修 士 論 文

並列項書換え抽象機械：Parallel TRAM の設計と実装

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 言語設計学講座

近藤 勝
学生番号 510054

1997 年 2 月 14 日

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 本研究の目的

項書換えシステム [18] は、関数型言語をはじめ代数仕様言語の直接実行など、等式論理を基礎とした様々な分野に応用可能な計算モデルである。一方、ハードウェア技術の進歩により、高い処理能力を有した並列計算機などが、今後、非常に使い易い形で提供されると考えられ、項書換えシステムも、これら並列計算機上に実装し、並列簡約を行なえば、逐次簡約に比べ大幅な効率向上が期待できる。

そこで本研究では、計算の実行モデルを明解に表現することが可能な（項書換えの）抽象機械を、並列計算機上で並列簡約が行なえるよう拡張し、その具体的な設計を行なっていく。また、設計完了した抽象機械は適当な並列計算機上に実装して、種々の評価を行なうものとする。

本研究で並列拡張の対象とする抽象機械は、簡約化戦略に E-戦略と呼ばれる戦略を採用した TRAM [11] とし、拡張の際 逐次簡約・並列簡約をユーザが陽に指定できるよう E-戦略自体にも変更を加えるものとする。

1.2 本研究の背景

項書換えシステムは、等式で表現された論理の世界を自然な形で計算の世界に結びつける計算モデルで、代数仕様言語や関数型言語の直接実行、等式論理の定理証明など幅広い分野の処理系として導入されている。また、実際の計算機上への実装という観点から見て

も、その計算機との相性は非常に良く、ナイーブな実装は比較的容易に行なうことが可能である。しかしながら、そういった性質とは対照的に、その実行効率は一般的に悪く、実用上支障のないレベルの項書換えシステムを得るためには、さまざまな工夫（要素技術）や最適化が必要とされる。

今回拡張の対象とするTRAMは、本学 言語設計学講座の緒方助手により設計、実装された項書換えシステムで、代数仕様言語 "CafeOBJ" の実行エンジンとしてもその採用が決定されている。TRAMは実行効率を向上させるために弁別ネットに代表される多くの要素技術が盛り込まれており、また、システム全般を抽象機械の形で設計しているため、実行過程を容易にトレースでき、システムの挙動を明解に理解することが可能である。しかしながら並列計算機上で並列簡約を行なうといった最適化はいまだなされていない。

本研究の最終目標物である "Parallel TRAM" は、TRAMに盛り込まれた多くの要素技術と、実行過程を明確に把握できるという抽象機械の特徴をそのまま受け継ぎ、なおかつ、並列計算機上で並列簡約を可能にした並列項書換えシステムである。このParallel TRAMが完成し、効率の向上が実現できれば、単にCafeOBJの並列化およびその実行効率の向上 という話だけにとどまらず、様々な分野の処理系として期待されるであろう。

1.3 本論文の構成

まず第２章において、本研究の土台となる項書換えシステムの基本概念と、関連研究について解説を行なう。また第３章では、今回並列拡張の対象としたTRAMについて、そのTRAMが採用しているE-戦略と、TRAMの基本的な動作について解説を行なう。基盤を同じくするTRAMの動作を把握するのは、本研究を理解する大きな助けになると考える。第４章では、並列計算機上で並列簡約を実現する "Parallel TRAM" の具体的な設計を行なっていく。まず、逐次簡約しか指定できなかったE-戦略に、引数項の並列簡約が指定できるシンタックスを追加し、並列E-戦略なる戦略を新たに定義する。そして、この並列E-戦略に基づいて並列簡約を行なうためのメカニズムを簡単に解説した後、この並列簡約を実現するための抽象命令を順番に定義していく。第５章では、４章で設計した抽象機械をどのようにして並列計算機上（LUNA-88k2）に実装したのか、その実装時に行なった種々の決定事項について説明を行ない、続く第６章では、実装したParallel TRAMにいくつかのベンチマークを実行させ、引数項の並列簡約によりどの程度の効率改善が実現できたのかを評価している。最後の第７章では、本研究のまとめを行ない、さらに今

後の課題について触れる。なお、付録として、TRAM/Parallel TRAM どちらの抽象機械にも共通に使用される抽象命令の定義と、評価で用いたベンチマークの定義 を添付している。

第 2 章

基本概念と関連研究

2.1 項書換えシステム

項書換え系とは、項の集合と書換え規則の集合の対で定義され、与えられた項を書換え規則を用いて書き換えていき、最終的にそれ以上書き換えられない項を計算結果と見なす計算モデルのことである。項書換えシステム（項書換え系）は、等式論理の定理証明、代数仕様言語や関数型言語の直接実行など、現在でも等式論理を基礎とした幅広い分野への応用がなされている。本節ではこの項書換えシステムの基本的な事項について、いくつかの定義を行なう。

階数 (アリティ)： 階数 (アリティ) とは、演算子の引数の個数のことである。

定数： 定数とは、階数が 0 の演算子のことである。

項： 項とは、以下の帰納的な定義により定められるものである。

1. 定数、変数は項である。
2. $t_1, t_2, \cdots, t_n$ が項で、f が階数を n とする関数なら、$f(t_1, t_2, \cdots, t_n)$ もまた項である。

定数項： 定数項とは、変数を１つも含まない項のことである。

部分項： 項 t の部分項とは、以下の条件から定められるものである。

1. t が定数または変数のとき、部分項は t それ自身となる。
2. t が $f(t_1, t_2, \cdots, t_n)$ のとき、部分項は $t_1, t_2, \cdots, t_n$ およびそれらの部分項と t 自身となる。

書換え規則： 書換え規則 $s \to t$（ただし s, t は項）は、以下の条件を満たすものである。

1. s は変数ではない。
2. t に出現する変数は、s にも出現しなければならない。

この書換え規則の s のことを**左辺**、t のことを**右辺**という。また、書換え規則の左辺に同じ変数が高々１回しか出現しないとき、その規則を左線形という。

項の照合（パターンマッチ）： 変数を含む項 t が、定数項 t' と照合する（マッチする）とは、ある適当な変数置換 σ が存在し、この変数置換を t に適用させて得られる定数項が t' と等しくなることをいう。

項の簡約（書換え）： 定数項 t が、書換え規則 $l \to r$ の左辺 l と変数置換σ によって照合するような部分項 t' を含む時、t の部分項 t' を σr に置き換えて得られる項を u とすると、t は u に簡約される または t は u に書換えられるという。

リデックス： 項書換え系 R において、その書換え規則の左辺と照合する項のことを R のリデックスという。

正規形： 正規形とは、その部分項にリデックスを１つも持たない項のことで、項書換え系の計算結果となる。

簡約化戦略（書換え戦略）： 項の簡約を行なうとき、どのような順序でリデックスの書換えを行なうか その順番を決定する手続きのことを簡約化戦略（書換え戦略）という。代表的な簡約化戦略として、**最左最外戦略**（最も左側で最も外側のリデックスから書換えを行なう）や、**最左最内戦略**（最も左側で最も内側のリデックスから書換えを行なう）などがある。

停止性： 項書換え系 R において無限の簡約列が存在しないとき、その項書換え系 R は停止性を満たすという。停止性を満たす項書換え系は、どの様な順番で書換えを行なっても必ず正規形を求めることができる。

合流性： 項書換え系 R の任意の項 t を 0 回以上書換えて t_1、t_2 という項が得られたとき、さらにこの２つの項を 0 回以上書換えて t_3 という等しい項を得ることができる場合、その項書換え系 R は合流性を満たすという。合流性を満たす項書換え系は、正規形をもつならばその値は必ず唯一に定まるという性質を持つ。

2.2 関連研究

項書換えシステム（項書換え系）は、様々な分野への応用が可能という性格上、理論・実装の両面にわたり盛んに研究が行なわれている分野の１つである。特に項書換えの処理系を、適度に抽象度を上げた抽象機械の形で設計するという研究は、Kamperman らが設計した ARM[9] を代表に、近年数多く行なわれている。

ARM(Abstract Rewrite Machine) は、代数仕様言語の処理系を意識して設計された抽象機械で、わずか４種の命令と、１つのヒープ領域、２つのスタック領域から構成されている。ARM で用いられる４種の抽象命令は、規則とのパタンマッチを行なう select、変数の束縛内容が等しいか調べる check、結合則を持ったリストのパタンマッチを行なう for、規則の右辺の項を生成する proceed の４つで、これらの抽象命令は C のソースコードに変換されて実行されるため、毎秒平均 80000 回の書換えを行なうという非常に効率の良い処理系に仕上がっている。

また、項書換えシステムの応用分野である関数型言語の処理系をターゲットに設計された抽象機械として Peyton Jones の行なった STG(Spineless Tagless G-machine)[16] がある。この STG は、ソースの関数型言語である Haskell を、シンタクティクシュガーの除去、型チェックおよび関数の重複定義の解析などを済ませた Core 言語なる言語に一度コンパイルし、それを抽象機械の動作を行なう STG 言語（抽象言語）に変換するというステップを踏む。STG 言語はそれ自体が関数型言語になっており、その操作的意味を状態遷移システム、つまり状態遷移に関する書換え規則を適用して抽象機械の処理を表現する、という形で与えているため、抽象機械のセマンティックスを明確に理解できるようになっている。

一方、項書換えシステム（グラフ書換えシステム）を並列に処理するという研究も今までに数多く行なわれてきた研究分野で、Peyton Jones の実装した GRIP[15] などが例として挙げられる。この論文では、実行効率の向上に重点を置いて話を進めており、並列簡約を効率良く行なうためのポイントとして、マシンの負荷配分が重要であると言及している。もともとこの GRIP も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、グラフリダクションのような高い並列性を示す処理を並列計算機で行なわせる場合、各プロセッサに割り振る処理の粒度を十分大きくし、プロセッサへの負荷をかけすぎないでしかもプロセッサを常に busy 状態にしておくことが必要であると説明している。また、簡約処理の分配を管理するスケジューラーについても論じており、LIFO(深さ優先) のスケジューリングはメモリの使用量を最小にし、FIFO(幅優先) のスケジューリングは並列性の拡大を最大にすると述べている。最も理想的なのは、こういったスケジューリングを動的に切換えることであるが、実は複雑なスケジューリングを実現するにはそれだけ多くのオーバーヘッドを必要とし、単純なスケジューリングだけでも十分な効果が得られると結論付けている。

また、項書換え系と直接関係はしないが、もともと逐次処理しか行なえなかったソフトウェアを、並列処理できるように拡張するといった研究もいくつか行なわれている。R.H.Halstead Jr. が行なった Multilisp[14] は非常に有名で、この Multilisp の中で並列処理を実現する主要な命令（概念）”future” は、その後の並列・並行システムにおいても使用されるなど非常にメジャーな概念である。(`future X`) は、式 `X` の将来値 (future) を即座に返し、`X` の評価を並行して行なわせる。そして`X` の評価が完了したら、先ほどの将来値がその評価結果と置こ換えられる。つまり、将来値 (future) は最初 ”未定値” で処理が行なわれるわけだが、評価が完了し値が確定した時点で ”決定値” に変換されるわけである。future の値が確定しないと計算が行なえないような演算は、値が確定するまで処理を中断させられることになるが、代入や引数渡しなどの多くの演算は値が確定している必要はなく、従って future のまま何の問題もなく処理を進めることができる。Multilisp ではこの future を用いることによって驚くほど多くの並列性が抽出できることを示しており、また future を含む並列計算実現の基本構成として、プロセッサの数だけインタープリタを複製する方針を取っている。

以上、本研究と関連する研究をいくつかピックアップして紹介を行なった。なお、本研究と最も深く関係し、今回の並列拡張の対象となる TRAM については次章で説明する。

第 3 章

項書換え抽象機械 TRAM

TRAM は、パタンマッチ処理部に**弁別ネット**（3.2.1 節）を用いるなど、効率の中でも特に処理速度の向上に重点を置いて設計・実装された項書換え抽象機械で、通常の単一プロセッサ上で動作することを想定している。また、TRAM 内部において項は、すべて抽象機械命令列で表現されており、従って、書換えの度に抽象命令列を変更していく”自己改変的” な項書換えシステムである。簡約化戦略には、**E-戦略**（3.1 節）と呼ばれる戦略を採用し、より緻密な書換え制御ができるようになっている。

3.1 E-戦略（Evaluation-Strategy）

E-戦略とは、演算子毎に簡約の順番を指定することができる戦略で、lazy、eager などの簡約方法をローカルに制御することが可能となる。簡約順序の指定は数列を用いて行なわれ、この数列の各要素は、

0：全体項簡約

n：n 番目の引数の簡約 ($0 < n \leqq$ 引数の個数)

を表している。また、数列の順番はそのまま簡約させる順番に対応している。

<例>

```
if(true, X, Y) -> X.
if(false, X, Y) -> Y.
```

if の指定戦略 : (1 0)

上記 `if` の場合、まず第 1 引数を簡約し、その後 全体項を簡約することを指定している。もしこの `if` を E-戦略でなく最内戦略（eager な戦略）で簡約したとすると、`X` もしくは `Y` の簡約が無駄なものになってしまう。つまり、E-戦略を用いることで、演算子`if`についてのみ効率の良い lazy な簡約を行なうことが可能になったわけである。

3.2 TRAM の概要

TRAM の構成は、図 3.1のようになる。

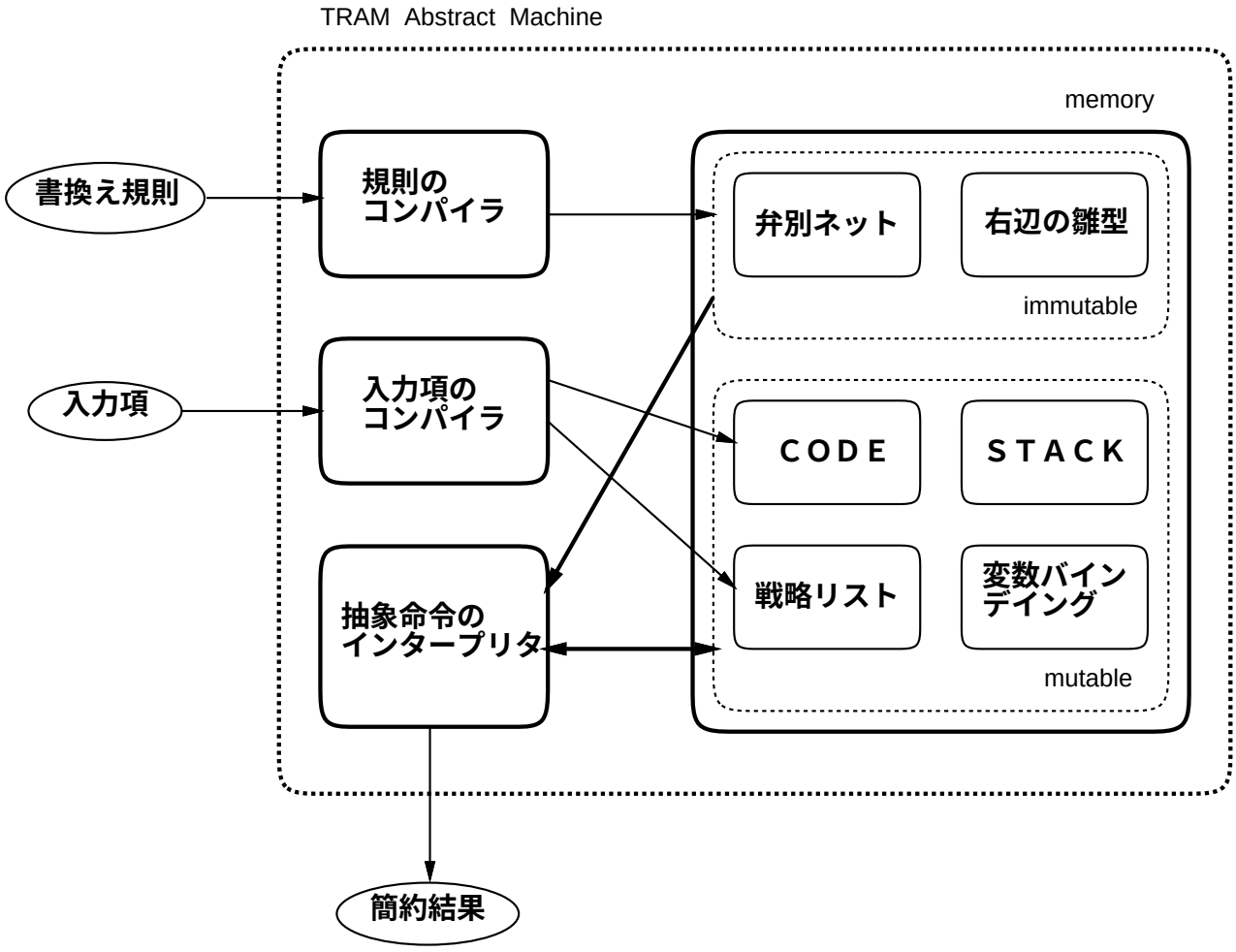

図 3.1: TRAM の構成

図 3.1からも明らかなように、TRAM は３つの処理ユニットと、６つのメモリ領域（う

ち４つは簡約の際内容が動的に変化し、残りの２つは変化しない）から構成されている。

それでは図 3.2で定義されるフィボナッチ数列を例に、各部の動作について説明していくことにする。

$$plus(X,\ 0) \longrightarrow X \tag{3.1}$$

$$plus(X,\ s(Y)) \longrightarrow s(plus(X,\ Y)) \tag{3.2}$$

$$fib(0) \longrightarrow 0 \tag{3.3}$$

$$fib(s(0)) \longrightarrow s(0) \tag{3.4}$$

$$fib(s(s(X))) \longrightarrow plus(fib(s(X)),\ fib(X)) \tag{3.5}$$

$$plus \text{ の指定戦略} \quad : \quad (1\ \ 2\ \ 0) \tag{3.6}$$

$$fib \text{ の指定戦略} \quad : \quad (1\ \ 0) \tag{3.7}$$

$$s \text{ の指定戦略} \quad : \quad (1\ \ 0) \tag{3.8}$$

$$0 \text{ の指定戦略} \quad : \quad (0) \tag{3.9}$$

図 3.2: フィボナッチ数列の定義

3.2.1 書換え規則、入力項のコンパイル

ここはいわゆる前処理のフェーズで、このフェーズにより、簡約に必要な環境が整えられる。

書換え規則のコンパイラは、規則の左辺を**弁別ネット**に、規則の右辺を**右辺の雛型**にコンパイルし、それぞれの領域に格納する。

同様に入力項のコンパイラは、入力項を**マッチングプログラム（CODE）**と**戦略リスト**にコンパイルし、それぞれの領域に格納する。

弁別ネット

弁別ネットとは、最外項シンボルをキーにして分岐したパタンマッチ用の木構造のことで、TRAM に組み込まれた主要な要素技術の１つである。この弁別ネットを用いること

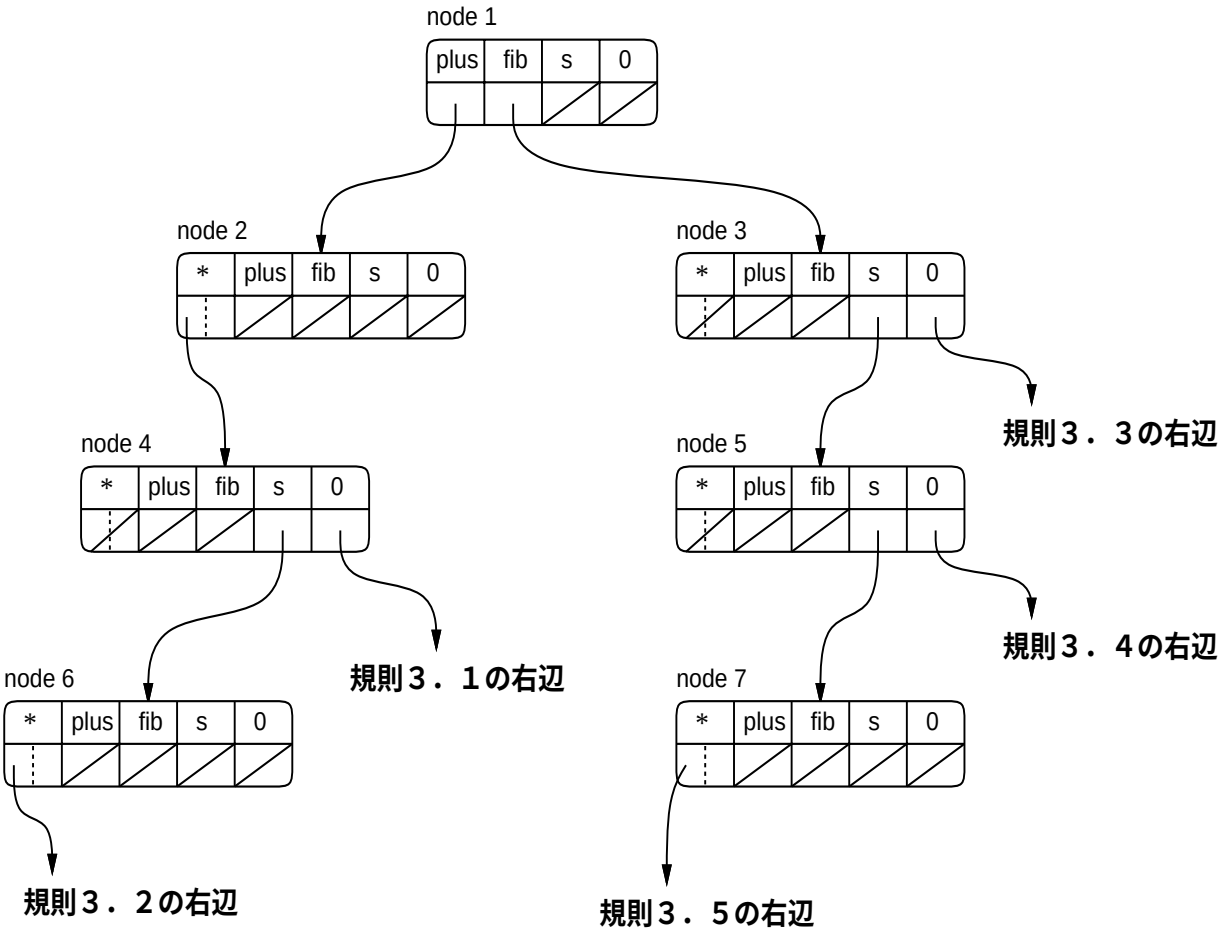

図 3.3: フィボナッチ数列の定義から生成される弁別ネット

により、マッチする規則を効率良く検索することが可能となる。図 3.3に示すのは、フィボナッチ数列の定義（図 3.2）の左辺をコンパイルして得られる弁別ネットである。

