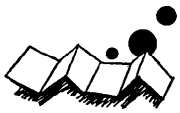
解　説



関数型言語とリダクションマシン†

小長谷　明　彦††　　山　本　昌　弘††

1. はじめに

近年，プログラムの生産性，透明性，検証の容易性などの観点から関数型言語が注目を集めている．しかしながら，関数型言語は通常のフォンノイマン型の計算機上では原理的に高速に実行できないという問題点を持つため[1]，関数型言語を指向した新しい原理に基づく非ノイマン型のアーキテクチャが活発に研究されている[5]．このような非ノイマン型のアーキテクチャとしてはデータ・フローマシンが良く知られているが，本稿ではこれとともに注目されているアプローチであるリダクションマシンについて解説する．

リダクションマシンでは関数型言語は次のようにして実行される．たとえば，関数fが以下のように定義されていたとする．

$\mathrm{def}\, f(x, y) = X*Y + X/Y$

そして，評価すべき式として

$f(4, 2)$

が与えられたとする．リダクションマシンは上記式を受け取ると仮引数を実引数に置き換えて関数呼び出しを関数本体での実行式に書き換える．

→4*2+4/2

次に，実行可能な組込命令を実行し，結果に書き換える．

→8+2

→10

これ以上，書き換えができないので，10を結果として返す．

リダクションとはこのような書き換えのステップをいう．そして，すべての計算を式の書き換えで実現しようというのがリダクションマシンである．このような関数型プログラムの実行方式は以下に示すような利点を持つといわれている．

（1） 計算の過程を人間が理解できる形式で示すことができる．

リダクションマシンの操作対象は式自身である．したがって，任意の時点でリダクションマシンを止めるとその時点で操作している式が得られる[12]．このことは通常の計算機が番地や数値を表わすビット列を操作していることに比べるときわめて高度な，ある意味では人間の思考に近いレベルで計算をしていると見ることができる．

（2） 言語モデルとの親和性が高い．

関数型言語の数学的な基礎になっているラムダ算法[2]，結合子算法[3]，Backus の fp[1]などは，リダクションを使って，計算の概念が定義されている．したがって，関数型言語とリダクションマシンでは言語モデルと実行モデルとのセマンティクス・ギャップが非常に小さい．たとえば，関数を引数として扱う機能（関数引数）や，引数の評価を必要になるまで遅らせる機能（遅延評価[5]）は通常の言語処理系で使用されるスタック・モデルで実現しようとすると相当の工夫を要するが，リダクション・モデルでは容易に実現される[16]．

（3） 並列処理に向いている．

書き換え規則は，相互作用のない限り独立に適用することが可能である．たとえば，上記例では 4*2 と 4/2 のリダクションを同時に行うことが可能である．また，ある種の条件が成り立てば，リダクションの順序によらずに同一の解が得られることが数学的に保証されている[2]ことも，関数型言語のプログラムを並列実行させることのよりどころの一つになっている．

これまでにリダクションマシンとして発表されたものとしては，外国では，西ドイツの RED マシン (GMD 大)[15]，米国の FFP マシン（ノースキャロライナ大)[27]，AMPS（ユタ大)[31]，イギリスの SKIM (ケンブリッジ大)[18]，ALICE（インペリアル・カレッジ)[25]，ニューキャッスル・リダクションマシン (ニューキャッスル大)[35]などがある．また，国内では，筆者らの提案した ARM（日電)[36]のほかには論理型

† Functional Languages and Reduction Machines by Akihiko KONAGAYA and Masayoshi YAMAMOTO (C & C Systems Laboratories NEC Corporation).

