:Concurrent Term Rewriting as a Model of Computation, *Proc. of a Workshop Santa Fe, New Mexico, USA, LNCS*, (1986), pp. 53–92.

[7] Halstead, R. H., Jr.: Multilisp:A Language for Concurrent Symbolic Computation, *ACM Transactions on Programming Languages and Systems*, Vol. 7. No.4 (1985), pp. 501–538.

[8] Meseguer J.:Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science, Logic, semantics and thory of programing*, Vol. 96, No.1 (1992), pp. 73–155.

[9] Ogata, K., Kondo, M., Ioroi, S. and Futatsugi, K.: Design and Implementation of Parallel TRAM, *Proc. of the Euro-Par'97, LNCS*, (1997), to appear.

[10] Ogata, K., Ohhara, K. and Futatsugi, K.: TRAM: An Abstract Machine for Order-Sorted Conditional Term Rewriting Systems, *Proc. of the Rewriting Techniques and Applications, LNCS*, Vol. 1232, (1997), pp. 335–338.

[11] 斉藤 嗣治, 緒方 和博, 二木 厚吉：並列書換えエンジンの共有メモリマルチプロセッサへの実装. 日本ソフトウェア科学会, 第14回大会論文集, (1997), pp. 417–420.

[12] Viry, P. and Kirchner, C.: Implementing Parallel Rewriting. Proc. of the International Workshop on Programming Language Implementation and Logic Programming, *LNCS*, Vol. 456, Springer-Verlag, (1990), pp. 1–15.

付録

実験で用いたベンチマークのリストをのせる

1. フィボナッチ数列

```
prim: off.
sorts: Nat Zero NzNat .
order: Zero < Nat  NzNat < Nat .
ops:   0 : -> Zero
       s : Nat -> NzNat
       add : Nat Nat -> Nat { strat: ({1 2} 0) }
       add : NzNat NzNat -> NzNat { strat: ({1 2} 0) }
       fib : Nat -> NzNat
       20  : -> NzNat
       23  : -> NzNat
       100 : -> NzNat .
vars:  X Y : Nat .
rules: add(X, 0) -> X
       add(X, s(Y)) -> s(add(X, Y))
       fib(0) -> s(0)
       fib(s(0)) -> s(s(0))
       fib(s(s(X))) -> add(fib(X), fib(s(X)))
       20  -> s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))
       23  -> s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0
```

```
                      )))))))))))))))))))))))
       100 -> s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
              s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
              s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
              s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
              s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0
              ))))))))))))))))))))))))))))))))))))))))
              ))))))))))))))))))))))))))))))))))))))))
              ))))))))))))))))))))
.
```

2. ソート

```
prim: off.
sorts: Bool Nat List .
order: .
ops:
    true : -> Bool
    false : -> Bool
    0 : -> Nat
    s : Nat -> Nat
    nil : -> List
    c : Nat List -> List
    sort : List -> List
    insert : Nat List -> List
    leq : Nat Nat -> Bool  { strat: ({1 2} 0) }

    test : -> List
    test0 : -> List
    listN : Nat -> List
```

```
    50 : -> Nat
    100 : -> Nat
    200 : -> Nat
    add100 : Nat -> Nat .

vars:
    N M : Nat
    L : List .

rules:
    sort( nil ) -> nil
    sort( c( N, L ) ) -> insert( N, sort( L ) )
    insert( N, nil ) -> c( N, nil )
    insert( N, c( M, L ) ) -> c( M, insert( N, L ) ) if leq( N, M ) = false
    insert( N, c( M, L ) ) -> c( N, c( M, L ) ) if leq( N, M ) = true
    leq( 0, M ) -> true
    leq( s( N ), 0 ) -> false
    leq( s( N ), s( M ) ) -> leq( N, M )

    test -> sort( listN( 200 ) )
    test0 -> sort( listN( 50 ) )
    listN( 0 ) -> nil
    listN( s( N ) ) -> c( s( N ), listN( N ) )
    200 -> add100(100)
    50 ->  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
           s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))
           ))))))))))))))))))))))))))))))
    100 -> s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
           s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
           s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
```

```
          s(s(s(s(s(s(s(s(s(s(0)))))))))))))))))))))))))))))))))))))))))))))))))
          )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
    add100( N )
        -> s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
          s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
          s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
          s(s(s(s(s(s(s(s(s(s(N)))))))))))))))))))))))))))))))))))))))
          ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))).
```

3．100×100 の計算

```
prim: off.
sorts: Zero NzNat Nat Bool .
order: Zero < Nat  NzNat < Nat .
ops: 0 : -> Zero
     s : Nat -> NzNat
     add : Nat Nat -> Nat
     sub : Nat Nat -> Nat
     mul : Nat Nat -> Nat
     div : Nat NzNat -> Nat

     true : -> Bool
     false : -> Bool
     ge : Nat Nat -> Bool .

vars: X Y Z : Nat
      NzX NzY : NzNat .

rules: add(X, 0) -> X
       add(X, s(Y)) -> s(add(X, Y))
       sub(X, 0) -> X
```

```
sub(s(X), s(Y)) -> sub(X, Y)
mul(X, 0) -> 0
mul(X, s(Y)) -> add(X, mul(X, Y))
div(X, NzX) -> add(s(0), div(sub(X, NzX), NzX)) if ge(X, NzX) = true
div(X, NzX) -> 0 if ge(X, NzX) = false

ge(X, 0) -> true
ge(0, s(Y)) -> false
ge(s(X), s(Y)) -> ge(X, Y) .
```