各ノードには、規則定義の中で使われる全てのシンボルが登録されており、書換え対象となる項中に現れるシンボルを、弁別ネットのルートから順番通りにたどっていけば、適用可能な規則の右辺に到達できるようになっている。ノード中の * は変数を表しており、このスロットを通過するときには、自動的に変数への束縛が行なわれる。（この変数束縛は、束縛する値を**変数バインディング**領域に格納し、変数スロットからこの値へポインタを張ることによって行なわれる。変数スロットが２つに分割されているのはこのためで、図中 変数スロットの空き領域がこのポインタのための領域である。）

例えば、$plus(s(0),\ s(0))$ という項に適用可能な規則を検索する場合、

1. 最初のシンボルは $plus$ なので、ルートノード（node1）の $plus$ のスロットをたどり node2 へ移る。

2. 次のシンボルは s であるが、s のスロットはNULL なので、変数スロットをたどり node4 へ移る。このとき、変数 X に $s(0)$ を束縛する。（変数 X は $plus$ の第１引数に対応しているので、s ではなく $s(0)$ を束縛する。）

3. 次のシンボルは s なので、s のスロットをたどり node6 へ移る。

4. 次のシンボルは 0 であるが、0 のスロットはNULL なので、2. と同様にして変数 Y に 0 を束縛し、規則 3.2 の右辺に到達する。

以上のような過程を経て、最終的に適用可能な規則の右辺として

$$right\ hand\ side = s(plus(X,\ Y)),\quad where\ \ X = s(0),\ \ Y = 0$$

という結果を得る。

マッチングプログラム

マッチングプログラムとは、上記弁別ネットの節で説明した、適用可能な書換え規則の検索と変数束縛を弁別ネットを用いて実際に行なう抽象命令のことである。マッチングプログラムは構造的に項と等価なもので、TRAM に入力された項は全てこのマッチングプログラムの形で蓄えられる。TRAM 内部においてマッチングプログラムは項を表現しているだけでなく、これをそのままインタープリタで実行すれば、規則の検索と変数束縛を行なえるという利点を持ち、規則検索の度に毎回項をトラバースする手間を省くことができる。TRAM が項をマッチングプログラムの形で取り扱うのもそういった理由による。また、当然のことながら書換えが行なわれる度にマッチングプログラムは動的にその内容を変化させ、それ故 TRAM は ”自己改変的” なシステムとなる。

図 3.4に示すのは、項 $plus(s(0),\ s(0))$ をマッチングプログラムへ変換した場合の変換例である。（図中マッチングプログラムの割り付けられたアドレス値に特別な意味は無い。実際にはマッチングプログラム領域の未使用部分に順次割り付けられていく。）

命令 `match_symbol` は、現在着目している弁別ネットのノードから、オペランドに指定されたシンボルのスロットをたどって次のノードに遷移する命令で、必要に応じて変数

$plus(s(0),\ s(0))$ $\xrightarrow{\text{変換}}$

アドレス	オペレータ	オペランド
100000:	match_symbol	”plus”
100001:	call	100003
100002:	call	100006
100003:	match_symbol	”s”
100004:	call	100005
100005:	match_symbol	”0”
100006:	match_symbol	”s”
100007:	call	100008
100008:	match_symbol	”0”

図 3.4: 項 $plus(s(0),\ s(0))$ のマッチングプログラム

の束縛も行なう。また命令 `call` は、オペランドの番地に制御を移し、その処理が終了したら元の番地に戻ってくるサブルーチンコール命令である。

図 3.4からも明らかなように、`match_symbol` 命令をノードに、`call` 命令をエッジに対応させれば、これは項と等価な木構造になることが分かる。

戦略リスト

戦略リストとは、簡約順序を制御するために用いられるリスト構造のデータのことで、意味的には E-戦略でユーザーが指定した指定戦略と全く等価なものである。

例えば項 $plus(s(0),\ s(0))$ が入力項として与えられ、それぞれの演算子には図 3.2に示される戦略が指定されていた場合、戦略リストは次のような要領で生成される。

1. 与えられた項をマッチングプログラム（図 3.4）に変換し、実際の項を（部分項も含めて）マッチングプログラムにおける`match_symbol` 命令のアドレスと対応づける。

2. 演算子毎に指定された指定戦略をもとにして、全体項、部分項の簡約に順序を付ける。

3. 2. で付けた順序通りに `match_symbol` 命令のアドレスを並べ、リストにする。

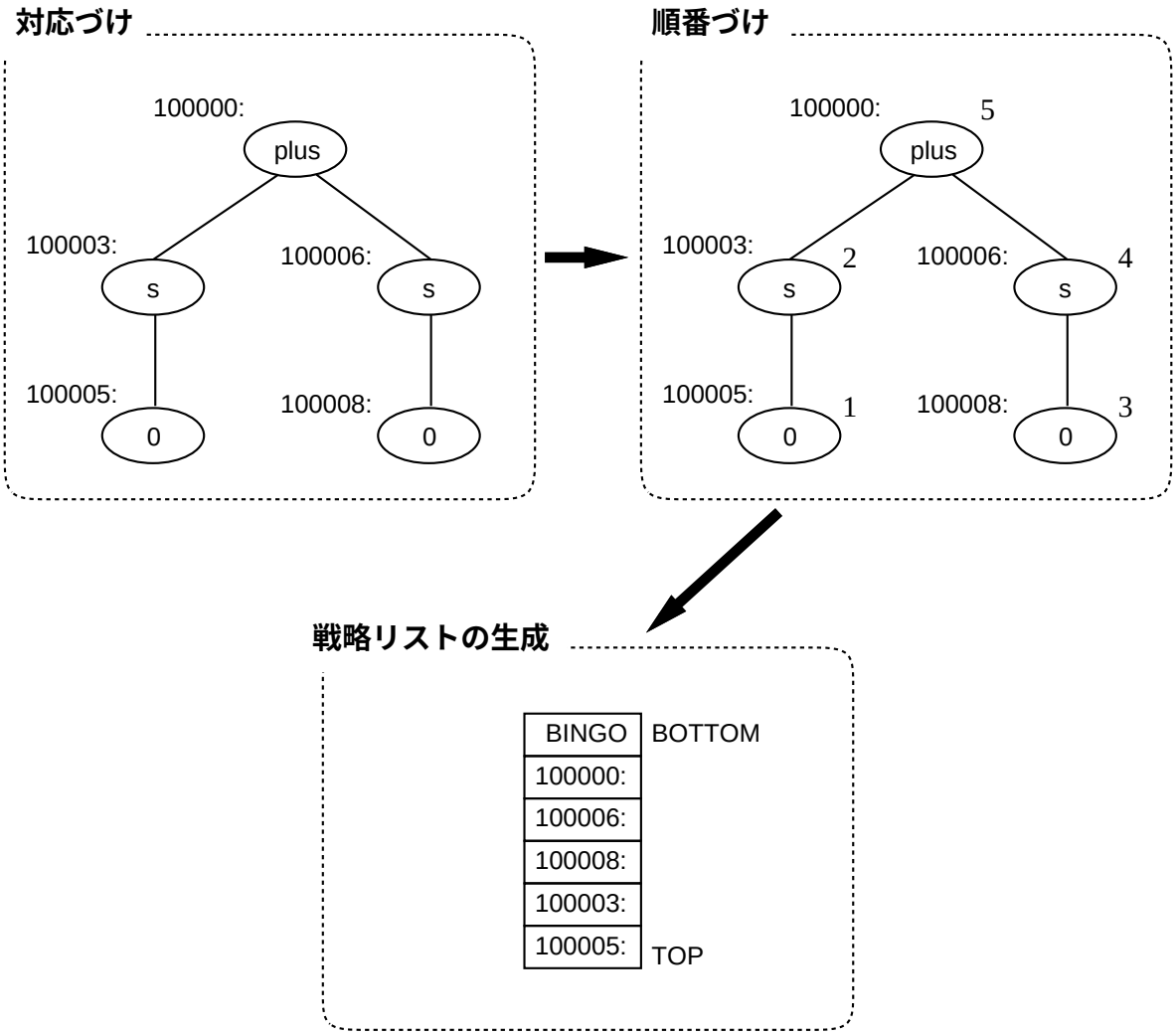

図 3.5: 項 $plus(s(0),\ s(0))$ の戦略リスト

戦略リスト生成の様子を図に表すと図3.5のようになる。

抽象命令のインタープリタは、この戦略リストの順番に従ってマッチングプログラムを実行し、書換えを進めていく。戦略リスト中に現れる "BINGO" は、ユーザー指定の戦略が終了したことを表しており、インタープリタが簡約作業を終了する終了条件になっている。

注意 1 : 実際の戦略リストは、3つのデータを1要素とした3つ組のリスト構造をしている。従って図3.5における戦略リストのそれぞれの要素間には、さらに2つのデータが挿入されることになる。この2つのデータの意味、役割については、3.2.2 節

で述べる。

注意 2 : 定数、構成子、既に正規形になっている項 など、事前に書換えの必要がないと分かる項に対してわざわざ書換えを行なうのは非効率である。TRAM でもこのような項に対応する戦略リストは生成しないという最適化を行なっている。従って図 3.5の戦略リストも、正確にいうと、シンボル `plus` に対するマッチングプログラムのアドレスのみとなる。ここでは、戦略リストの意義をきちんと理解してもらうために、最適化前の戦略リストを例にして説明を行なった。

右辺の雛型

先にも述べたように、TRAM では項をマッチングプログラムで表現する。従って書換えが行なわれ入力項の構造が変化する（部分項がマッチした規則の右辺で置き換えられる）と、それにともないマッチングプログラムも変化させなければならない。また同様に戦略リストも再構成しなければならない。”右辺の雛型” とは、その置き換えられるマッチングプログラムの雛型と、再構成する戦略リストの雛型を表している。

これら２つの雛型生成は、基本的にマッチングプログラムおよび戦略リストの生成と同じである。唯一異なる点は、

- 置き換えられるマッチングプログラムがどのアドレスに割り付けられるか分からない。
- 規則に変数を含む場合、その変数に何が束縛されるか分からない。

という２つの理由から、アドレスがオフセットあるいは弁別ネット上の変数スロットになっているという点である。

図 3.6に示すのは、フィボナッチ数列の定義（図 3.2）における規則 3.2 を右辺の雛型にコンパイルした結果である。

3.2.2 抽象命令インタープリタ

書換え規則のコンパイル、入力項のコンパイルが終了すると、抽象命令を解釈・実行するインタープリタが本格的な書換え作業を開始する。書換えは、次に示す非常にシンプルな抽象命令プログラムで実現されている。

```
  1:      LOOP: init_trs
  2:            e_next
  3:  GO_AHEAD: go_ahead
  4:            select
  5:            e_rewrite
  6:            goto LOOP
-------------------------
  7:     BINGO: bingo
```

このプログラムは抽象機械のインタープリタ上で次のように動作し、項の書換えを実際に行なっていく。

1. 命令 init_trs で TRAM を初期化する。(ここでいう初期化とは、1回の書換えに必要な大域レジスタなどの初期化を指す。)

2. 命令 e_next で戦略リストの先頭からマッチングプログラムのアドレスを1つ取り出し、そのアドレスのマッチングプログラムに制御を移す。取り出したアドレスが BINGO の場合は、命令bingo を実行する。

3. 2. 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出し go_ahead に戻ってくる。適用可能な規則が見つかった場合、この go_ahead 命令でバックトラック (注1) を起こし、他の適用可能な規則を全て探し出す (注2)。マッチングプログラムから戻ってきたときに適用可能な規則が1つもなかった場合は、go_ahead 命令で LOOP にジャンプする。

4. 命令 select で適用可能な規則の中から規則を1つ選択する。

5. 命令 e_rewrite により 2. のマッチングプログラムを 4. で選択した規則の右辺で置き換え (注3)、戦略リストを再構成する (注4)。

6. 命令 goto LOOP で LOOP にジャンプする。

7. 命令 bingo が実行されると、そのときのマッチングプログラムが簡約結果として返される。

(注1) 弁別ネットをたどるパターンが複数パターンある場合、分岐する直前の情報 (分岐した弁別ネット上の位置、マッチングプログラムの位置) を **STACK** 領域に積んでいくため、このバックトラックが可能となる。

（注 2） 適用可能な規則を全て探し出すということはつまり、非決定的な書換えにも対応しているということである。

（注 3） 規則の右辺は、変数の値を具象化したあと、マッチングプログラムの空き領域に割り付けられる。しかしこれだけでは、規則の右辺は孤立した状態になってしまい、左辺を右辺に置き換えたことにはならない。そこで、前節において説明を保留した 戦略リスト３つ組データのうちの２つ目のデータを用いる。ここには、１つ目のデータ（マッチングプログラムの match_symbol 命令）を実際に呼び出す call 命令のアドレス（親アドレス）が格納されていて、この call 命令のオペランドを、新しく割り付けたマッチングプログラムの開始アドレスに書き換えれば、左辺→右辺の置き換えを行なうことができる。

（注 4） 例えば、（1 0 2 3 0） という戦略指定の２番目の要素 0（全体項簡約）で書換えに成功し全体項の構造が変化したとすると、戦略リストを再構成するときに（2 3 0）に対応する戦略リストは必要なくなる。こういう場合には、戦略リスト３つ組データの３番目のデータを用いて不要な戦略リストを取り除く。つまり、全体項簡約に対応する戦略リストには、書換えに成功した場合に戦略リストのどの要素までを無効にするか、という情報を３つ目のデータとして持たせるわけである。

以上が TRAM の概要である。本研究の目的は Parallel TRAM の設計および実装であるため、今回は TRAM の最低限の動作と、Parallel TRAM を理解するのに必要と思われる箇所を中心に説明した。従って、今回説明を省略した部分もかなりある。実際の TRAM は条件付きの書換え規則も扱えるし、順序ソートへのサポートもなされている。TRAM 関して更に詳しく知りたい読者は、参考文献 [10][11] [12][13] を参照していただきたい。

規則 3.2 の右辺 $s(plus(X,\ Y))$ $\xrightarrow{\text{変換}}$

オフセット	オペレータ	オペランド
0:	match_symbol	"s"
1:	call	2
2:	match_symbol	"$plus$"
3:	call	Node X
4:	call	Node Y

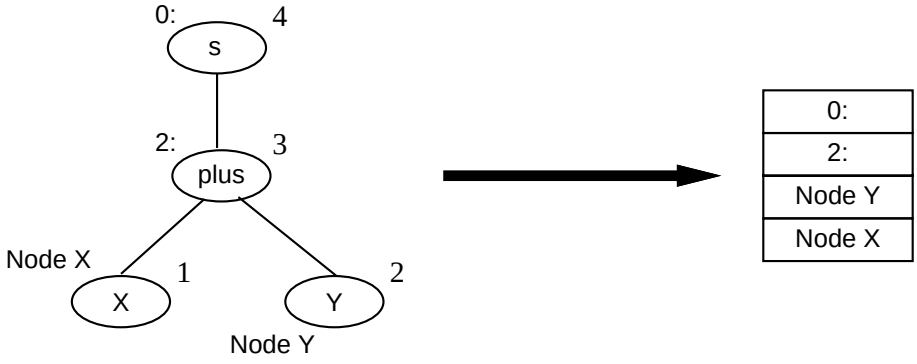

図 3.6: 規則 3.2 の "右辺の雛型"

第 4 章

Parallel TRAM の設計

この章では、TRAM が並列計算機上で動作することを想定して実際に並列簡約が行なえるよう拡張し、最終目標物である ”Parallel TRAM” の具体的な設計を行なっていく。

一般に、項の書換えを並列に行なわせるといった場合、大きく分けて ”非決定的”な書換えを並列に行なわせる場合と、”決定的”な書換えを並列に行なわせる場合の２つの可能性が考えられる。”非決定的”な書換えというのは、１つのリデックスに対して複数の書換え規則が適用できるというもので、もちろん現在の TRAM でもそういった書換えへのサポートはなされている（3.2.2 節、注 2 参照）。しかしながら、その計算処理を項書換え系でモデル化することができる対象のほとんどは、CafeOBJ で記述される諸仕様も含めて、”決定的” な書換えだけで十分な場合が多く、さらに”非決定的”な書換えを含む項書換え系でさえも、その大部分は”決定的”な書換えで占められているのが普通である。つまり TRAM の実行効率の向上という観点から見た場合、”非決定的”な書換えを並列に行なわせる方法は、それほどの効率向上は期待できず、従って Parallel TRAM において ”非決定的” な書換えは、特に並列簡約の対象とはしない。

そこで今回は、並列性が高く、しかも少ないオーバーヘッドで並列性の抽出が可能な、各演算子における**引数項**の簡約を並列に行なわせるものとする。この引数項の並列簡約を実現する戦略としてまず第一に考え付くのは、全ての引数項を自動的に並列簡約させるという戦略である。しかし、超並列計算機などの大規模な計算機をターゲットとして、計算機依存の特別な処理系を開発するといった場合でない限り、プロセッサ数や記憶領域などの資源は非常に限られている（少ない）場合が多く、資源量に対して並列に簡約させるプロセス数の方がはるかに多く生成されてしまい、逆に効率の悪化を招く恐れがある。

つまり並列計算機の性能を最大限に活用するには、並列簡約の指定をユーザに開放し、並列性の拡大を制御する必要がある。従って Parallel TRAM において、引数項の並列簡約を行なわせる場合は、それを明示的に指定する方針を取ることにする。これは、E-戦略がユーザの意志を非常に尊重した戦略であるという思想ともうまく合致すると考えられる。そこで、Parallel TRAM の設計に入る前に、E-戦略に並列簡約を指定できるよう若干の変更を加えることにする。