†† 日本電気(株) C & C システム研究所

言語を対象とした PIM-R (ICOT)[38]，京大リダクションマシン[39]の研究がある．その目的も並列実行を狙ったもの[27),25),35),36),38),39]，VLSI アーキテクチャ[10]を狙ったもの[27),36]，遅延評価付きのインタプリタの高速実行を狙ったもの[18),31]とさまざまである．ただ，リダクションマシンの研究は始まったばかりであり，現時点でリダクションマシンの良し悪しを評価することは難しい．本稿では 2. でリダクションマシンにまつわる諸概念の整理を行い，3. で典型的なリダクションマシンのイメージについて述べ，また，4. でリダクションマシンの主な研究事例について紹介する．ただし，論理型言語に基づくリダクションマシンについては，本稿の主題である関数型言語から外れてしまうこと，および，関数型言語におけるリダクションと論理型言語におけるリダクションとで少し考え方が異なることから，本稿では割愛させて頂くことにする．

2. リダクション・モデルとリダクション戦略

本章では関数型言語の実現モデルとしてのリダクション・モデルの考え方を，コントロールフロー・モデル，データフロー・モデルとの違いから述べる．さらに，データ駆動，要求駆動といった命令駆動方式とリダクションとの関係，および，ストリング・リダクションとグラフ・リダクションとの違いをリダクション戦略の観点から言及する．

2.1 コントロールフロー・モデル，データフロー・モデル，リダクション・モデル

同様な分類で各計算モデルの比較を行った解説として Treleaven の解説[8]があるが，本稿では関数型言語の実現モデルという観点から各モデルの違いを述べる．

(1) コントロールフロー・モデル

コントロールフロー・モデルは命令の実行順序を指定したコントロールフローに従って計算が進められる計算モデルであり，通常の（フォンノイマン型）計算機および PASCAL や FORTRAN といった命令指向のプログラム言語がこのモデルに分類される．コントロールフロー・モデルの特徴は副作用によるデータ格納を基本とするメモリ語とコントロールフローの中での現在実行中の命令を指すプログラム・カウンタに集約される．

一方，関数型言語では，プログラムは入力に対する結果を規定しているだけであり，プログラム・カウンタやメモリ語に相当する概念はない．すなわち，関数適用 $f(g(x),h(x))$ があったとき，関数 f から実行するか，g あるいは h から実行するかは処理系の解釈の自由である*．また，変数の値の結合は一度であり，メモリ語のように副作用の対象とはならない**．

このように，関数型言語とコントロールフロー・モデルは計算原理が大きく異なるため，関数型言語をコントロールフロー・モデル（ノイマン型計算機）の上で実現するためには SECD マシン[43]のように再帰的な関数呼び出しや変数領域を関数呼び出しごとに，動的にアロケートするような機構が必要になる．

(2) データフロー・モデル

データフロー・モデル[4),41]は命令の実行順序を規定するのはコントロールフローではなく各命令間のデータの移動経路を示すデータフローであるという観点から考案された計算モデルである．データフロー・モデルにはさまざまな定義が考えられるが，リダクション・モデルとの違いを明確にするという立場から，トークン・モデル[41]を考えよう．トークン・モデルでは 図-1 に示すようにプログラムはノードに命令を持つ有効グラフで示される．各矢印はデータの依存性（データフロー）を示しておりグラフ上をトークンが移動することにより計算が行われる．

データフロー・モデルの特徴は 図-1 に示すようにオペランドが揃った命令から実行するという‘データ駆動’にある．このようなデータフロー・モデルはデータフローに添ってコントロールが分岐するマルチ・コントロールフロー・モデル[9]とみなすこともできるが，むしろ，データフローに添って実行すればコントロールフローという概念を用いずに命令レベルの並列計算が実現できることを示した点にデータフロー・モデルの特徴があるといえよう．