4.1 E-戦略の並列指定

引数項の並列簡約を実現するため、E-戦略による戦略指定の構文を図 4.1のように定義し直す。

アリティ n の演算子 f に対し、

$$
\begin{aligned}
<StrategyDefinition> &::= \varepsilon \mid \text{"\{"}\ \text{"strat :"}\ <UserDefinedStrategy>\ \text{"\}"} \\
<UserDefinedStrategy> &::= \text{"("}\ \text{")"} \mid \text{"("}\ <ReductionSeq>\ <Whole>\ \text{")"} \\
<ReductionSeq> &::= \varepsilon \mid <ReduceElement>\ <ReductionSeq> \\
<ReduceElement> &::= <Whole> \mid <Arg> \mid <ParallelReduction> \\
<ParallelReduction> &::= \text{"\{"}\ <ArgReductions>\ \text{"\}"} \\
<ArgReductions> &::= \varepsilon \mid <Arg>\ <ArgReductions> \\
<Whole> &::= \text{"0"} \\
<Arg> &::= \text{"1"} \mid \text{"2"} \mid \ldots \mid \text{"}n\text{"}
\end{aligned}
$$

図 4.1: 並列指定を加えた E-戦略の構文

この構文で記述された戦略は、図 4.2のようなアルゴリズムで簡約されるものとする。つまり、次のような意味を持つことになる。

- $<Whole>$ は全体項簡約を表し、$<Arg>$ は引数項簡約を表している。

- "(…)" で囲まれた要素は、左から逐次に簡約される。(逐次指定)
- "{…}" で囲まれた要素は、並列に簡約される。(並列指定)
- 構文からも明らかなように、並列指定できるのは引数項簡約のみで、全体項簡約は並列指定できない。
  (全体項簡約を並列指定できないようにしたのは、非常に特別な場合を除き、全体項簡約には、引数項簡約の結果が必要となるからである。)
- "{…}" で並列簡約を指定した場合、並列に簡約させた全ての簡約が終了するまで次の簡約に移行しない。
  (こうしておかないと、"(…)"で囲まれる要素は逐次簡約、"{…}"で囲まれる要素は並列簡約、という原則が破られてしまう。)
- ただ1つの引数項簡約を並列に指定した場合、その並列指定は無効(逐次簡約)になる。また、ε(空)を並列指定した場合には、その記述自体が無視される。
- $<UserDefinedStrategy>$ に "( )" を指定した場合、その演算子に関する簡約は引数項を含めて一切行なわれない。
- $<StrategyDefinition>$ に ε を指定した場合、その演算子には、
  `{ strat: (1 2 ... n 0) }` なる戦略(最左最内戦略)が指定されたものと見なされる。

<例>

```
f(X, Y, Z, W) -> …
f の指定戦略 : (1 {2 3} {} {4} 0)
```

この場合、まず第1引数を簡約し、続いて第2〜第3引数を並列に簡約する。この並列簡約が全て終了したら、第4引数を簡約し、そして最後に全体項を簡約する。途中に出てくる "{}" と、第4引数の並列指定は無視される。

```
procedure
REDUCE( t: 簡約させたい項 ) {
  f ← t の最外演算子;
  if ( f の <StrategyDefinition> = ε ) {
    f の <StrategyDefinition> ← "{ strat: (1 2 ... n 0) }";
                                /* ただし n は f のアリティ数 */
  }
  while ( f の <ReductionSeq> ≠ ε ) {
    m ← <ReductionSeq> の先頭要素;
    switch ( m ) {
      case <Whole> :
        if ( t 全体にマッチする規則 r がある ) {
          t' ← t を r の右辺で置き換えた結果;
          return REDUCE( t' );
        }
        <ReductionSeq> から m を取り除く;
      case <Args> :
        t' ← t の m 番目の引数項;
        t ← t の m 番目の引数項を REDUCE( t' ) で置き換えた結果;
        <ReductionSeq> から m を取り除く;
      case <ParallelReduction> :
        if ( <ArgReductions> = ε ) {
          <ReductionSeq> から m を取り除く;
        } else if ( <ArgReductions> = <Arg> ) {
          <ReductionSeq> から m を取り除く;
          <ReductionSeq> の先頭に <Arg> を追加する;
        } else {
          n_1, ..., n_i ← <ArgReductions> のそれぞれの要素;
            1. t_j ← t の n_j 番目の引数項;
            2. t ← t の n_j 番目の引数項を REDUCE( t_j ) で置き換えた結果;
          上記 1. 2. を j = 1 から j = i まで並列に行なわせる;
          <ReductionSeq> から m を取り除く;
        }
    }
  }
  return t;
}
```

図 4.2: 簡約動作のアルゴリズム

この構文定義に、さらに次の制約を付け加える。

1. $<Whole>$（全体項簡約）は、重複しても良いが、連続して現れた場合は１個の 0 と見なされる。

   $(\{1 \quad 2\} \quad \underline{0} \quad \underline{0} \quad \{3 \quad 4\} \quad 0) \quad \rightarrow \quad (\{1 \quad 2\} \quad \underline{0} \quad \{3 \quad 4\} \quad 0)$

2. $<Arg>$（引数項簡約）は、逐次指定・並列指定の区別を問わず、重複してはならない。重複した場合、２回目以降の指定は無視される。

   $(\underline{1} \quad \{\underline{1} \quad 2 \quad 3\} \quad 4 \quad \underline{1} \quad \{5 \quad \underline{1} \quad 6\} \quad 0) \quad \rightarrow \quad (\underline{1} \quad \{2 \quad 3\} \quad 4 \quad \{5 \quad 6\} \quad 0)$

以上、この構文定義と制約により規定され、図 4.2のアルゴリズムで簡約される戦略を「**並列 E-戦略**」とし、Parallel TRAM では、この並列 E-戦略を簡約化戦略として採用することにする。

4.2 Parallel TRAM の構成

Parallel TRAM は、前節で定義した「並列 E-戦略」に基づく並列簡約を、実際に実現する項書換え抽象機械である。この Parallel TRAM の簡約処理の動作を大まかに推測してみると、

1. 引数項の簡約を複数のプロセッサ上で並列に行なう。
2. それら複数のプロセッサ上に点在する引数項簡約の結果を用いて、あるプロセッサが全体項の簡約を行なう。

という２つのフェーズでその大半が占められると考えられる。つまり、異なったプロセッサ間でのデータの受渡しや参照が頻繁に行なわれることになる。この Parallel TRAM を完全な分割メモリ型の並列計算機上に実装した場合、そのデータの授受にかかるオーバーヘッドは多大なものになると容易に想像でき、従って Parallel TRAM が動作する基本環境として、共有メモリ型の並列計算機を選択することにする。共有メモリ型の並列計算機は、各プロセッサに共通のアドレススペースを持ち、異なるプロセッサ間でのデータの授受や参照を、非常に小さなオーバーヘッドで行なうことが可能となる。

共有メモリ型並列計算機（マルチプロセッサ）上で動作する Parallel TRAM の構成は、lisp をマルチプロセッサ向きに進化させた multilisp[14] の例を参考にして、インタープリタを複製する方式をとることにする。その構成は図 4.3のようになる。

それぞれのプロセッサには、抽象命令を解釈・実行するインタープリタと、簡約の際 動的に内容が変化する４つの領域（CODE、STACK、戦略リスト、変数バインディング）を複製し、抽象機械のレベルで仮想的に割り付ける。（この１まとまりを ”プロセスユニット” と呼ぶことにする。）また、動的に内容の変化しない読み出し専用の領域（弁別ネット、右辺の雛型）は、これら以外の共通に参照されるデータとあわせ、グローバル領域に格納される。このグローバル領域に現れる ”プロセスユニット状態” というのは、各ユニットの動作状態（busy、idle、etc..）を保持しておくテーブルのことで、このテーブルにより、プロセスユニットの動作状態を見ながら実際に他ユニット上で並列簡約を実行させるかどうか決定するという、非常に簡単なスケジューラーを抽象機械に組み込むことが可能となる。

> **注意**：動作環境として共有メモリ型並列計算機を想定しているため、構成上ローカル・グローバルの区別がなされていても、データアクセスの効率は全く変わらない。つまり各ローカル間のデータアクセスも、グローバル領域へのアクセスと全く同様に行なうことができる。

また、メインとなる特別なプロセスユニット（図 4.3におけるプロセスユニット 1）を１つ設け、このユニット上で、規則のコンパイル、入力項のコンパイルといった簡約に入る前の前処理を全て逐次に行なうものとする。

4.3 並列簡約のメカニズム

それでは、並列 E-戦略によりユーザーが指定した並列簡約を、Parallel TRAM がどうやって実現していくのか、そのメカニズムについて説明していくことにする。

弁別ネットの構築と、右辺の雛型および入力項に対するマッチングプログラムの生成は、TRAM の場合の生成と全く変わらない。ただ戦略リストの生成だけは、E-戦略を並列 E-戦略に変更したことにともないユーザーの並列指定が反映されるよう変化する。これは TRAM において、戦略リストがユーザーの指定戦略と等価なものであったことを考えれば当然といえ、さらに、この戦略リストの情報をもとに簡約順序が制御されていたこ

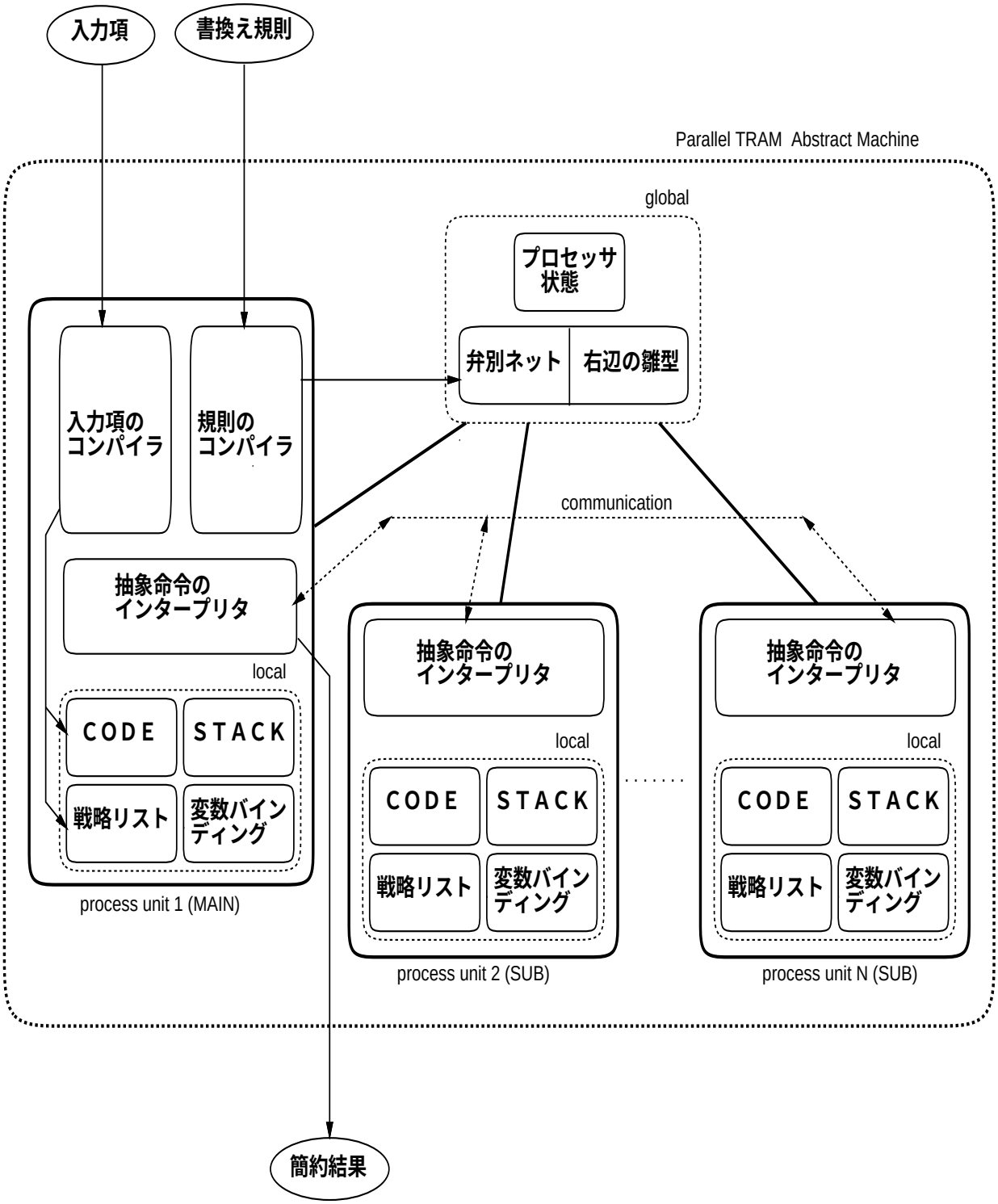

図 4.3: Parallel TRAM の構成

とを考えると、戦略リストこそ並列簡約を実現する重要なポイントであることは容易に想像できる。そこで Parallel TRAM でも、この戦略リストを用いて並列簡約を実現していくことにする。

4.3.1 戦略リストの生成

図4.4に示すのは、3章で定義したフィボナッチ数列の *plus* に `{ strat: ({1 2} 0) }` という戦略が指定され、入力項として `plus(fib(s(0)),fib(0))` が与えられた場合に生成される戦略リスト（ただし、この入力項のマッチングプログラムは、L1〜L6 に割り付けられているとする）である。また、一般的な項

`f(A1,A2, ⋯ ,An)`、 `f:{ strat: ( ⋯ {i ⋯ j k} ⋯ ) }`

が入力項として与えられた場合に生成される戦略リストも同図に載せておく。

この図からも分かるように、Parallel TRAM の戦略リストは次の点で TRAM の戦略リストと異なっている。

1. 引数項簡約は、その区切りが分かるようにブロック化されている。
2. 並列指定された引数項簡約は、その最後の引数項簡約を除いて全て先頭に "FORK" が付加される。
3. 並列指定の終りには "JOIN" が付加される。

FORK、JOIN というのは、並列簡約を実現するため新たに付け加えられた抽象命令で、並列簡約を指定した場合にはこれらの命令に制御を移せるよう、戦略リストの適切な位置にこれら命令へのラベルが挿入される。上記 2. 3. はその挿入の法則を表しており、機械的な作業で並列指定から戦略リストを生成することができる。また、戦略リストの雛型もこれと同様の方法で生成されることになる。
（3章で説明したように戦略リストは3つ組のリストである。残りの2つのデータにどのような情報を持たせるかについては、4.4.2 節で説明する。また、Parallel TRAM でも書換えの必要が無い戦略リストは生成しないという最適化を行なうつもりである。これに関しては、4.4.3 節で触れる。）

4.3.2 戦略リストに基づく並列簡約

インタープリタは、戦略リスト中に現れる新命令を実行することで並列簡約を行なっていく。この新たに追加された抽象命令の詳しい定義は次節で行なうものとして、ここでは簡単にそれぞれの命令の動作を説明しておく。

FORK: idle 状態のプロセスユニットがある場合には、そのユニットに戦略リストの次に続く１ブロックの簡約を EXIT 命令とともに割り付ける。idle 状態のユニットが無い場合、この命令は無視される。

JOIN: FORK した簡約が全て終了するまで待機する。また待機中は自分自身のユニット状態を idle にし、他の簡約を受け付けられるようにする。

EXIT: FORK された簡約が終了したことを親ユニットに伝える。(この命令は、FORK 命令が行なわれたときに割り付けられる。)

つまり並列指定された引数項簡約は、FORK 命令によって他ユニット上で簡約が行なわれ、JOIN/EXIT 命令で結果が統合される。この様子を図に表すと、図 4.5のようになる。

このように、最初はメインのプロセスユニット上で簡約が始められるわけだが、簡約が進み FORK 命令が実行されるにつれ、次第に複数のプロセスユニット上で並列簡約が行なわれるようになる。

また、並列指定された最後の引数項簡約をあえて FORK しないのは、この簡約の後には必ず JOIN 命令があり、この簡約を FORK してしまうと、次の JOIN 命令で待機状態に入る可能性が非常に高くなるからである。つまり、この簡約を自分自身で行なえば、無駄な FORK を減らすことが可能となる。(FORK を行なうにはそれなりのオーバーヘッドがかかるため、無駄な FORK をさせるのは効率が悪い。)

4.4 Parallel TRAM の定義

本節では、今まで説明してきた Parallel TRAM の構成・動作を整理して抽象命令を定義し、Parallel TRAM の設計を完成させる。なお、TRAM で用いられる抽象命令は、意味定義を一切変えることなく Parallel TRAM でも用いられており、並列簡約の実現は、TRAM の抽象命令に新しい命令を付け加えることで行なわれる。従って本節で定義する

のは、新たに追加された抽象命令だけにとどめ、TRAM、Parallel TRAM どちらの抽象機械にも共通に使用される抽象命令の定義は、付録 A に添付することにする。

4.4.1 大域レジスタと記憶領域

個々の抽象命令の定義を行なう前に、抽象機械の中で大域的に用いられる各種レジスタと、抽象機械が使用する記憶領域をまとめておく。その結果を表 4.1と図 4.6に示す。表 4.1中の * 印が付けられたレジスタおよび記憶領域は、プロセスユニットの数だけ複製されるものを表している。

抽象命令を定義するにあたり、これらのレジスタ・記憶領域を用いるわけだが、表記に関する約束として、レジスタ名_i、記憶領域名_i と記述した場合、それはプロセスユニット i 上の該当するレジスタと記憶領域を表しているものとする。また、自ユニットにおけるレジスタ・記憶領域を表す場合、この添字は省略する。ただし、自ユニットの識別番号を単独で用いる場合には、self と記述することにする。

4.4.2 戦略リストの構造

4.3.1 節で、Parallel TRAM における戦略リストの生成法について説明したが、FORK / JOIN / EXIT などの命令を問題無く行なうためには、これら命令へのラベル以外にもいくつかのデータが必要となる。今までにも何度か説明したように、戦略リストはその１つの要素に３つのデータを抱かせることができるため、FORK/JOIN/EXIT などで必要な情報は、この戦略リスト３つ組の空き領域に格納することにする。それぞれが必要とする情報は、次のようになる。

```
FORK : < LabelFORK、BlockEND、JoinAddress >
JOIN : < LabelJOIN、NumOfFORK、-- >
EXIT : < LabelEXIT、JoinAddress、ParentUnit >
```

LabelFORK : FORK 命令へのラベル。

BlockEND : FORK 命令で idle 状態のプロセスユニットに簡約を割り付ける際、LabelFORK の次の戦略リストから、この BlockEND までの戦略リストに該当する簡約を割り付ける。つまり、BlockEND は、引数項簡約の区切りを表しており、その値は、戦略リスト中のある要素のアドレスを指している。

JoinAddress： FORK した簡約が合流すべき JOIN のアドレス。

LabelJOIN： JOIN 命令へのラベル。

NumOfFORK： FORK 命令でつくり出された子プロセスの数。JOIN は、この値が 0 になるまで同期をとって待機することになる。

－：（未使用）

LabelEXIT： EXIT 命令へのラベル。

JoinAddress： 割り付けられた簡約が終了したときに合流すべき JOIN のアドレス。

ParentUnit： 簡約の割り付けを行なった親ユニット。

また、図 4.5におけるメインプロセスユニットとサブプロセスユニット 1 の戦略リストを正確に表すと次のようになる。

```
MAIN プロセスユニット                    SUB1 プロセスユニット
  100: <BINGO, -- , -- >                   2000: <EXIT, 106, MAIN >
  103: <L1, L1 を呼び出す親アドレス, 0 >     2003: <L2, L2 を呼び出す親アドレス, 0 >
  106: <JOIN, 1, -- >                      2006: <L4, L4 を呼び出す親アドレス, 0 >
  109: <L3, L3 を呼び出す親アドレス, 0 >     2009: <L6, L6 を呼び出す親アドレス, 0 >
  112: <L5, L5 を呼び出す親アドレス, 0 >
  ----------------------------------
  115: <L2, L2 を呼び出す親アドレス, 0 >
  118: <L4, L4 を呼び出す親アドレス, 0 >  ──────────┘ 割り付け
  121: <L6, L6 を呼び出す親アドレス, 0 >
  124: <FORK, 115, 106 >
```

上記において、戦略リストが割り付けられているアドレスに特別な意味はない。また、戦略リストの SL 領域への格納の仕方は、3 つ組の 1 番目、3 つ組の 2 番目、3 つ組の 3 番目、… の順で１次元的に格納される。従って、SL 領域のポインタ ST の値は、３づつの増減でインクリメント、デクリメントされ、３つ組の２番目、３番目の値を参照する場合には、SL[ST+1] 、SL[ST+2] のようにして参照する。

4.4.3 抽象命令の定義

並列処理を行なうという性格上、Parallel TRAM にはいくつかのクリティカルセクションが存在し、このクリティカルセクションを排他的に制御するためには、ロック機構が当然のごとく必要となる。そこで、抽象命令の定義の中で、

```
lock( var );   unlock( var );
```

という記述が出てきた場合、それは var という変数に対して、ロックをかけた（ロックを解除した）ということを表すものとする。

また、同様に

```
suspend( unit_i );    resume( unit_i );
```

という記述が出てきた場合は、unit_i というユニットを待機（再開）させることを表すものとする。

以上のいずれの動作も、実装する計算機、あるいは実装方法に大きく依存する動作であるため、抽象機械のレベルでは、これらをプリミティブとして扱い、これをさらに定義するといったことは行なわないことにする。また、これらの動作以外にも抽象度を害すると思われる補助的な動作については、自然言語のみの説明にとどめる場合がある。

> （※）抽象機械の最大の長所は、適度に抽象度を上げた命令群で全体の動作を定義するため、機械の挙動を容易に理解できる、という点にある。従って、全てを詳細に定義し、機械の動作を分かりずらくするよりも、多少厳密さに欠けたとしても動作の理解のし易さを優先させるべきであると考え、このような方針をとることにした。

それでは、並列簡約を実現するのに必要な抽象命令を１つづつ定義していくことにする。定義には、C 言語ライクなシンタックスを用いている。

FORK

この命令は、他ユニット上に引数項の簡約を割り付け、実際に並列簡約を行なわせるための命令である。その定義を図 4.7に示す。

この命令に制御が移ると、まず getIdleProsessor() を呼び出して、idle 状態のプロセスユニットを捜し出す。この返り値が NULL つまり、idle 状態のユニットが１つもなかった場合には、この FORK は子プロセスとして生成されなくなるので、合流すべき JOIN の NumOfFORK の値をデクリメントする。（JOIN が保有するこの NumOfFORK

は、現在の待ちプロセスの数を表しているため、デクリメントしないと永遠に JOIN で待機してしまうことになる。）このとき、NumOfFORK は排他的に制御しなければならないので、lock、unlock を前後で行なう。idle 状態のプロセスユニットが見つかった場合は、そのユニットの戦略リストに EXIT（付随するデータを含む）を格納し、続いて codeAndStrategyCopy() で BlockEND までの戦略リストと、この戦略リストに該当するマッチングプログラムを割り付ける。簡約に必要な環境が整ったら、最後にこのプロセスユニットを再開させる。

getIdleProcessor() の定義は図 4.8、codeAndStrategyCopy() の定義は図 4.9のようになる。

codeAndStrategyCopy() は少々複雑なので、ここで簡単に説明しておくことにする。この手続きで行なうことは戦略リストのコピーと、コピーした戦略リストに対応するマッチングプログラムのコピーなのであるが、このコピーを行なう際に考慮しなければならない重要なポイントが２点ある。まず１点目は、コピー先のマッチングプログラムのアドレスに対応して、戦略リストの内容も変化させなければならないということである。つまり、単純にそれぞれの領域をコピーするだけでは、戦略リストとマッチングプログラムの対応関係が狂ってしまい、うまく動作しないということである。そこで、マッチングプログラムをコピーする際（codeCopy()）には、SV 領域にコピー前、コピー後のアドレスを格納していき、戦略リストをコピーするとき（codeAndStrategyCopy() の 8 行目以降）には、この SV 領域の情報を見ながら正しい値を格納するものとする。SV 領域は、マッチングプログラムを実行しているときのみ必要な領域であるため、FORK 命令の実行時にこのような臨時の情報検索テーブルとして用いても何ら問題ない。また、戦略リストの要素の中には、SL 領域上のあるアドレスを保持するものがいくつかある。これらの要素に対しては、コピー前に戦略リストが割り付けられていたアドレスと、コピー後に割り付けられるアドレスからオフセット値（定義中の offset）を計算し、このオフセット値を用いてデータの補正を行なっている。

続いて重要なポイントの２点目であるが、これはマッチングプログラムのコピーに絡んだ問題である。戦略リストの１ブロックの書換えに対応するマッチングプログラムをコピーするといった場合、一番単純な方法として考えられるのは、その戦略リスト１ブロックに格納されている全てのアドレスからたどれる全てのマッチングプログラムを、全部コピーしてしまうというやり方である。しかしながらこの方法は非常に効率が悪い。な

ぜなら、戦略リスト１ブロック分の書換えというのは、ある演算子 f のある引数項 t_n の簡約に対応しているため、その１ブロックに格納されているアドレスは、全て引数項 t_n の部分項簡約に対応したアドレスとなるからである。つまり、１ブロックに格納されている全てのアドレスをルートとして、マッチングプログラムをコピーしてしまうと、見事に重複した無駄なコピーを次々と行なうことになってしまう。逆にいえば、引数項 t_n の全体項に対応するアドレスさえ分かれば、無駄なコピーを行なわず効率良く１ブロック分のコピーを行なうことが可能となる。ここで、並列 E-戦略の定義を思いだして頂きたいのだが、並列 E-戦略ではその戦略指定の最後に必ず全体項簡約を指定することを義務づけている。これは結果として非常に良い性質をもたらしてくれる。その性質とは、戦略リスト１ブロックの最後の要素には、必ずその１ブロック中の簡約のトップのアドレスが格納されるという性質である。従って、今回のコピーでは戦略リスト FORK に格納される BlockEND の位置の戦略リストのアドレスをルートアドレスとしてコピーすれば良い。図 4.9の 6 行目で、`codeCopy()` の引数に `SL[start + 1]` を渡しているのは、このためである。

マッチングプログラムのコピーを行なう `codeCopy()` は、図 4.10に示すような再帰的な手続きで定義される。

JOIN

この命令は、FORK した簡約が全て終了するまで待機する同期命令である。その定義は、図 4.13のようになる。

この命令に制御が移ると、まず NumOfFORK にロックを掛け、この値が 0 であるかどうか調べる。この値が 0 だった場合、それは FORK された簡約が全て終了していることを表しているので、NumOfFORK のロックを解除し、この JOIN の処理から抜ける。値が 0 でなかった場合は、FORK された簡約の中でまだ未終了のものが残っているということになるので、自ユニットをサスペンドし、排他的に自ユニットの状態を idle にセットして、最後に NumOfFORK のロックを解除する。

> （※）自ユニットをサスペンドした後で、状態を idle にセットし、ロックを解除するという順番は、間違いではなく正しい順番である。それどころか、この順番は非常に重要で、もしこの実行順序が破られたとすると動作は保証されない。（再開させたつもりが待機状態のまま、といった不具合が生じる。）自ユニットを待機させた後で、どうやって残りの処理を行なうかは実装時に考えるべき問題で、抽象機械の動作定義の本質ではない。（実際に、実現の方法はいくつか考えられる。）従って、抽象命令の

定義においては、こういった一見矛盾（実は全然矛盾ではない）を含むような記述を許すものとする。

EXIT

この命令は、FORK によって割り付けられた簡約が終了したことを、親ユニットに伝えるための命令である。その定義は、図 4.14のようになる。

この命令に制御が移されるのは、FORK 命令によって割り付けられた簡約が終了したときである。従ってまず、合流すべき JOIN の NumOfFORK にロックを掛け、この値をデクリメントする。これは、処理の終了を親ユニットに伝えたことと等しい。そして、デクリメント後の値が 0 だった場合は、親ユニットの同期を解除しなければならないので、親ユニットの状態を確認し、idle だった場合には状態を busy にセットし直して親ユニットを再開させる。親ユニットが既に busy だった場合は、少なくとも今着目している JOIN で待機状態に移行することは有り得ないので、特に何も行なわない。

SLEEP

今まで定義した３命令の他に、この SLEEP 命令なる命令をさらに追加する。この命令は、サブプロセスユニット上で簡約すべき処理が全て無くなった時（のみ）に実行される命令で、この命令に制御が移ると、この命令を実行したユニットは自らの状態を idle にして待機する。また、この命令に制御を移す方法も、今まで説明した３命令と同じく、戦略リスト中にこの命令へのラベルを置くことで行なわれる。従って、サブプロセスユニットの戦略リストの最後には、デフォルトでこの SLEEP 命令へのラベルが挿入されることになり、それ以外の箇所でこの命令が呼び出されることはない。また、SLEEP 命令のラベルを戦略リストに格納するとき、３つ組の 2 番目と 3 番目の要素には特に何も必要としない（未使用）。

この命令の定義は、図 4.15のようになる。

NOP

この命令も SLEEP 命令と同じくここで初めて登場した命令で、動作としては何も行なわない空命令である。３章、戦略リストの節の注意 2 でも説明したように、実際の戦略リストの要素のなかには、最適化の対象となって取り除かれるものがいくつか出てくる。し

かし、この最適化が適用できるのは、書換えが行なわれ戦略リストを再構成するときに限られる。従って、右辺の雛型段階ではまとまった構造をしていた戦略リストも、書換えを行ない戦略リストを再構成した段階で、空（empty）の戦略リストを FORK したり、その FORK を JOIN で待つなどといった、おかしな戦略リストに仕上がる場合がある。そういう場合、FORK、JOIN の代わりにこの NOP 命令を埋め込んで、戦略リストの補正を行なう。SLEEP 命令同様、３つ組の 2 番目、3 番目は未使用である。

この NOP 命令は何もしない命令なので、定義については省略する。

以上が並列簡約を実現するための抽象命令である。これらの命令を TRAM に組み込めば、並列 E-戦略により指定された引数項の並列簡約を実現できるようになる。

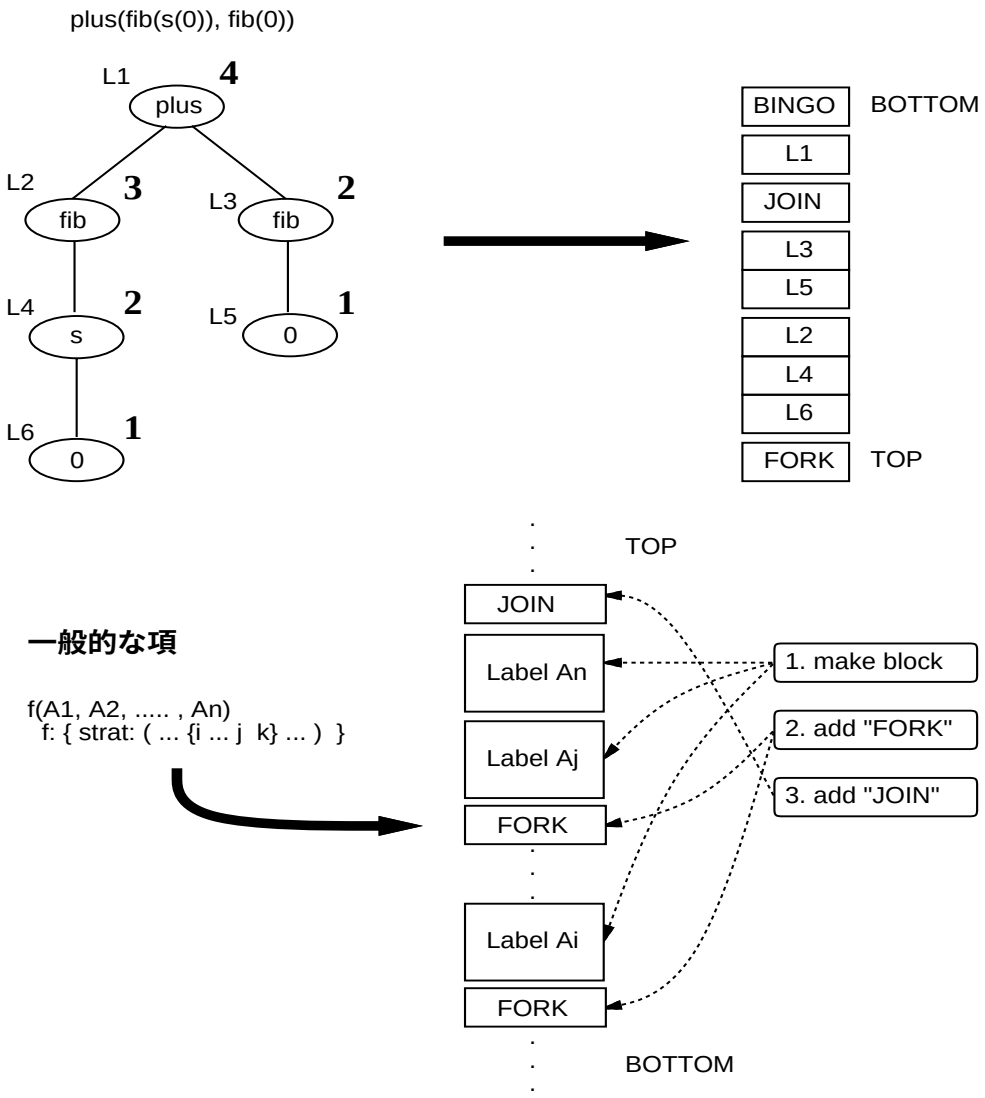

図 4.4: 並列指定で生成される戦略リスト

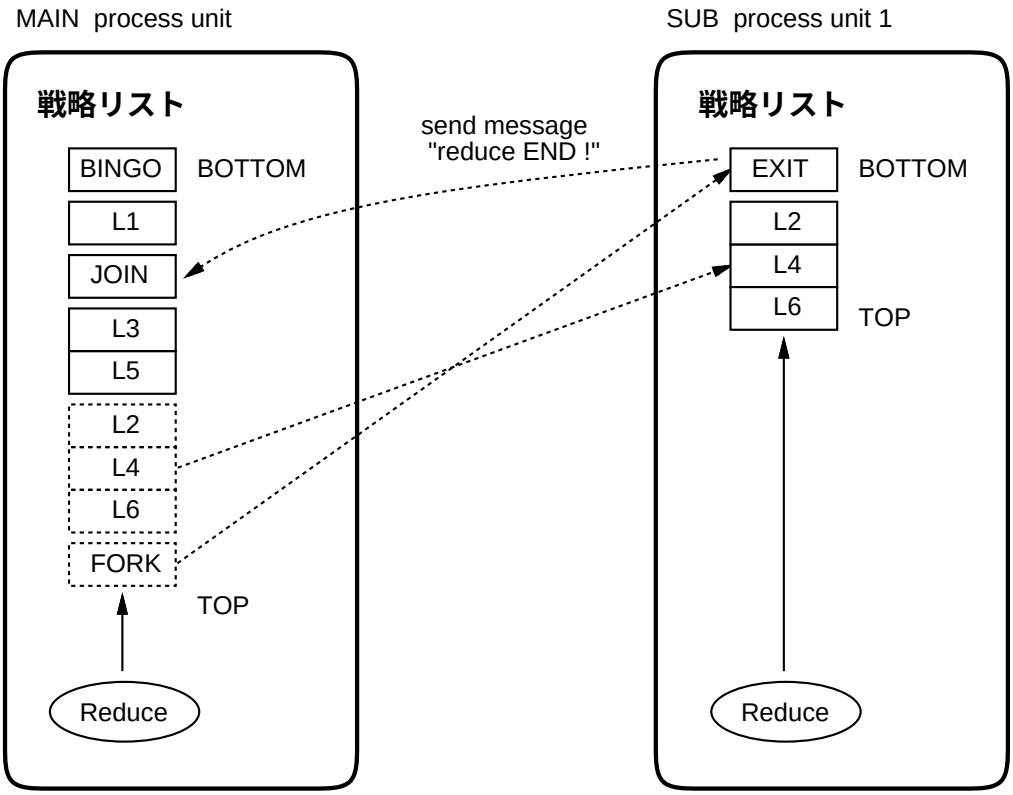
MAIN process unit
SUB process unit 1
戦略リスト
BINGO
BOTTOM
L1
JOIN
L3
L5
L2
L4
L6
FORK
TOP
Reduce
send message
"reduce END !"
戦略リスト
EXIT
BOTTOM
L2
L4
L6
TOP
Reduce

図 4.5: 並列簡約の様子

領域名	表す内容
DNET	弁別ネット領域
CR	右辺の雛型領域
CODE*	マッチングプログラム領域
SL*	戦略リスト領域
STACK*	スタック領域
SV*	変数バインディング領域
STATE	プロセスユニットの状態を格納する領域（排他制御の必要有り）

レジスタ名	表す内容
DB	DNET（弁別ネット領域）の開始アドレス
DT	DNET 上のポインタ（DNET の空き領域の開始アドレス）
CRB	CR（右辺の雛型領域）の開始アドレス
CRT	CR 上のポインタ（CR の空き領域の開始アドレス）
CB	CODE（マッチングプログラム領域）の開始アドレス
CT*	CODE 上のポインタ（CODE の空き領域の開始アドレス）
SB*	SL（戦略リスト領域）の開始アドレス
ST*	SL 上のポインタ（SL の空き領域の開始アドレス）
BOS*	STACK（スタック領域）の開始
EOS*	STACK（スタック領域）のエンド
SP*	スタックポインタ
SVB*	SV（変数バインディング領域）の開始アドレス
SVT*	SV 上のポインタ（SV の空き領域の開始アドレス）
CN*	現在たどっている弁別ネット上のノードアドレス
P*	プログラムカウンタ
RP*	サブルーチンコールの戻り番地
FD*	適用可能な規則の個数