データフローグラフでの計算は入力されたトークンのみに依存し，また各命令は副作用を持たないため関数型言語の自然な実現モデルの一つとなっている．

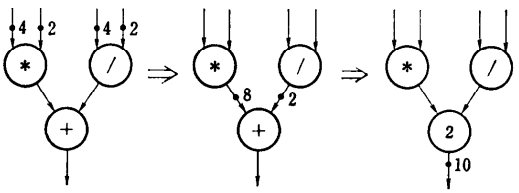


図-1 算術式 4*2+4/2 のデータフロー・モデルによる実行例

* もちろん，実際の処理系では実行順序が定まる場合が多いがそれは言語本来の性質ではない．

** ここでは関数型言語として副作用のないものを仮定している．

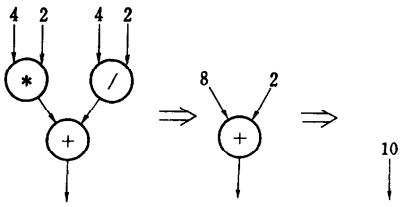


図-2 算術式 4*2+4/2 のリダクション・モデルによる実行例

ただ，データ（トークン）と命令（グラフ）が分離されているため，関数（グラフ）をデータとして扱ったり，関数（グラフ）を値として返す関数は扱いにくい．

（3） リダクション・モデル

関数型言語のもう1つの自然な計算モデルがリダクション・モデルである．リダクション・モデルの特徴は命令とデータを区別せず，プログラム全体を1つの構造として扱う点にある．たとえば，図-1 のデータフロー・モデルによる計算をリダクション・モデルとして実行すると **図-2** のようになる．命令とデータを区別せず，式として同一視することの利点は，関数をデータとして自由に扱える点にある．たとえば，

def $f(x)=g$ where $g(y)=x+y$

という関数 f は入力 x に対し，y を入力として $x+y$ を出力とする関数 g を生成する関数生成関数を表わす．このような関数 f をデータフロー・モデルで直接実現するのは困難であるが，リダクション・モデルでは容易に実現することができる．たとえば，ラムダ式（**4.1.1** で述べる）を用いれば関数適用 $(f(3))(2)$ は次のように計算される．

$(f(3))(2)\rightarrow(\lambda y.3+y)(2)\rightarrow 3+2\rightarrow 5$

ここで，$f(3)$ や $(\lambda y.3+y)$（2）や 3+2 のように書き換え可能な（部分）式は 'リデックス" (REDuciable EXpression) と呼ばれ，そのときに使用された規則（関数呼び出しの関数本体での置き換え，組込命令の実行）は '書き換え規則' と呼ばれる．

2.2 データ駆動，要求駆動とリダクション戦略

次にリダクション・モデルの性質と命令駆動方式との関係をリダクション戦略の観点から述べる．はじめに関数呼び出しの順序と関数の性質の関係について述べておこう．関数型言語では $f(g(x),h(x))$ のように関数がネストしているときに，$g(x)$, $h(x)$ をどの時点で実行するかによって関数 f のふるまいが変わってくる．たとえば，f が第一引数の値が0ならば常に1を返す関数とする．すると $h(y)$ の値がたとえ求まらなくても $g(x)$ の値が0であることさえわかれば f の結果を求めることができる．すなわち，g, h を実行してから f を起動するような関数呼び出しの順序では g, h の両方が止まらないかぎり，f の結果を求めることができないが，f を先に実行して，必要になった時点で g または h を実行するような関数呼び出しの順序にすれば，たとえ h が止まらない場合でも f の結果を求めることができる．このような関数呼び出しの順序の指定は計算規則[6]と呼ばれ，再帰関数や上記の f のような関数（ノンストリクトな関数）の意味を考える上で重要な働きをする[7]．

リダクション・モデルではこの計算規則に相当するものはリダクション戦略と呼ばれ，リデックスが複数個あったときのリダクション順序を表わす．たとえば，g, h のような内側の関数の適用（リデックス）から（並列に）リダクションする戦略は内側（並列）リダクション戦略と呼ばれ，f のような外側の関数の適用（リデックス）から（並列に）リダクションする戦略は外側（並列）リダクション戦略と呼ばれる[11]．リダクション・モデルの性質はリダクション戦略によって規定される．たとえば，内側並列リダクション戦略をとれば引数の並列実行が可能となるし，外側リダクション戦略をとれば関数の遅延評価や無限木などのデータ操作が可能となる．