表 4.1: Parallel TRAM の大域レジスタ

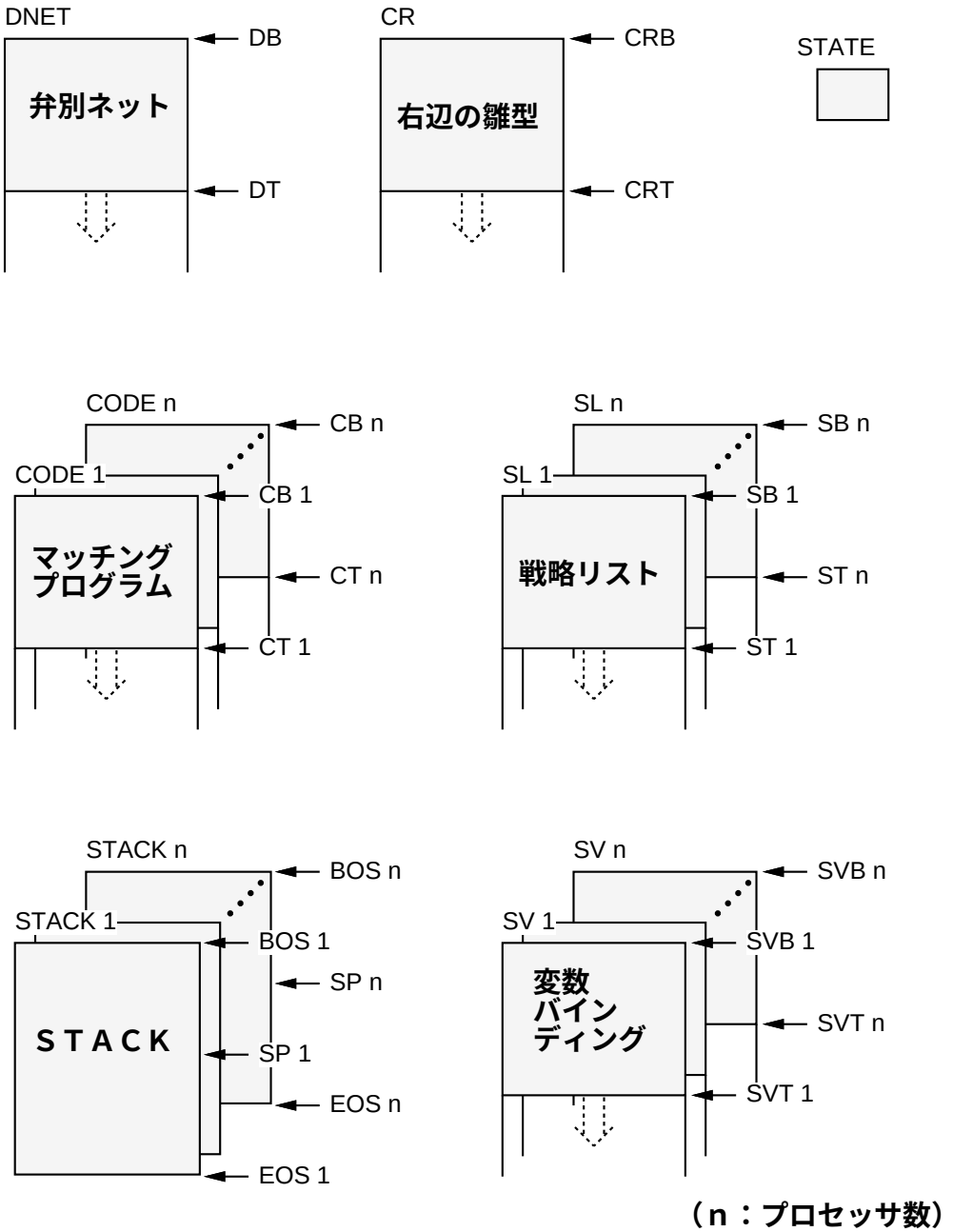

図 4.6: Parallel TRAM の記憶領域

```
 1: FORK( )
 2: {
 3:   JoinAddress ← SL[ST + 1];
 4:   idle ← getIdleProcessor( );
 5:   if ( idle = NULL ) {
 6:     lock( SL[JoinAddress + 1] );
 7:     SL[JoinAddress + 1] ← SL[JoinAddress + 1] - 1;
 8:     unlock( SL[JoinAddress + 1] );
 9:   } else {
10:     SL_idle[ST_idle] ← LabelEXIT;
11:     SL_idle[ST_idle + 1] ← JoinAddress;
12:     SL_idle[ST_idle + 2] ← self;
13:     ST_idle ← ST_idle + 3;
14:     codeAndStrategyCopy( SL[ST + 2], ST - 3, idle );
15:     resume( idle );
16:   }
17: }
```

図 4.7: FORK の定義

```
 1: getIdleProcessor( )
 2: {
 3:   i ← 1;
 4:   while ( i ≦ N ) {
 5:     lock( STATE[i] );
 6:     if ( STATE[i] = idle ) {
 7:       STATE[i] ← busy;
 8:       unlock( STATE[i] );
 9:       return i;
10:     }
11:     unlock( STATE[i] );
12:     i ← i + 1;
13:   }
14:   return NULL;
15: }
```

図 4.8: getIdleProcessor の定義

```
 1: codeAndStrategyCopy( start:SL 上のコピー開始アドレス,
 2:                      end:SL 上のコピー終了アドレス,
 3:                       id:コピー先のユニット番号     )
 4: {
 5:   SVT ← SVB;   /* 情報検索用テーブルのポインタの初期化 */
 6:   ADDR ← codeCopy( SL[start], id );     /* CODE 領域のコピー */
 7:   CODE[ SL[start + 1] ] ← "call ADDR";
                      /* コピー前 → コピー後ポインタ張り換え */
 8:   offset ← ST_id - start;
 9:   while ( start ≦ end ) {
10:     switch( SL[start] ) {
11:       case FORK:
12:         SL_id[ST_id] ← FORK;
13:         SL_id[ST_id + 1] ← SL[start + 1] + offset;
14:         SL_id[ST_id + 2] ← SL[start + 2];
15:       case JOIN:
16:         SL_id[ST_id] ← JOIN;
17:         SL_id[ST_id + 1] ← SL[start + 1];
18:       default:
19:         SL_id[ST_id] ← reference( SL[start] );
20:         SL_id[ST_id + 1] ← reference( SL[start + 1] );
21:         if ( SL[start + 2] = 0 ) {
22:           SL[ST_id + 2] ← 0;
23:         } else {
24:           SL[ST_id + 2] ← SL[start + 2] + offset;
25:         }
26:     }
27:     start ← start + 3;
28:   }
29: }
```

図 4.9: codeAndStrategyCopy の定義

```
 1: codeCopy( ptr:CODE 上のアドレス, id:コピー先のユニット番号 )
 2: {
 3:   CODE_id[CT_id] ← CODE[ptr];
 4:   entry( ptr, CT_id );
 5:   ar ← arity( CODE[ptr] );
 6:   if ( ar = 0 ) {
 7:     CT_id ← CT_id + 1;
 8:     return CT_id - 1;
 9:   } else {
10:     ct ← CT_id;
11:     CT_id ← CT_id + ar + 1;
12:     for ( i ← 1; i ≦ ar; i ← i + 1 ) {
13:       addr ← address( CODE[ptr + i] );
14:       X ← copyCode( addr, id );
15:       CODE[ct + i] ← "call X";
16:     }
17:     return ct;
18:   }
19: }
```

図 4.10: codeCopy の定義

情報検索用テーブルへの情報登録を行なう手続き

```
1: entry( from:コピー元のアドレス, to:コピー先のアドレス )
2: {
3:   SV[SVT] ← from;
4:   SV[SVT + 1] ← to;
5:   SVT ← SVT + 2;
6: }
```

情報検索テーブルの参照を行なう手続き

```
1: reference( from:コピー元のアドレス )
2: {
3:   for ( svt ← SB; svt < SVT; svt ← svt + 2 ) {
4:     if ( SV[svt] = from ) {
5:       return SV[svt + 1];
6:     }
7:   }
8:   return from; /* 登録されていない場合は、from をそのまま返す */
9: }
```

図 4.11: entry、reference の定義

アリティ数を取得する手続き

```
 1: arity( prog:マッチングプログラムの match_symbol 命令 )
 2: {
 3:   return prog のオペランドシンボルのアリティ数;
 4: }
```

call 命令のオペランドアドレスを取得する手続き

```
 1: address( prog:マッチングプログラムの call 命令 )
 2: {
 3:   return prog のオペランドのアドレス;
 4: }
```

図 4.12: arity、address の定義

```
 1: JOIN( )
 2: {
 3:   NumOfFORK ← SL[ST + 1];
 4:   lock( NumOfFORK );
 5:   if ( NumOfFORK = 0 ) {
 6:     unlock( NumOfFORK );
 7:   } else {
 8:     suspend( self );
 9:     lock( STATE[self] );
10:     STATE[self] ← idle;
11:     unlock( STATE[self] );
12:     unlock( NumOfFORK );
13:   }
14: }
```

図 4.13: JOIN の定義

```
 1: EXIT( )
 2: {
 3:   JoinAddress ← SL[ST + 1];
 4:   parent ← SL[ST + 2];
 5:   lock( SL_parent[JoinAddress + 1] );  /* NumOfFork のロック */
 6:   SL_parent[JoinAddress + 1] ← SL_parent[JoinAddress + 1] - 1;
 7:   if ( SL_parent[JoinAddress + 1] = 0 ) {
 8:     lock( STATE[parent] );
 9:     if ( STATE[parent] = idle ) {
10:       STATE[parent] ← busy;
11:       resume( parent );
12:     }
13:     unlock( STATE[parent] );
14:   }
15:   unlock( SL_parent[JoinAddress + 1] );
16: }
```

図 4.14: EXIT の定義

```
1: SLEEP( )
2: {
3:   suspend( self );
4:   lock( STATE[self] );
5:   STATE[self] ← idle;
6:   unlock( STATE[self] );
7: }
```

図 4.15: SLEEP の定義

第 5 章

Parallel TRAM の実装

引数項簡約を並列に行なわせることによって、どの程度効率が改善されたのかを見極めるには、４章で定義した Parallel TRAM を適当な並列計算機上に実装する必要がある。そこで本章では、Parallel TRAM をどのようにして実際の並列計算機にマッピングし、実装を行なったか、その主なポイントについて述べていく。

今回実装の対象とした並列計算機は、MC88100 を４台搭載した共有メモリ型並列計算機であるオムロン社製 LUNA-88k2 で、実装言語としてはC言語を使用した。

5.1 プロセスユニットの管理

5.1.1 プロセスユニットの実装

Parallel TRAM を並列計算機上に実装するにあたり、一番問題となってくるのは、抽象機械におけるプロセスユニットの概念をどうやって実現するかである。今回の実装の目的は、引数項の並列簡約を Parallel TRAM の簡約メカニズムで実行させたとき、どれくらいの効率改善が期待できるのかを見極めるのが目的で、計算機のアーキテクチャに多分に依存した特別な処理系を作り上げることが目的ではない。従って今回の実装では、計算機上の処理の基本単位となるスレッドを、実際のプロセッサの数だけ生成し、そのスレッドをプロセスユニットと等価と見なして実装を行なうことにする。生成するスレッドの優先度を全て同じ優先度に設定し、スレッド数もプロセッサと同じ数しか存在しなければ、スレッド間で CPU パワーを奪い合うといった競合もなくなり、抽象機械のプロセスユニッ

トを、かなり理想的な形でシミュレートすることができると考える。そこで、図 5.1に示すような方法で、プロセスユニットの概念を実装することにする。

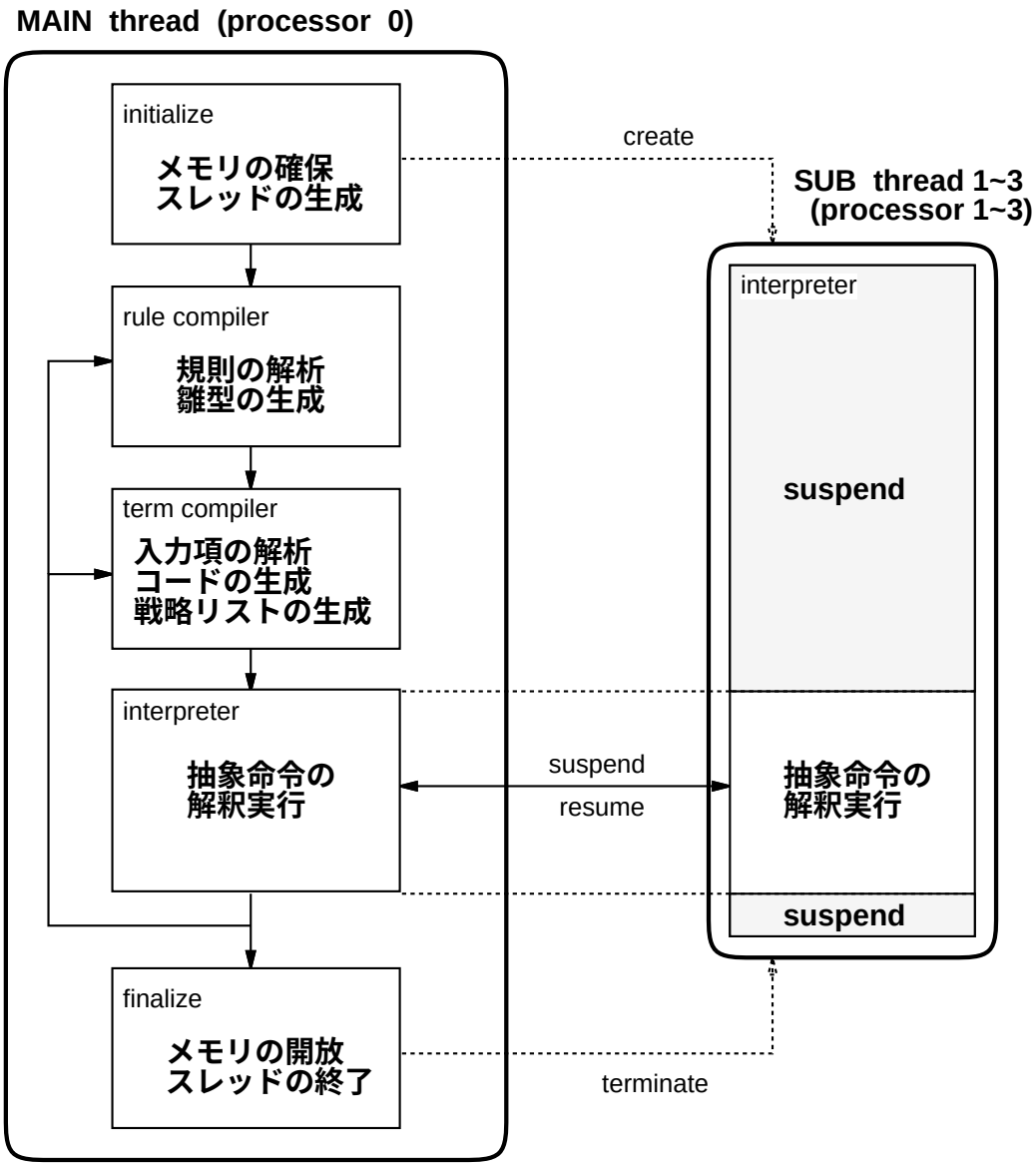

図 5.1: プロセスユニットの実装

メインプロセスユニットの役割を果たすメインスレッドは、Parallel TRAM 起動時に、抽象命令の解釈実行のみを専用で行なうサブスレッドを３つ生成する。生成されたサブスレッドは、デフォルトで戦略リストに挿入されている SLEEP 命令によりサスペンド状態

に入る。従って、メインスレッドが規則のコンパイル、入力項のコンパイルを行なっている間、サブスレッドはサスペンド状態で待機していることになる。そして、メインスレッド上でインタープリタが動作し始めると、FORK 命令により必要に応じてサブスレッドが起動し、簡約処理を進めていく。

5.1.2 プロセスユニットの待機、再開

Parallel TRAM のプロセスユニットは、必要なときだけ再開（resume）されて、必要のないときには待機（suspend）している。このプロセスユニットの待機、再開を実現する方法として、

- スレッドの suspend、resume による実現。
- busy-wait 方式による実現。

の２つの方法が考えられる。１つ目の方法は、プロセスユニットがスレッドと等価であることから必然的に生じてくる方法で、いわゆる基本的なスレッド操作を行なうことに等しい。２つ目の方法は、busy-wait つまり、スレッドを完全にサスペンド状態にするのではなく、待機状態にさせたい場合は条件ループを実行し、制御の流れをその箇所で足止めさせておくという方法である。ただしこの方法は、ある条件（この場合 resume が呼ばれる）が満たされるまでループをひたすら回り続けるため、他に実行させたい処理があっても、CPU パワーをこのループ処理に取られてしまい、実行効率が悪くなるというのが一般的である。

しかし、今回の Parallel TRAM が待機状態に入るタイミングは、その時点で簡約すべき項が無いときに限られているため、busy-wait で待機させても何ら問題ない。それどころか、スレッドを suspend、resume させるといった余分なオーバーヘッドをなくすことができ、かえって効率は良くなるものと考えられる。従って本実装では、busy-wait 方式を採用することにする。

5.1.3 アイドル・キュー

４章でも説明したように、Parallel TRAM には、idle 状態のプロセスユニットが存在するときだけ引数項簡約の FORK を行なうという非常に簡単なスケジューラが組み込

まれおり、プロセスユニットの状態をどのように管理するかは、このスケジューラー（getIdleProcessor）の構造と深く関わってくる。単純な配列のみでこのプロセスユニットの状態を管理しても良いのだが、この方法で待機状態のプロセスユニットを検索しようとすると、プロセスユニット数に線形な計算量を必要としてしまう。そこで今回の実装では、この待機ユニットの検索効率のことを考慮して、配列による状態テーブルだけでなく、待機状態のプロセスユニットをキュー（FIFO）構造で連結させた”アイドル・キュー” を用いてプロセスユニットの状態を管理することにする。アイドル・キューは図5.2のようになる。

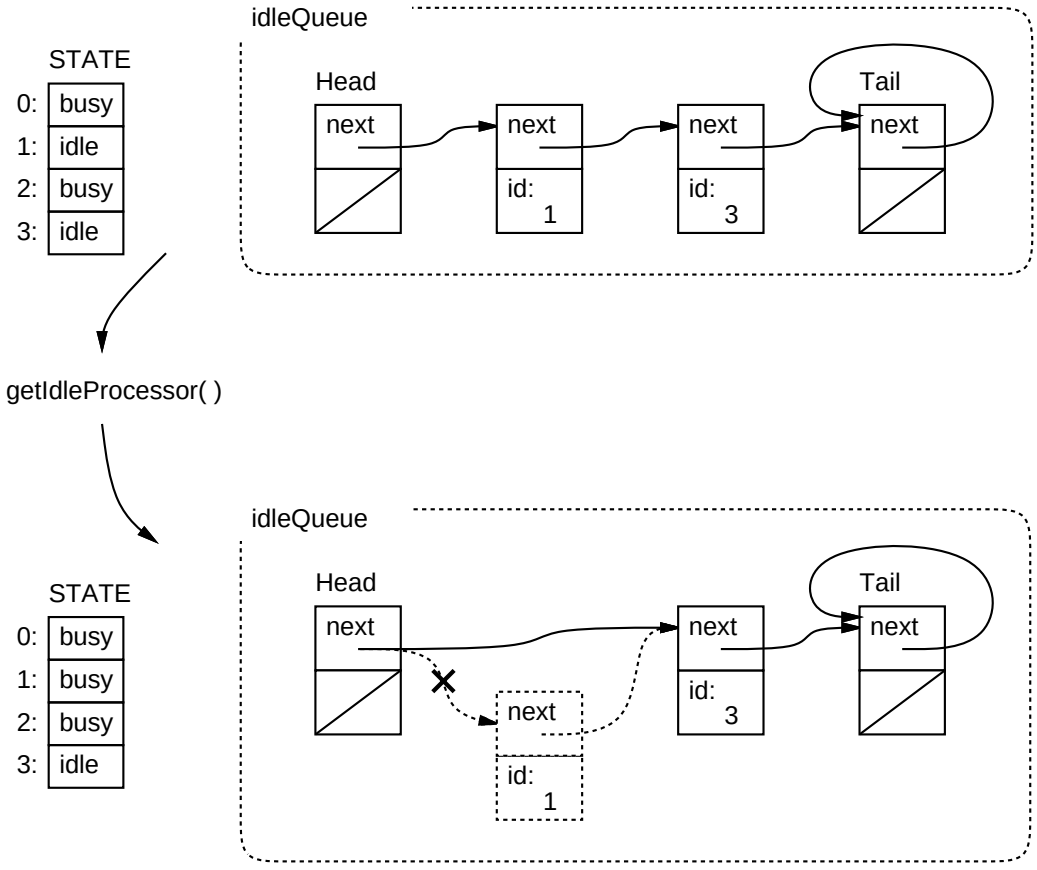

図 5.2: アイドル・キュー

この方法だと、単純なポインタ操作（１回のポインタ参照と１回のポインタ張りかえ）だけで待機ユニットの検索を行なうことができ、さらにプロセスユニットの数が増えても常に一定の計算量で検索が行なえるという利点を持つ。また、待機状態のプロセスユニッ

トが１つも無い場合における getIdleProcessor の返り値は Tail となり、Tail のネクストポインタが自分自身を指しているのは、待機ユニットの有無に関わらず全く同じポインタ操作で getIdleProcessor を実現するための工夫である。

5.2 メモリの構成

LUNA-88k2 は、共有メモリ型の並列計算機で、４プロセッサ共通のアドレススペースとして 16〜64MB（今回もちいた LUNA は 48MB）の記憶領域を持っている。抽象機械のレベルでは、複製される領域、されない領域、全てを独立した記憶領域として表現していたのだが、計算機に実装するには、それぞれの領域をどのように割り振るか具体的に決定してやる必要がある。そこで、今回の実装では、図 5.3のようなメモリの構成をとることにする。

複製されないレジスタ、弁別ネット、右辺の雛型など、全てのプロセスユニットで共通的に使用される領域を下位のスペースに割り付ける。その後には、プロセスユニットの数だけ複製されるレジスタ・領域（ただし、CODE 領域は除く）をユニットごとにまとめて、順番に割り付けていく。これは、プロセスユニットごとのローカリティーを高めるための配慮である。

> （※）LUNA-88k2 は、共有メモリ型の並列計算機なので、ローカリティーなどの話はあまり関係ないと思われるかもしれないが、実際のところ、頻繁にアクセスされるレジスタなどは、各プロセッサのキャッシュに４ワード単位で格納されることになる。例えば、それぞれのプロセスユニットで使用されるP（プログラムカウンタ）を連続した領域に割り付けたとすると、各プロセッサのキャッシュには、自分自身では全く使用しない他プロセスユニットの P まで格納してしまうことになり、そうすると、それぞれのプロセスユニットで P の値が書き換えられる度に、キャッシュのコヒーレンシー（整合性）を取るため、本来全く関係のない自プロセッサの処理を中断し、キャッシュの内容を書き換えるという非効率なことが起こってしまう。これを避けるために、プロセスユニットごとで用いるデータをまとめて格納するわけである。

そして最後に CODE 領域を割り付ける。CODE 領域もプロセスユニットごとに複製される領域なので、他の複製される領域と同じような形で割り付けても良いのだが、CODE 領域は性格的に最も激しく消費される領域で、簡約の割り付けかたによっては、領域を大量に消費するユニット、それほど消費しないユニットの差が顕著に現れてくる領域でもあ

る。従って、それぞれのプロセスユニットごとで固定したサイズの領域を割り付けたとすると、他のユニットにはまだ十分な領域が残っているにも関わらず、GC を行なわなければならないという状況が頻繁に発生すると考えら、今回は CODE 領域を効率良く消費するために、このようなまとまった領域を確保することにした。この CODE 領域をどのように使用するかは次節で説明する。

5.3 メモリ（CODE 領域）の管理

前節でも説明したように、CODE 領域だけはメモリの消費効率を考慮して、各ユニット共通のまとまった領域が確保されている。この CODE 領域を図に表すと、図 5.4のようになる。

また、CODE 領域は次のようにして管理される。

- CODE 領域全体を比較的小さなブロック（0.5MByte）に分割し、CODE 領域の未使用ブロックの先頭を表すポインタ `GlobalCT` を用意する。
- あるユニットにおいて CODE 領域が必要な場合は、`getMemory( )` を呼び出して、未使用のブロックを１つ獲得する。
- `getMemory( )` が呼び出された時、`GlobalCT` が`EndOfCODE`（CODE 領域の終端）に到達していた場合は、GC モードに移行する。

つまり CODE 領域を小さなブロックに分割したことにより、それぞれのプロセスユニットは、本当に必要な分だけのメモリをブロック単位で消費することができるようになったわけである。この方法だと、ユニット間で CODE 領域の消費量にバラツキがあっても、効率良くメモリを消費することができ、GC の回数も必要最低限の回数におさえることが可能となる。

> **注意：** Parallel TRAM の起動直後は、メインプロセスユニットにのみ１ブロックが割り当てられ、その他のユニットには特にブロックを割り当てたりしない。メインユニットに１ブロックを割り当てるのは、入力項をコンパイルする時に CODE 領域が必要となるからである。その他のユニットは、簡約が始まり必要に応じて`getMemory( )` を呼び出し、領域を確保する。

5.4 ガベージ コレクション

Parallel TRAM で GC の必要があるオブジェクトは、CODE 領域に蓄えられるマッチングプログラムだけである。なぜなら、書き換えにより変更されるマッチングプログラムは部分的なマッチングプログラムで、変更の仕方も、変更後のマッチングプログラムを空き領域に格納し、親アドレスの値をその新しい格納先に書き換えるだけなので、変更前の不必要なマッチングプログラムが CODE 領域に残されたままとなるからである。従って CODE 領域が一杯になったら必要なオブジェクトだけを一箇所に集め、CODE 領域を新たに使用できるよう GC してやる必要がある。

一方その他の領域では、その領域上のポインタ（ST、SP など）より下位アドレスのオブジェクトが必要なオブジェクトで、上位アドレスのオブジェクトが不必要なオブジェクトであることがはっきりしているため、GC を行なう必要はない。

では、どのような方式で CODE 領域の GC を行なうかであるが、今回の実装では各プロセスユニット間でグローバルな同期を取ったコピー方式の GC を採用することにした。グローバルな同期というのは、あるプロセスユニットが`getMemory( )` を呼び出した時に `GlobalCT` が`EndOfCODE` に到達していたら、全てのユニットの処理を中断させて GC を行なうというものである。このとき、CODE 領域へアクセスしている最中に処理を止められ、GC が行なわれるといった不都合が起こらないよう配慮する必要がある。そのため本実装では、GC の必要性の有無を表したフラグ `GCFLAG` をグローバルに設け、このフラグを使って GC 全般を制御している。GC 検知から GC 終了までの流れを簡単に表すと次のようになる。

1. あるユニットが GC の必要性を検知したら `GCFLAG` を立てる。
2. それぞれのユニットは、１回の書換えが終るごとに `GCFLAG` をチェックし、このフラグが立っていた場合は自分のユニット状態を ”GC” にして待機する。
3. 1. で GC の必要性を検知したユニットは、他ユニットが全て待機状態になったのを確認して GC を行なう。
4. GC が完了したら、`GCFLAG` を元に戻し、”GC” 状態で待機しているユニットを再開させる。

以上この `GCFLAG` を用いてグローバルな同期を実現している。

（※）もちろん、GC の必要性を検知したユニットだけが GC に移行し、その他のユニットは自分の処理を続けるという、独立した GC の方法も考えられるのだが、この方法だと CODE 領域の全てのセルを排他制御する必要があり、かえって効率が悪くなる可能性が極めて高い。

また、GC そのものはコピー方式の GC を CODE 領域用にアレンジした、次のような GC を採用している。

1. 戦略リストに格納されているマッチングプログラムのアドレスをルートとして、このルートからたどれる全てのマッチングプログラムを FUTURE 領域にコピーする。

2. 1. のコピーを、全てのプロセスユニット中の全ての戦略リスト要素（もちろん既に消費した戦略リスト要素は除く）に対して行なう。

3. コピーが終了したら、FUTURE 領域と PAST 領域を入れ換え、CODE 領域上のポインタ `GlobalCT` を初期化（`GlobalCT` ← `CB`）する。

つまり、上記 1.2. の作業で必要なマッチングプログラムが全て FUTURE 領域に集められ、そうすると CODE 領域のマッチングプログラムも、PAST 領域のマッチングプログラム（前回の GC で集められたマッチングプログラム）も必要なくなり、どちらの領域も開始アドレスから上書きできるようになるわけである。この様子を図に表すと図 5.5のようになる。

5.5 ロック機構

設計段階でも説明したように、Parallel TRAM には排他的にアクセスしなければならないクリティカルセクションがいくつか存在する。当初はこの排他制御を実現するためのロック機構として、単純な test-and-set 方式のスピンロックを採用していたが、処理の重い不可分な交換命令（fetchAndStore）を常に実行し続けるため、あまり良い効率は得られなかった。そこで、常に fetchAndStore を実行し続けるのではなく、キャッシュされている値を見て本当にアンロック状態のときだけ fetchAndStore を実行する test-and-test-and-set のスピンロックを、遅延が導入できるように改良して用いることにした。遅延を導入することで、スピンロックの性能が向上することは、Anderson[2] により報告されている。そのスピンロックは図 5.6のようになる。

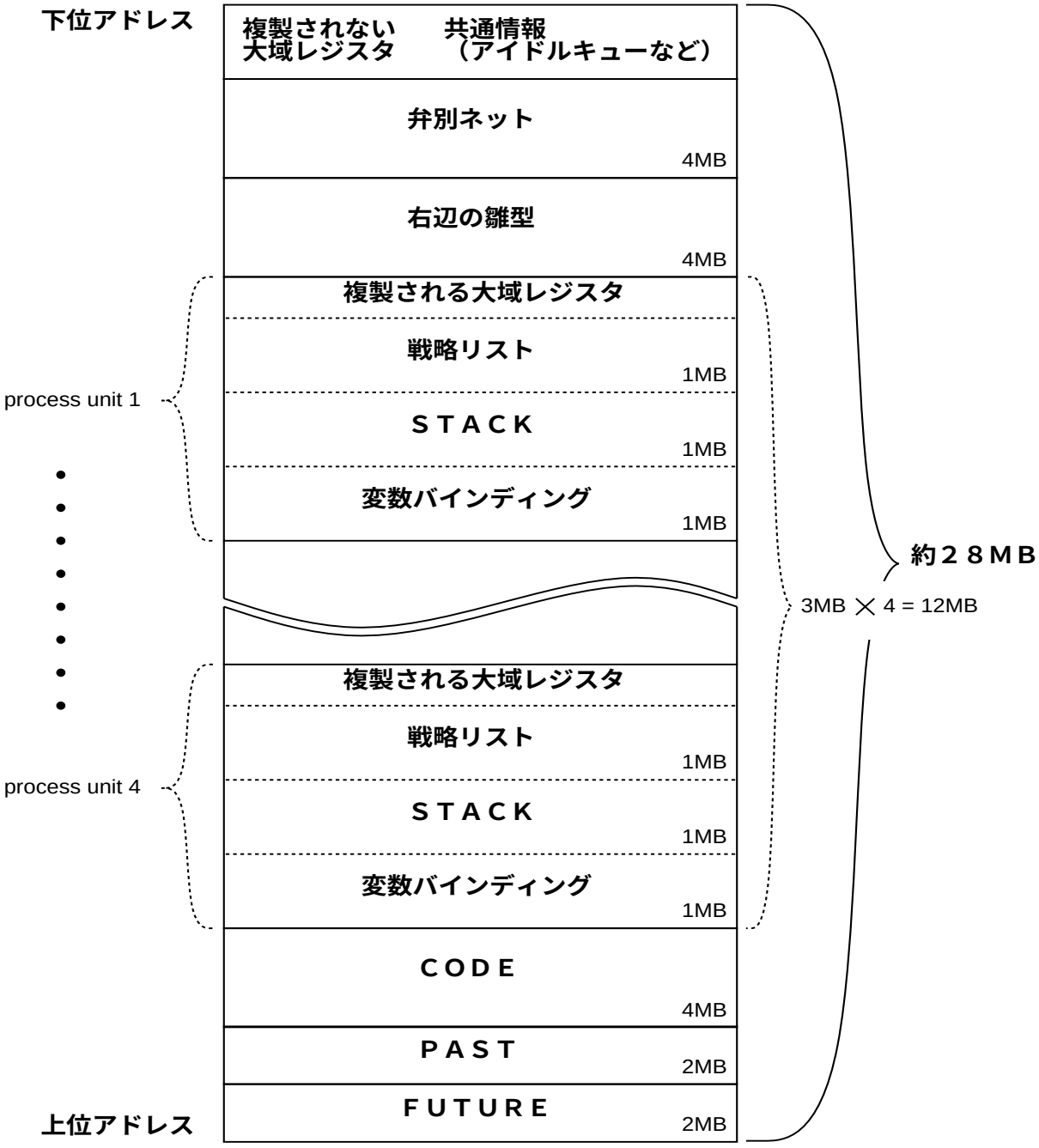
下位アドレス
複製されない　共通情報
大域レジスタ　（アイドルキューなど）
弁別ネット
4MB
右辺の雛型
4MB
複製される大域レジスタ
戦略リスト
1MB
process unit 1
ＳＴＡＣＫ
1MB
変数バインディング
1MB
約２８ＭＢ
3MB × 4 = 12MB
複製される大域レジスタ
戦略リスト
1MB
process unit 4
ＳＴＡＣＫ
1MB
変数バインディング
1MB
ＣＯＤＥ
4MB
ＰＡＳＴ
2MB
ＦＵＴＵＲＥ
2MB
上位アドレス

図 5.3: メモリの構成

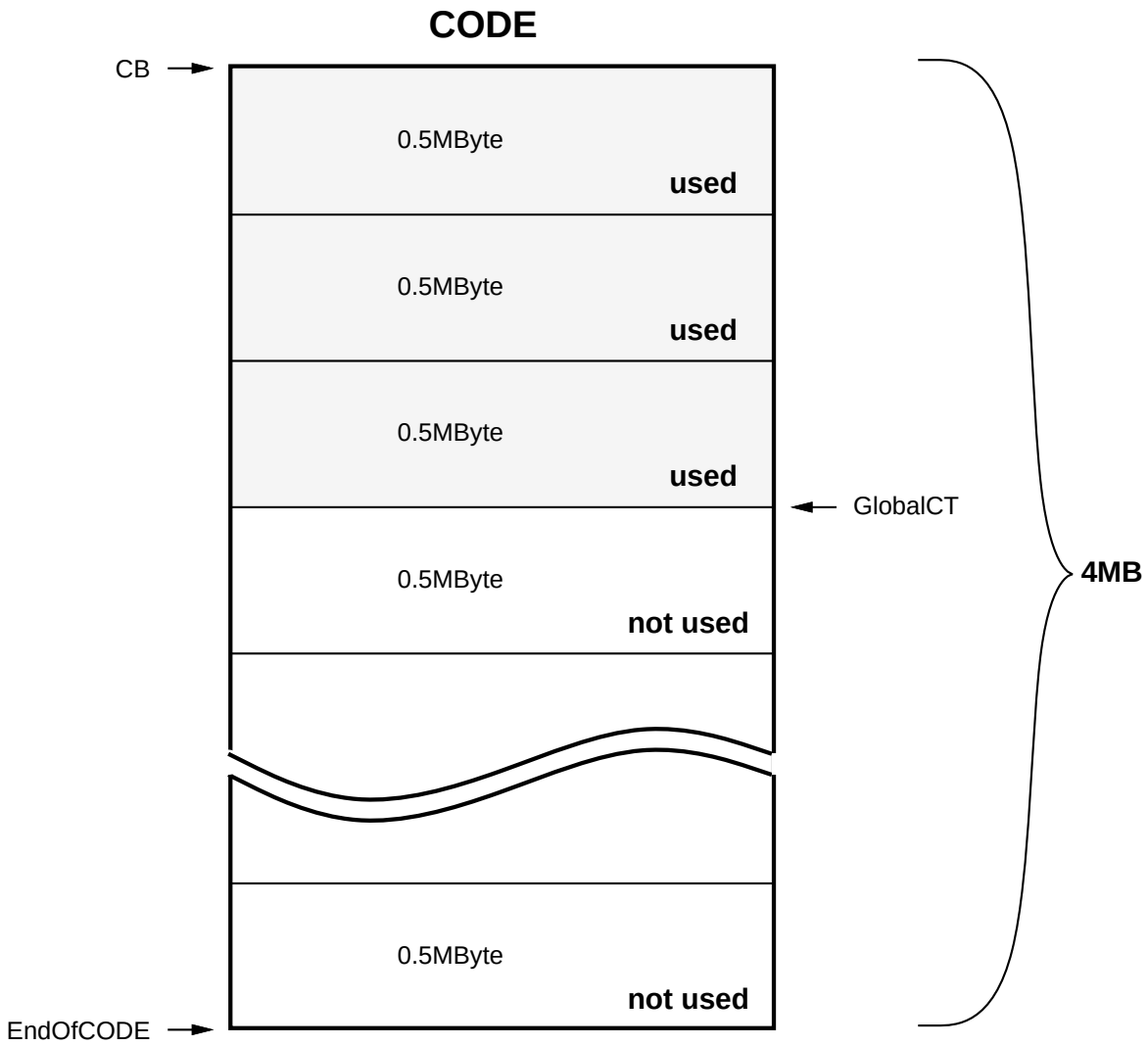

図 5.4: CODE 領域の管理

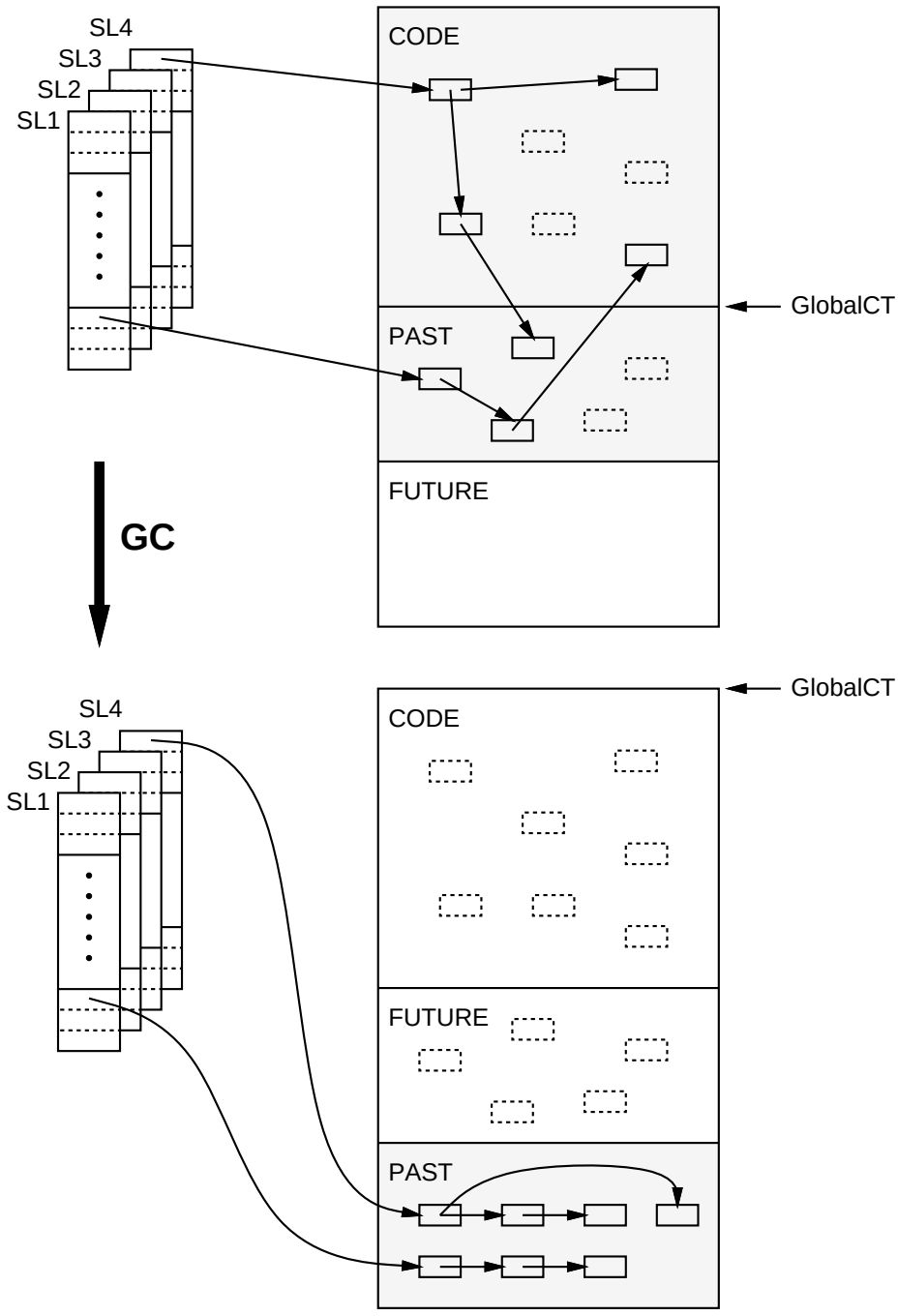

図 5.5: CODE 領域の GC

```
typedef enum{ UNLOCK, LOCK } Lock;   /* Lock 型の定義 */

void getLock( Lock *lock )  /* ロックの獲得 */
{
  unsigned int delay, gain, i;

  delat = gain = 1;
  while ( TRUE ) {
    if ( fetchAndStore(lock, LOCK) == UNLOCK )  return;
    while ( TRUE ) {
      for (i = 0; i < delay; i++)  ;   /* 空ループ（遅延） */
      if ( *lock == UNLOCK )
        break;
      delay += gain;
    }
  }
}

void releaseLock( Lock *lock )  /* ロックの解除 */
{
  *lock = UNLOCK;
}

-----------------------------------------------------------------------
;;; MC88100 での fetchAndStore の実現（アセンブラコード） ;;;
_fetchAndStore:
        xmem    r3, r2, 0        ;; 不可分な swap
        or      r2, r0, r3
        jmp     r1
```