リダクション戦略とデータ駆動，要求駆動といった命令駆動方式とは以下の関係にある．データ駆動はデータ・フロー・モデルで提案された駆動方式であるが，リダクション・モデルでは内側並列リダクション戦略の実現方式として使用できる．データ駆動を採用したリダクションマシンとしては FFP マシン，ARM がある．リダクション・モデルでデータ駆動を採用したときの利点は関数の並列実行が容易に実現される点にあるが，欠点としては制御構造が実現しにくい点が挙げられる．たとえば，条件式

if $p(x)$ then $f(x)$ else $g(x)$

をデータ駆動でリダクションしようとすると，x の値が求まった時点で $p(x)$ だけでなく $f(x)$ や $g(x)$ のリダクションも開始されてしまう．この問題を解決する方法として ARM では $f(x)$, $g(x)$ の代わりに $f(x)$, $g(x)$ を計算する関数を結果として返し，$p(x)$ の値が求まった時点でいずれか一方の関数を起動する方式を採用している[36]．

一方，要求駆動方式[31]では結果が必要となったリデックスに結果要求を送ることにより，リダクションが開始される．要求駆動を用いると上記の条件式は次の

ようにしてリダクションすることができる.

- 条件式のリダクションには $p(x)$ の結果が必要である.

→$p(x)$ をリダクションする.

- $p(x)$ の結果が真ならば条件式の結果として $f(x)$ の結果が必要である.

→$f(x)$ をリダクションする.

- $p(x)$ の結果が偽ならば条件式の結果として $g(x)$ の結果が必要である.

→$g(x)$ をリダクションする.

要求駆動方式はデータ駆動方式に比べてリダクション順序の制御が容易なためさまざまなリダクション戦略が実現可能である. たとえば, 内側並列リダクション戦略は内側のリデックスにたいして常に結果を要求することにより実現される. しかしながら, 要求駆動ではオペランドの転送のほかに要求の転送が必要なため算術式のリダクションのように内側のリデックスの結果を常に必要とする場合にはかえってオーバヘッドが大きくなる. したがって, 一般的には外側リダクション戦略の実現方式として使用されることが多い (AMPS, SKIM).

そのほかのリダクションマシンにおける命令駆動方式としては RED マシンのようにリダクション戦略を制御する情報を式中に埋め込む方式がある. これなどはある意味で制御駆動と考えることができる. また, ALICE ではすべてのリデックスを任意の順序でリダクションする戦略を採用している[25]が, これなどはリデックスにマッチした書き換え規則が適用されるという意味で 'パターン駆動' と呼ぶことができよう.

2.3 ストリング・リダクション, グラフ・リダクションとリダクション戦略

リダクションマシンの分類基準の1つにストリング・リダクションおよびグラフ・リダクションという用語があるが, 困った問題は, これらの用語が式の表現形式の違いだけでなく, リダクション戦略の違いを表わす用語としても使用されている点である[8]. より正確な議論をするためには式の表現形式とリダクション戦略は明確に区別されなければならない. 本稿ではストリング・リダクション, グラフ・リダクションという言葉を式の表現形式の違いを表わす用語として使用する. すなわち, 構造体および式のネストを表現するさいに, 一次元的に表現された式 (文字列あるいはシンボルの列) をリダクションするのがストリング・リダクションであり, ポインタを使って二次元的に表現された式 (木構造あるいはグラフ) をリダクションするのがグラフ・リダクションである.

従来, ストリング・リダクションとグラフ・リダクションの違いとして問題にされてきたことは, 引数をコピーしたときストリング・リダクションでは同じ引数を何度もリダクションしなければならないが, グラフ・リダクションでは同一の引数のリダクションは一度で済むというものであった. たとえば,

$f(x)\rightarrow x+x$

$g(y)\rightarrow y*y$

という書き換え規則で, 式 $f(g(3))$ を外側リダクション戦略でリダクションすることを考える. このときストリング・リダクションを用いると,

$$f(g(3))\rightarrow g(3)+g(3)\rightarrow 3*3+3*3 \rightarrow 9+9\rightarrow 18$$