図 5.6: test-and-test-and-set に遅延を導入したスピンロック

第 6 章

性能評価

この章では、並列 E-戦略に基づく（引数項の）並列簡約により、どの程度の効率改善が実現できたのかを、いくつかのベンチマークを実行させて評価を行なう。基本的な評価方法としては、LUNA-88k2 上で動作する Parallel TRAM と、同じく LUNA-88k2 上に移植した TRAM の実行速度を比較することによって行なうものとする。(一部そうでないものもある。)

6.1 ベンチマークおよび計測結果

今回評価を行なったベンチマークは、次の通りである。

1. 逐次簡約性能の評価

   (a) TRAM における 7 の階乗 fact(7) の計算。

   (b) Parallel TRAM、逐次簡約 における fact(7) の計算。

2. 最大性能の評価
   規則: `p(X,Y,Z,W) -> 0.  p:{ strat:(1 2 3 4 0) }`
   規則: `p'(X,Y,Z,W) -> 0.  p':{ strat:({1 2 3 4} 0) }`

   (a) TRAM における
   `p(fact(7),fact(7),fact(7),fact(7))` の計算。

(b) Parallel TRAM、プロセスユニット数 = 4、における
`p'(fact(7),fact(7),fact(7),fact(7))` の計算。

3. GC オーバーヘッドの計測

(a) Parallel TRAM、プロセスユニット数 = 4、コード領域の容量 = 4MB、における fib(24) の計算。

(b) Parallel TRAM、プロセスユニット数 = 4、コード領域の容量 = 24MB、における fib(24) の計算。

4. 基本性能の評価

(a) TRAM におけるフィボナッチ数列 fib(25) の計算。

(b) Parallel TRAM、プロセスユニット数 = 2、におけるフィボナッチ数列 fib(25) の計算。(plus の演算子に `{ strat: ({1 2} 0) }` を指定)

(c) Parallel TRAM、プロセスユニット数 = 3、におけるフィボナッチ数列 fib(25) の計算。(plus の演算子に `{ strat: ({1 2} 0) }` を指定)

(d) Parallel TRAM、プロセスユニット数 = 4、におけるフィボナッチ数列 fib(25) の計算。(plus の演算子に `{ strat: ({1 2} 0) }` を指定)

(e) Parallel TRAM、プロセスユニット数 = 4、におけるフィボナッチ数列 fib(25) の計算。(plus の演算子に `{ strat: ({2 1} 0) }` を指定)

(f) TRAM におけるクイックソート(要素数 400)。

(g) Parallel TRAM、プロセスユニット数 = 4、におけるクイックソート(要素数 400)。(append の演算子に `{ strat: ({1 2} 0) }` を指定)

(※) 上記ベンチマークの詳細は、付録 B に添付する。

また、これらベンチマークに対する計測結果を表 6.1に示す。

6.2 考察

6.2.1 逐次簡約性能の評価

逐次簡約の指定をした７階乗の計算を、TRAM と Parallel TRAM に行なわせ、その実行時間を比較してみた。その結果、Parallel TRAM は TRAM とほぼ同じ速さ（0.96 倍）で処理を実行することができ、逐次簡約に関しては、TRAM と比較してもほとんど遜色のない処理能力を有していることが確認できた。

6.2.2 最大性能の評価

実装したParallel TRAMが、設計意図通りのものに仕上がっているかを確認するには、最大性能の評価を行なうのが適切と考え、この評価を行なってみた。評価の仕方は、独立した全く同じ仕事量の簡約（ここでは、1-(b) の７階乗をやらせてみた）を、それぞれのプロセスユニット上で並列に行なうというものである。理論的には、プロセスユニット数と同じ倍率の効率が得られるはずである。しかしながら、実際に得られた結果は、ユニット数４のときで３倍という結果であった。当初は、GC の回数もそれほど大差ないため、FORK のオーバーヘッドが原因だと考えていた。ここで、FORK のオーバーヘッドを簡単に計算してみると次のようになる。

理論的には、1-(b) と同じ実行時間になるはずである。

$$\text{その差} \ = \ 68.81 - 52.6 \ = \ 16.21$$

また、2-(b) では３回の FORK が行なわれている。従って、

$$FORK \text{ のオーバーヘッド} \ = \ 16.21 \div 3 \ = \ 5.4 \, sec \ ??$$

これは、非常に信じがたい結果である。なぜなら、”fact(7)” で生成されるマッチングプログラムは３ワード、戦略リストにいたってはわずかの２ワードで、高々５ワードの領域をコピーするのに秒のオーダーがかかるとは考えられないからである。これには、別の原因がからんでいる可能性が非常に高い。それを裏付けるために、fact(7) を fact(6) に代えて同じように FORK のオーバーヘッドを計算してみた。（コピー量は fact(6) も fact(7) と同じ。）すると、FORK のオーバーヘッドの値として 137msec という結果を得た。もし、FORK のオーバーヘッドのみが関係しているのであれば、こんなにも大きくデータがバラツクことはあり得ない。

では、最大性能が劣化した本当の原因は何か。ここで、１つ大きなボトルネックがあることを説明しておかねばならない。それは、LUNA-88k2 のメモリバスは１つしか存在しないということである。つまりそれぞれのプロセッサは、メモリアクセスの際、１つしか存在しないメモリバスをめぐって激しい競合を起こしているわけである。この競合がメモリアクセス全体に対し、どれくらいの割合で起こっているかは分からないが、性能劣化の最大の原因をこのメモリアクセスの競合と考えると、先に示したデータもうまく説明することができる。

fact(7) の簡約も、fact(6) の簡約も同じ割合で競合が起こっていると仮定すると、実行時間に比例したロスタイムが発生する。

$$fact(6) \text{ の簡約時間} \quad = \quad 1.45\ sec$$

$$fact(6) \text{ を 4 つ並列に行なった時の簡約時間} \quad = \quad 1.86\ sec$$

$$\frac{1.86 - 1.45}{1.86} \quad \overset{nearly}{=} \quad \frac{68.81 - 52.6}{68.81}$$

この仮説が本当に正しいかどうかは、さらに詳しく検証する必要があるが、メモリアクセスの競合が何らかの形で性能劣化に関係していることはほぼ間違い無いと考えられる。

6.2.3 GC にかかるオーバーヘッドの計測

CODE 領域を大きく確保し、意図的に全く GC させないようにして簡約を行なったものと、普通に簡約を行なったものの実行時間を計測し、GC にかかるおおよそのオーバーヘッドを算出してみた。その結果は次のようになる。

$$GC\text{のオーバーヘッド} \quad = \quad \frac{6.69 - 5.3}{6} \quad = \quad 232\ msec$$

（※）一般にコピー方式の GC の場合、GC 領域の容量に比例したオーバーヘッドがかかってくる。（コピー量が増える為。）Parallel TRAM は通常状態で 4MB の CODE 領域を確保しているため、今回の算出も 4MB 時の簡約時間を基準に算出している。（CODE 領域を意図的に小さくして GC 回数を増やし、割算の分母を大きくした方が算出結果の精度が上がるというわけではない。）また、Parallel TRAM で採用した GC は、グローバルな同期を取って１つのプロセッサが GC するというものなので、前節で説明したようなメモリアクセスの競合は一切起こらない。

6.2.4 基本性能の評価

プロセスユニット数を２～４に変化させて実行速度を計測してみると、

	ユニット数 2	ユニット数 3	ユニット数 4
speed up	1.42	1.63	1.97

という結果を得る事ができた。それでは、プロセスユニット数４の場合のデータに着目して、この速度向上が妥当なものであるのか検証してみることにする。

書換え回数を簡約時間と等価であると見なし、まず、Parallel TRAM の並列簡約メカニズムが理想的な形で実行された場合の速度向上を算出してみる。

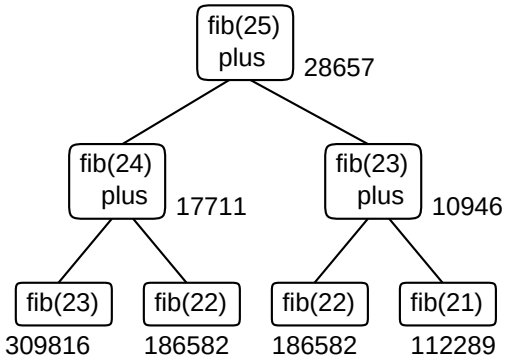

図 6.1: 理想的な fib(25) の計算

図 6.1からも分かるように、最も理想的な形で fib(25) の計算が行なわれたとすると、

1. fib(23)、fib(22)、fib(22)、fib(21)、の簡約が４つのプロセッサ上で並列に行なわれる。

2. fib(23) と fib(22) の簡約結果を plus する
   fib(22) と fib(21) の簡約結果を plus する
   という２つの簡約が２つのプロセッサ上で並列に行なわれる。

3. fib(24) と fib(23) の簡約結果を plus する。

といったステップで簡約が進むはずである。それぞれのステップで必要な書換え回数を計算してみると次のようになる。

1. ４つの簡約の中で最も書換え回数の多い fib(23) の書換え回数が必要と考えがちだが、fib 自体 非常に高い並列性を持っているため、早く簡約が終了したプロセッサ

には、未終了簡約の部分簡約をさらに割り付けることができる。従って理想的には４つの書換えは平均化されることになり、

$$1.\text{で必要な書換え回数} \quad = \quad \frac{309816 + 186582 + 186582 + 112289}{4} \quad = \quad 198817$$

となる。

2. plus 演算子の全体項簡約に並列性はないため、２つの簡約のうち処理の長い方、すなわち、17711 回の書換えが必要となる。

3. いうまでもなく 28657 回の書換えが必要である。

従って、理想的な書換え回数は、

$$\text{理想的な書換え回数} \quad = \quad 198817 + 17711 + 28657 \quad = \quad 245185$$

となる。一方、逐次に fib(25) を行なった場合の書換え回数は 852580 回なので、理想的な場合の速度向上は、

$$\text{理想的な場合の速度向上} \quad = \quad \frac{852580}{245185} \quad = \quad 3.48$$

となる。

この理想的な速度向上と、実際の速度向上を単純に比較するのは、確かに重要ではあるけれども、あまりにも無茶である。なぜなら、LUNA-88k2 の実装では 6.2.2 節で述べたようなボトルネックが生じているからである。そこで、もう少し現実的な比較を行なうためにこの理想的な値を補正する。すると、

$$\text{理想的な速度向上 (補正後)} \quad = \quad 3.48 \times \frac{3.02}{4} \quad = \quad 2.62$$

となる。

理想的な速度向上が 2.62 に対して、実際の速度向上は 1.97 である。速度向上がそれほど伸びなかったのは、やはり、FORK にかかるオーバーヘッドによるものと思われる。今回は FORK のオーバーヘッドを算出する良いベンチマークを思いつくことができなかったため、はっきりと断言することはできない。しかし、これら速度向上の値から逆に FORK のオーバーヘッドを概算することは可能である。実際に計算してみると次のようになる。FORK にかかる（とおそらく思われる）オーバーヘッドがなかった場合は、2.62 の速度向上が得られるわけだから、次のような方程式を得ることができる。

$$\frac{23.02}{11.66 - 3320x} \quad = \quad 2.62$$

この方程式を解いて、FORK のオーバーヘッドはおよそ 0.87msec ではないかと予想される。

最後に、4-(e) の計測結果について少し触れておく。このデータと 4-(d) のデータを比較すると、FORK の成功回数に大きな差があるのが確認できる。このデータは、フィボナッチ数列における plus 演算子の並列指定を逆にして計測したもので、演算子 plus が２引数の演算子であることを考えると、これは 他ユニット上に FORK する簡約を入れ換えたことに等しくなる。つまり、4-(d) では fib(s(X)) を FORK し、4-(e) では fib(X) を FORK するわけである。なぜこのような現象が起こったかについては、どうやら fib(s(x)) の簡約と fib(X) の簡約の処理の重さに原因があるようである。この２つの処理の重さを比較すると、当然ながら fib(s(X)) の方が重い。従って、4-(d) の方が 4-(e) に比べ JOIN で待機状態（idle）に入る可能性が高くなり、結果的に他ユニットからの FORK を受け入れ易くなったわけである。Parallel TRAM には特に複雑なスケジューラーは組み込まれてないが、並列簡約の指定を工夫すれば、ある程度 並列性を制御することが可能であると考えられる。

ベンチマーク	時間 (s)	書換え回数	R/S	GC 回数	FORK 成功回数	FORK 失敗回数	speed up
1-(a)	50.72	1857927	36631	23	-	-	基準
1-(b)	52.6	1857927	35281	23	0	0	0.96
2-(a)	207.69	7431709	35782	92	-	-	基準
2-(b)	68.81	7431709	108003	94	3	0	3.02
3-(a)	6.69	514108	76847	6	2308	72716	-
3-(b)	5.3	514108	97001	0	2241	72783	-
4-(a)	23.02	852580	37036	9	-	-	基準
4-(b)	16.26	852580	52434	10	66	121326	1.42
4-(c)	14.09	852580	60638	12	3041	118351	1.63
4-(d)	11.66	852580	73120	10	3320	118072	1.97
4-(e)	11.71	852580	72807	10	863	120529	1.96
4-(f)	6.47	242698	37511	1	-	-	基準
4-(g)	3.87	242698	62712	2	138	162	1.67

R/S： 1秒あたりの書換え回数

FORK 成功回数： 実際に FORK を行なった数

FORK 失敗回数： idle 状態のプロセスユニットが無く、FORK を断念した回数

表 6.1: 計測結果

第 7 章

おわりに

本研究では、単一プロセッサ上での使用を想定して設計された項書換え抽象機械である ”TRAM” に、マルチプロセッサにおいて引数項の並列簡約を行なうためのメカニズムを組み込み、”Parallel TRAM” の設計を完成させた。また、この設計完了した Parallel TRAM を、LUNA-88k2（MC88100 ×４台搭載、オムロン社製）に実装し、種々の評価を行なった。

7.1 まとめ

本研究で得た結論をまとめると次のようになる。

7.1.1 書換え戦略（並列 E-戦略）

TRAM が採用している E-戦略では、並列簡約を指定するための記述方法が存在しなかった。そこで E-戦略のシンタックスに、ユーザーが陽に引数項の並列簡約を指定できるような構文を追加し、並列 E-戦略なる戦略を新たに定義した。この並列 E-戦略を導入したことにより、ユーザは E-戦略の特徴であった”lazy、eager な簡約のローカルな制御” だけでなく、引数項の並列簡約を明示的に指定できる結果として、過度の並列性拡大を抑制することもできるようになった。

7.1.2 並列簡約の実現（Parallel TRAM の設計）

並列簡約を実現するために新たに追加された抽象命令は、FORK、JOIN、EXIT、SLEEP、NOP の５命令である。ユーザーにより指定された並列簡約は、戦略リストにこれら５命令を呼び出すためのラベルを埋め込むことで実現される。また、この５命令以外の命令は、全て TRAM で用いられる抽象命令を、意味定義を一切変えることなく用いているため、TRAM に組み込まれた様々な要素技術や長所をそのまま受け継いだ形で並列簡約を実現することが可能となった。

7.1.3 Parallel TRAM の実装

プロセスユニット（抽象機械上で簡約処理を行なうための１実行ユニット）をスレッドと等価と見なし、実装する計算機（LUNA-88k2）のプロセッサ数と同じ数のスレッドを生成する方針で実装を行なった。その結果、かなり理想的な形でプロセスユニットの概念を実現することが可能となった。またこれ以外にも、プロセスユニット状態の管理、メモリ領域の管理、ガベージコレクション、クリティカルセクションを保護するロック機構 などを効率良く行なうための工夫をいくつか盛り込み、かなり精度の高い実装を行なうことができた。

7.1.4 性能評価

逐次簡約性能の評価、最大性能の評価、GC のオーバーヘッド計測、基本性能の評価 という４つの項目について、実装した Parallel TRAM の評価を行なった。LUNA-88k2 の持つ大きなボトルネック（メモリバスが１つしか存在しないために生ずるメモリアクセスの競合）の関係で、約 25%ほどの性能劣化を余儀なくさせられたものの、プロセスユニット数４の場合におけるフィボナッチ数が 1.97 倍、クイックソートが 1.67 倍 という速度向上を確認することができ、Parallel TRAM の並列簡約メカニズムでかなりの効率改善が期待できることを実証することができた。また、各演算子に指定する指定戦略を工夫することで、並列性の拡大を実際に制御できることが確認できた。

7.2 今後の課題

本研究において実施できなかったことも含め、今後の課題をまとめると、次のようになる。

7.2.1 より緻密な並列簡約

現在の Parallel TRAM で実行させることのできる並列簡約は、並列に簡約させた全ての簡約が終了するまで同期を取って待機するという比較的簡単な並列簡約だけである。これをさらに発展させ、より緻密な並列簡約を実現できれば、Parallel TRAM の実行効率はさらに良くなるものと考えられる。

投機的な簡約の並列化

投機的な簡約とは、if 式に代表される必要な簡約と不必要な簡約を前もって特定できない簡約（if 式の場合、条件節の簡約結果が出ないと、then 節、else 節のどちらの簡約が有効になるか判定できない）のことで、現在の Parallel TRAM で if 式を簡約するには、マシンパワーの浪費を覚悟で３つの引数を並列に簡約させるか、あるいは並列簡約をあきらめて逐次に簡約させるかのどちらかを選択するしか方法がない。もし、３つの引数を並列に簡約させておいて、条件節の結果が出た時点で必要な簡約だけを残し、不必要な簡約を破棄するようなメカニズムを組み込めれば、投機的な簡約を非常に効率良く実行することが可能となる。このような並列簡約を実現するには、現在の並列 E-戦略にさらに投機的簡約のためのシンタックスを追加し、簡約を途中で破棄させるための抽象命令を付け加える必要があると考えられる。

非決定的な書換えの並列化

４章の初めでも説明したように、これは今回の Parallel TRAM では対象としなかったもう一つの並列性である。一般的には非決定的な書換えを並列に行なわせてもそれほどの効率改善は期待できない。しかし、非常に特別な場合においては、効率の向上を期待することができると考えられる。そういった特例対応への必要性の議論を抜きにして考えた場合、この非決定的な書換えを並列に行なわせるというのは興味深い話である。

7.2.2 設計仕様の形式化

本来これは設計段階で行なわれなければならないのだが、今回設計した Parallel TRAM を仕様記述言語を用いて形式化し、本当に正しく動作するのかを確認する必要がある。特に並列計算を行なう場合には、デッドロック等の不具合が起こる可能性が出てくるため、動作の正当性を検証しておくことは、非常に重要である。

7.2.3 他の並列計算機への実装

今回の実装の目的は、Parallel TRAM の並列簡約メカニズムでどの程度の効率改善が実現できるのかを見極めることであったため、LUNA-88k2 という比較的手軽な並列計算機を選択し実装を行なった。しかし手軽に扱える分、細かな制御を行ないづらいという欠点もある。Parallel TRAM で効率の改善を十分に実現できることが確認できた今、さらに大型の並列計算機に、細かなチューニングを施しながら実装するのは非常に意義のあることだと考えられる。

7.2.4 他の並列項書換えシステムとの比較

今回の評価では、本研究のベースである TRAM との比較しか行なうことができなかった。Parallel TRAM の実力を正確に把握するには、他の並列項書換えシステムとの比較を客観的に行ない、それぞれの性能を冷静に分析する必要がある。

謝辞

本研究を行なう機会を与えて下さり、終始ご指導下さった二木厚吉教授に心から感謝致します。また、有益な助言をして下さった渡部卓雄助教授、緒方和博先生、Răzvan Diaconescu 先生に感謝致します。特に、緒方和博先生は TRAM の設計者であり、今回の Parallel TRAM 設計の際にも色々と相談に乗って頂きました。ここに深くお礼を申し上げます。最後に五百蔵重典氏をはじめとする博士後期過程の先輩方、共に研究活動に打ち込み励まし合ってきた言語設計学講座の皆様にお礼を申し上げます。

1997 年 早春

情報科学研究科棟にて

近藤 勝

参考文献

[1] Alfred V.Aho, Margaret J.Corasick. Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM, Vol.18, No.6, June 1975.