となり, $g(3)$ のリダクションを2回繰り返さなければならない. これに対し, グラフ・リダクションでは $g(3)$ をポインタで共有することが可能なので $g(3)$ のリダクションを外側リダクション戦略でも一度で済ますことができる. しかしながら, 式の共有は '式の名前' のようなものを考えればストリング・リダクションでも表現することが可能である[29),30)]. たとえば, 式の名前を@名前で表わせば

$$\begin{aligned} f(g(3))&\rightarrow @X+@X \quad @X=3*3\\ &\rightarrow @X+@X \quad @X=9\\ &\rightarrow 9+9\\ &\rightarrow 18 \end{aligned}$$

逆に, 式がグラフで表現されていても引数を共有しないでコピーするようなリダクションが可能なことは明らかであろう.

すなわち, リダクションしたとき引数をコピーするか共有するかは式の表現形式とは独立な問題であり, リダクション戦略として別に定義されるべきものである. 本稿では, 引数がコピーされるようなリダクション戦略をコピー有りリダクション戦略, 引数が共有されるようなリダクション戦略をコピー無しリダクション戦略と呼ぶ.

3. リダクションマシンのイメージについて

具体的なリダクションマシンの説明をする前に, リダクションマシンのイメージについて述べる. 仮想的なリダクションマシンの概念図を **図-3** に示す. この仮想リダクションマシンはリダクションの対象となる式を格納する格納領域と, 書き換え規則を備えたプロ

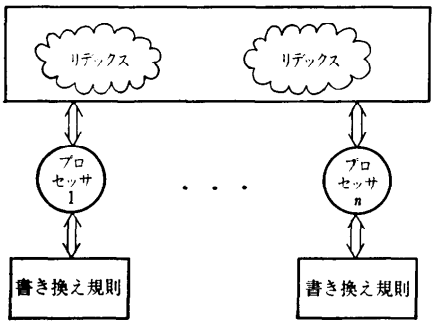


図-3 仮想リダクションマシンのイメージ

セッサから構成される．この仮想リダクションマシンは以下のアルゴリズムに従って動作する．

① 式を走査してリデックスを見つける．

② 採用したリダクション戦略に基づいて実行すべきリデックスを決定し，書き換え規則を適用する．

③ リデックスが見つからない場合には停止して，そのときの式を結果として表示する．

プロセッサが複数個ある場合には，リデックスがないという判断はもう少し複雑になるが，多くのリダクションマシンの基本動作は上記のアルゴリズムに基づいていると言えよう．このアルゴリズムを実際にリダクションマシンとして実現しようとしたときには，以下の点が問題となる．

（1） 目的のリデックスをいかにして高速に決定するか．

（2） 条件式などの制御構造をいかにしてリダクション戦略に反映させるか．

リデックスを検出する方式としては，式全体を走査する方式（RED マシン，FFP マシン）あるいは，リデックスとなる可能性のある命令をキューにつないでおき，適当に取り出して実行可能か否かを調べる方式（ALICE）が基本であるが，リデックスを決定するまでに無駄な処理が入るため高速とは言い難い．リデックスの検出を決定的に行う方式としては，式の最左端が必ずリデックスとなるという結合子の性質を用いた方式（SKIM），および，命令実行に必要なオペランドの個数を数えておいてデータ駆動で起動させる方式（ARM）などがある．また，リダクション戦略も，内側並列リダクションや外側並列リダクションというような単純なものではなく，特別な構成子を使ってリダクション戦略を動的に変更する方式（GMD マシン）や式を関数化させることによりリデックスの発生を抑制する方式（ARM）などが提案されている．4.ではこれらの実現法を中心にリダクションマシンのモデル，言語，アーキテクチャなどについて述べる．

4. リダクションマシンの実現例

これまでに，提案されたリダクションマシンのまとめを表-1に示す．本章では，このうち，実際にハードウェアが作製された Berkling らの RED マシン，Clarke らの SKIM, Darlington らの ALICE を中心に紹介する．

4.1 RED マシン

RED マシン[13)~15)]は，Berkling が設計したラムダ計算に基づくリダクション言語（関数型言語）[12)]を直接実行するために GMD で作製された逐次型文字列