[2] 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.

[3] Barson R.V., Black D., Bolosky W.,Chew J., Draves R.P., Golub D.B., Rashid R.F., Tevanina Jr.A., Young M.W., MACH Kernel Interface Manual, Department of Computer Science Carnegie-Mellon University, 1990.

[4] Christoph M.Hoffmann, Michael J.O'donnell. Pattern Matching in Trees. Journal of the Association for Computing Machinery, Vol.29, No.1, pp.68-95, January 1982.

[5] Christoph M.Hoffmann, Michael J.O'donnell. Programming with Equations. ACM Transactions on Programming Language and System, Vol.4, No.1, pp.83-112, January 1982.

[6] David A.Kranz, Robert H.Halstead,Jr, Eric Mohr. Mul-T: A High-Performance Parallel Lisp. SIGPLAN '89 Conference on Programming Language Design and Implementation, Vol.24, No.7, July 1989.

[7] Eric C.Cooper, Richard P.Draves, C Threads, Department of Computer Science Carnegie-Mellon University, 1990.

[8] Francis CAUDAL, Bernard LECUSSAN. Design and Evaluation of a Multi-threaded Architecture for Parallel Graph Reduction. Parallel Computing Technologies, volume 964 of Lecture Notes in Computer Science, springer, 1995.

[9] J.F.Th.Kamperman, H.R.Walters. ARM(Abstract rewriting machine). ASF+SDF Group, University of Amsterdam, 1993.

[10] Kazuhiro Ogata, Koichi Ohhara, Kokichi Futatsugi. Term Rewriting Abstract Machine for Implementing Algebraic Specification Language on Stock Hardware. JAIST.

[11] Kazuhiro Ogata, Koichi Ohhara, Kokichi Futatsugi. TRAM: An Abstract Machine for Order-Sorted Conditional Term Rewriting Systems. JAIST.

[12] Kazuhiro Ogata, Koichi Ohhara, Kokichi Futatsugi. Representing Terms as Pattern-Matching Programs. JAIST.

[13] Kazuhiro Ogata, Koichi Ohhara, Kokichi Futatsugi. Optimization of Term Rewriting Abstract Machine with Continuations. JAIST.

[14] Robert H Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. ACM Transactions on Programming Languages and Systems, Vol.7, No.4, October 1985.

[15] Simon L. Peyton Jones, Chris Clack and Jon Salkild. High-performance Parallel Graph Reduction. PARLE '89 Parallel Architectures and Languages Europe, volume 365 of Lecture Notes in Computer Science, Springer-Verlag, 1989.

[16] Simon L.Peyton Jones. Implementing lazy functional language on stock hardware: the Spineless Tagless G-machine. Cambridge University Press, Functional Programming 2(2), pp.127-202, April 1992.

[17] Yoshihito Toyama. Fast Knuth-Bendix completion with a term rewriting system compiler. Information Processing Letter 32, pp.325-328, 1989.

[18] 二木 厚吉, 外山 芳人. 項書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.

[19] 大原 幸一, 緒方 和博, 二木 厚吉. 項書換えシステムのための抽象機械の設計について. 情報処理学会 第 51 回全国大会 (5), pp.41-42, 1995.

[20] 笠原 博徳. 並列処理技術. コロナ社, 1991.

[21] 戸村 哲. TRS Compiler I:正規簡易戦略式に基づく項書換え系コンパイラ. プログラミング言語システム「つくばね」における言語処理系構成法の研究 第 4 章, pp.68-90.

第 A 章

共通に使用される抽象命令の定義一覧 (TRAM/Parallel TRAM)

match_symbol 命令

現在着目しているノードから、オペランドに指定されたシンボルのスロットをたどって次のノードに遷移する命令。変数スロットを通過する時は、変数の束縛も行なう。

```
match_symbol( idx:シンボルのインデックス No. )
{
  if ( DNET[CN + idx + 1] = NULL )
    if ( DNET[CN] = NULL ) {
      /* idx スロットも変数スロットもたどれない */
      P ← GO_AHEAD;
    } else {
      /* 変数スロットのみたどれる */
      SV[SVP] ← P;
      DNET[CN + 1] ← SVP;
      SVP ← SVP + 1;
      CN ← DNET[CN];
      nextP( );
    }
  } else if ( DNET[CN] = NULL ) {
```

```
    /* idx スロットのみたどれる */
    CN ← DNET[CN + idx + 1];
    nextP( );
  } else {
    /* idx スロットも変数スロットもたどれる */
    SV[SVP] ← P;
    DNET[CN + 1] ← SVP;
    SVP ← SVP + 1;
    STACK[SP] ← DNET[CN];
    STACK[SP + 1] ← RP;
    SP ← SP + 2;
    CN ← DNET[CN + idx + 1];
    nextP( );
  }
}
```

nextP 命令（補助命令）

マッチングプログラム（木構造）の leaf に到達したかどうかを調べ、到達していたら return する。到達していなければ次のマッチングプログラムに制御を移す。

```
nextP( )
{
  if ( arity( P ) = 0 ) {
    P ← RP;
  } else {
    P ← P + 1;
  }
}
```

call 命令

オペランドの番地に制御を移し、その処理が終了したら元の番地に戻ってくるサブルーチンコール命令。

```
call( L:アドレス )
{
  RP ← P + 1;
  P ← L;
}
```

init_trs 命令

１回の書換えで使用する各種レジスタを初期化する。

```
init_trs( )
{
  CN ← root;
  SP ← BOS;
  FD ← 0;
  P ← P + 1;
}
```

e_next 命令

戦略リストの先頭要素を１つ取り出し、そのアドレスに制御を移す。

```
e_next( )
{
  ST ← ST - 3;
  RP ← P + 1;
  P ← SL[ST];
}
```

go_ahead 命令

バックトラックを起こし、適用可能な規則を全て見つけ出す。見つけ出した規則は、STACK 領域の EOS から下位アドレスに向かって割り付ける。適用可能な規則が１つも無い場合は LOOP に制御を移す。

```
go_ahead( )
{
  if ( CN が右辺の雛型を指している ) {
    STACK[EOS - FD] ← CN;
    FD ← FD + 1;
  }
  if ( SP ≠ BOS ) {
    SP ← SP - 2;
    CN ← STACK[SP];
    P ← STACK[SP + 1];
  }
  if ( FD = 0 ) {
    P ← LOOP;
  } else {
    P ← P + 1;
  }
}
```

select 命令

適用可能な規則の中から規則を１つ選択する。

```
select( )
{
  FD ← choose( FD );
  if ( FD ≠ NULL ) {
    P ← P + 1;
  } else {
    P ← LOOP;
  }
}
```

関数 choose で書換えに用いる規則を１つ選ぶ。このとき、規則に複数回現れる変数に、等価な部分項が束縛されているかどうかのチェックも同時に行なう。従ってこの段階で、適用できる規則が実は１つも存在しないと判明する場合がある。このような場合、関数 choose は NULL を返す。

e_rewrite 命令

リデックスを、select 命令で選択した規則の右辺で置き換える。

```
e_rewrite( )
{
  RHS ← STACK[EOS - FD];
  X ← CT;
  instantiate( RHS );
  CODE[ SL[ST + 1] ] ← "call X"; /* LHS → RHS の置き換え */
  slmerge( RHS, SL[ST + 2] );
}
```

instantiate(RHS)：CR 領域、RHS アドレスに格納されたマッチングプログラムの雛型の変数を具現化し、CODE 領域の空き領域（アドレス CT）に割り付ける。

slmerge(RHS)：CR 領域、RHS アドレスに格納された戦略リストの雛型を具現化し、SL 領域に割り付ける。

goto 命令

オペランドのアドレスに制御を移す。

```
goto( L:アドレス )
{
  P ← L;
}
```

bingo 命令

現在のマッチングプログラムを簡約結果として出力する。

```
bingo( )
{
  convert( SL[ST + 1] );
  /* SL[ST+1] に項のルートアドレスが格納されている */
}
```

convert() は、マッチングプログラムを項の表記に変換する手続きである。

第 B 章

ベンチマーク一覧

B.1 フィボナッチ数列

```
sorts: Nat.
order: .
operators:
    0 : -> Nat
    24 : -> Nat
    25 : -> Nat
    s : Nat -> Nat
    10 : Nat -> Nat
    plus : Nat Nat -> Nat
    fib : Nat -> Nat
vars: X Y : Nat.
rules:
    24 -> 10(10(s(s(s(s(0))))))
    25 -> 10(10(s(s(s(s(s(0)))))))
    10(X) -> s(s(s(s(s(s(s(s(s(s(X))))))))))
    plus(X, 0) -> X
    plus(X, s(Y)) -> s(plus(X, Y))
    fib(0) -> 0
    fib(s(0)) -> s(0)
    fib(s(s(X))) -> plus(fib(s(X)),fib(X)).
```

B.2 階乗の計算

```
sorts: Nat.
order: .
```

```
operators:
    0 : -> Nat
    6 : -> Nat
    7 : -> Nat
    s : Nat -> Nat
    plus : Nat Nat -> Nat
    times : Nat Nat -> Nat
    minus : Nat Nat -> Nat
    fact : Nat -> Nat.
vars: X Y : Nat.
rules:
    6 -> s(s(s(s(s(0))))))
    7 -> s(s(s(s(s(s(0)))))))
    plus(X, 0) -> X
    plus(X, s(Y)) -> s(plus(X, Y))
    times(X, 0) -> 0
    times(X, s(Y)) -> plus(X, times(X, Y))
    fact(0) -> s(0)
    fact(s(X)) -> times(s(X), fact(X)).
```

B.3 クイックソート

```
sorts: Bool Nat List .
order: .
operators:
    true : -> Bool
    false : -> Bool
    0 : -> Nat
    s : Nat -> Nat
    nil : -> List
    c : Nat List -> List
    sort : List -> List
    qsort : List -> List
    partition : Nat List List List -> List
    leq : Nat Nat -> Bool
    append : List List -> List

    list100 : -> List
    list200 : -> List
    list300 : -> List
    list400 : -> List

    1 : -> Nat    2 : -> Nat    3 : -> Nat    4 : -> Nat    5 : -> Nat
```

```
    6 : -> Nat    7 : -> Nat    8 : -> Nat    9 : -> Nat    10 : -> Nat
    11 : -> Nat   12 : -> Nat   13 : -> Nat   14 : -> Nat   15 : -> Nat
    16 : -> Nat   17 : -> Nat   18 : -> Nat   19 : -> Nat   20 : -> Nat
    21 : -> Nat   22 : -> Nat   23 : -> Nat   24 : -> Nat   25 : -> Nat
    26 : -> Nat   27 : -> Nat   28 : -> Nat   29 : -> Nat   30 : -> Nat
    31 : -> Nat   32 : -> Nat   33 : -> Nat   34 : -> Nat   35 : -> Nat
    36 : -> Nat   37 : -> Nat   38 : -> Nat   39 : -> Nat   40 : -> Nat
    41 : -> Nat   42 : -> Nat   43 : -> Nat   44 : -> Nat   45 : -> Nat
    46 : -> Nat   47 : -> Nat   48 : -> Nat   49 : -> Nat   50 : -> Nat
    51 : -> Nat   52 : -> Nat   53 : -> Nat   54 : -> Nat   55 : -> Nat
    56 : -> Nat   57 : -> Nat   58 : -> Nat   59 : -> Nat   60 : -> Nat
    61 : -> Nat   62 : -> Nat   63 : -> Nat   64 : -> Nat   65 : -> Nat
    66 : -> Nat   67 : -> Nat   68 : -> Nat   69 : -> Nat   70 : -> Nat
    71 : -> Nat   72 : -> Nat   73 : -> Nat   74 : -> Nat   75 : -> Nat
    76 : -> Nat   77 : -> Nat   78 : -> Nat   79 : -> Nat   80 : -> Nat
    81 : -> Nat   82 : -> Nat   83 : -> Nat   84 : -> Nat   85 : -> Nat
    86 : -> Nat   87 : -> Nat   88 : -> Nat   89 : -> Nat   90 : -> Nat
    91 : -> Nat   92 : -> Nat   93 : -> Nat   94 : -> Nat    95 : -> Nat
    96 : -> Nat   97 : -> Nat   98 : -> Nat   99 : -> Nat   100 : -> Nat .

vars:
    X Y Pivot : Nat
    L L1 Large Small : List .

rules:
    sort(L) -> qsort(L)
    qsort(nil) -> nil
    qsort(c(X, nil)) -> c(X, nil)
    qsort(c(X, c(Y, L))) -> partition(X, c(Y, L), nil, nil)
    partition(Pivot, nil, Small, Large)
             -> append(qsort(Small), c(Pivot, qsort(Large)))
    leq(Pivot, X) = true =>
        partition(Pivot, c(X, L), Small, Large)
                  -> partition(Pivot, L, Small, c(X, Large))
    leq(Pivot, X) = false =>
        partition(Pivot, c(X, L), Small, Large)
                  -> partition(Pivot, L, c(X, Small), Large)
    append(nil, L) -> L
    append(c(X, L1), L) -> c(X, append(L1, L))

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

```
list100 -> c(61, c(4, c(83, c(59, c(20, c(28, c(30, c(35, c(68,
           c(52, c(63, c(50, c(33, c(89, c(67, c(70, c(16, c(57,
           c(91, c(3, c(29, c(92, c(87, c(22, c(12, c(44, c(74,
           c(66, c(5, c(1, c(31, c(26, c(13, c(49, c(71, c(78,
           c(45, c(77, c(94, c(51, c(24, c(73, c(34, c(75, c(9,
           c(65, c(18, c(58, c(0, c(8, c(62, c(2, c(40, c(97,
           c(19, c(54, c(60, c(37, c(23, c(86, c(69, c(85, c(42,
           c(39, c(53, c(96, c(27, c(55, c(76, c(32, c(95, c(43,
           c(17, c(41, c(11, c(10, c(84, c(64, c(99, c(6, c(88,
           c(72, c(46, c(98, c(25, c(36, c(90, c(38, c(21, c(56,
           c(7, c(14, c(81, c(80, c(82, c(15, c(48, c(93, c(79,
           c(47, nil)))))))))))))))))))))))))))))))))))))))))
           ))))))))))))))))))))))))))))))))))))))))))))))))))

list200 -> c(89, c(9, c(87, c(27, c(25, c(53, c(34, c(62, c(0,
           c(4, c(74, c(35, c(76, c(24, c(70, c(83, c(57, c(16,
           c(79, c(22, c(21, c(80, c(46, c(15, c(85, c(36, c(82,
           c(6, c(88, c(20, c(30, c(11, c(12, c(92, c(2, c(86,
           c(81, c(65, c(78, c(98, c(96, c(56, c(3, c(10, c(37,
           c(17, c(23, c(18, c(77, c(32, c(99, c(75, c(72, c(52,
           c(26, c(19, c(41, c(40, c(54, c(14, c(64, c(61, c(90,
           c(63, c(13, c(48, c(50, c(7, c(5, c(49, c(43, c(42,
           c(60, c(69, c(47, c(94, c(93, c(29, c(39, c(67, c(45,
           c(28, c(38, c(51, c(68, c(33, c(91, c(31, c(84, c(73,
           c(95, c(66, c(97, c(1, c(59, c(55, c(44, c(8, c(71,
           c(58, list100)))))))))))))))))))))))))))))))))))))))
           ))))))))))))))))))))))))))))))))))))))))))))))))))

list300 -> c(84, c(33, c(46, c(30, c(29, c(5, c(2, c(11, c(28,
           c(81, c(83, c(10, c(44, c(24, c(39, c(43, c(57, c(25,
           c(42, c(87, c(97, c(69, c(23, c(91, c(73, c(56, c(14,
           c(7, c(17, c(13, c(98, c(19, c(76, c(21, c(71, c(15,
           c(37, c(80, c(74, c(66, c(41, c(85, c(27, c(67, c(65,
           c(64, c(94, c(75, c(61, c(16, c(82, c(78, c(93, c(36,
           c(99, c(86, c(8, c(40, c(6, c(63, c(48, c(32, c(79,
           c(55, c(20, c(0, c(90, c(18, c(12, c(52, c(3, c(95,
           c(45, c(60, c(62, c(31, c(77, c(53, c(35, c(47, c(1,
           c(92, c(70, c(34, c(49, c(96, c(58, c(54, c(89, c(68,
           c(51, c(38, c(88, c(4, c(59, c(26, c(72, c(9, c(50,
           c(22, list200)))))))))))))))))))))))))))))))))))))))
           ))))))))))))))))))))))))))))))))))))))))))))))))))

list400 -> c(5, c(88, c(31, c(98, c(44, c(68, c(43, c(7, c(8,
           c(24, c(47, c(86, c(93, c(61, c(70, c(82, c(41, c(84,
```

```
            c(39, c(67, c(13, c(20, c(90, c(83, c(69, c(37, c(95,
            c(66, c(1, c(89, c(94, c(74, c(52, c(12, c(75, c(34,
            c(45, c(40, c(10, c(3, c(32, c(64, c(22, c(2, c(76,
            c(97, c(23, c(87, c(0, c(53, c(35, c(71, c(17, c(49,
            c(18, c(14, c(65, c(29, c(59, c(54, c(96, c(28, c(55,
            c(15, c(9, c(73, c(26, c(42, c(56, c(36, c(63, c(27,
            c(60, c(57, c(11, c(19, c(72, c(4, c(99, c(91, c(85,
            c(16, c(38, c(79, c(92, c(21, c(58, c(6, c(33, c(48,
            c(30, c(50, c(81, c(25, c(51, c(62, c(80, c(77, c(78,
            c(46, list300))))))))))))))))))))))))))))))))))))))
            ))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

```
1 -> s(0)    2 -> s(1)    3 -> s(2)    4 -> s(3)    5 -> s(4)    6 -> s(5)
7 -> s(6)    8 -> s(7)    9 -> s(8)    10 -> s(9)    11 -> s(10)
12 -> s(11)    13 -> s(12)    14 -> s(13)    15 -> s(14)    16 -> s(15)
17 -> s(16)    18 -> s(17)    19 -> s(18)    20 -> s(19)    21 -> s(20)
22 -> s(21)    23 -> s(22)    24 -> s(23)    25 -> s(24)    26 -> s(25)
27 -> s(26)    28 -> s(27)    29 -> s(28)    30 -> s(29)    31 -> s(30)
32 -> s(31)    33 -> s(32)    34 -> s(33)    35 -> s(34)    36 -> s(35)
37 -> s(36)    38 -> s(37)    39 -> s(38)    40 -> s(39)    41 -> s(40)
42 -> s(41)    43 -> s(42)    44 -> s(43)    45 -> s(44)    46 -> s(45)
47 -> s(46)    48 -> s(47)    49 -> s(48)    50 -> s(49)    51 -> s(50)
52 -> s(51)    53 -> s(52)    54 -> s(53)    55 -> s(54)    56 -> s(55)
57 -> s(56)    58 -> s(57)    59 -> s(58)    60 -> s(59)    61 -> s(60)
62 -> s(61)    63 -> s(62)    64 -> s(63)    65 -> s(64)    66 -> s(65)
67 -> s(66)    68 -> s(67)    69 -> s(68)    70 -> s(69)    71 -> s(70)
72 -> s(71)    73 -> s(72)    74 -> s(73)    75 -> s(74)    76 -> s(75)
77 -> s(76)    78 -> s(77)    79 -> s(78)    80 -> s(79)    81 -> s(80)
82 -> s(81)    83 -> s(82)    84 -> s(83)    85 -> s(84)    86 -> s(85)
87 -> s(86)    88 -> s(87)    89 -> s(88)    90 -> s(89)    91 -> s(90)
92 -> s(91)    93 -> s(92)    94 -> s(93)    95 -> s(94)    96 -> s(95)
97 -> s(96)    98 -> s(97)    99 -> s(98)    100 -> s(99) .
```