表-1 リダクションマシンの分類

名 称（作製機関）	表現形式	命令駆動方式	リダクション戦略	現 状	参考文献
RED マシン (GMD 大)	文字列	制御駆動	内側，外側混合	ハードウェア稼動中	12~15
FFP マシン (ノースキャロライナ大)	文字列	データ駆動	内側並列	ペパーマシンシミュレーション	27~30
AMPS (ユタ大)	グラフ	要求駆動	外側並列	ペパーマシンシミュレーション	31~34
SKIM (ケンブリッジ大)	グラフ	要求駆動	外側逐次	ハードウェア稼動中	18, 19
ALICE (インペリアル・カレッジ)	グラフ	パターン駆動	内側，外側混合並列	ハードウェア作製中	25, 26
ニューキャッスル・リダクションマシン (ニューキャッスル大)	文字列	データ駆動	内側並列	ペパーマシン	35
ARM (日本電気)	グラフ	データ駆動	内側並列	ペパーマシン（シミュレーション）	36, 37

リダクションマシンである．ハードウェアは Kluge らを中心にして1978年に作製され，800！を 12 秒で計算した（Berkling 談）．RED マシンは7本のスタックと4つの機能ユニットから構成され，文字列（シンボル列）で与えられた式をスタックを用いて走査しながらリダクションが行われる．

4.1.1 ラムダ式とラムダ計算

Berkling のリダクション言語の紹介をする前にラムダ式とラムダ計算について簡単に触れておく．ラムダ式は関数に名前を付けずに関数を式の形で表現する手法の一つであり，ラムダ計算はラムダ式を用いて計算の概念を定義した計算モデルである．ラムダ式を用いるとある変数 X を引数とし E を値とする関数はラムダ記号 (λ) と変数 X を E の前に付加した式 $\lambda X.E$ で表わされる．たとえば，$\lambda X.X+1$ は X を入力引数とし，$X+1$ を値とする関数を表わす．ラムダ式の特徴の一つは，ラムダ記号を式の前に付加することにより，より高階な関数を次々に表現することができる点である．たとえば，$\lambda Y.\lambda X.X+Y$ は Y を入力引数とし，$\lambda X.X+Y$ という関数を値とする関数生成関数を表わす．ここで，注意すべき点は，ラムダ記号で示された入力引数の有効範囲である．ラムダ記号で指定された変数に対して，式中の変数から左に走査して同一名の変数を指定したラムダ記号があればその変数を束縛変数といい，そうでない変数を自由変数と呼ぶ*．たとえば，変数 Y は $\lambda X.X+Y$ においては自由変数であり，$\lambda Y.\lambda X.X+Y$ においては束縛変数となる．

ラムダ式のリダクションはラムダ記号で指定された入力引数に対応する束縛変数を実引数に置き換えることにより実現される．たとえば，

$(\lambda Y.\lambda X.X+Y)\ (2)\ (3)$
$\rightarrow(\lambda X.X+2)\ (3)$
$\rightarrow 3+2$
$\rightarrow 5$

となる．

4.1.2 Berkling のリダクション言語

Berkling の設計したリダクション言語はラムダ式による関数記法を採用しているが，ラムダ計算と本質的に異なる点は，関数と引数のリダクション順序を指定する構成子が式中に明示的に指定されている点である．Berkling は関数適用のために以下の4つの適用構成子を導入している．

（1） $\leftarrow fa$ apply f to a
（2） $\cdot fa$ apply f to reduced a
（3） Δfa apply reduced f to a
（4） $:af$ apply f to reduced a

このようなリダクション順序を指定する適用構成子を導入するとプログラミング言語に表われるさまざまな概念をリダクション言語の中で実現することができるようになる．たとえば，引数を評価してから関数を適用する call by value は(2)または(4)の適用形を使えばよいし，引数を評価しないで関数を適用する call by name は(1)または(3)の適用形を使えばよい．また，条件式 if p then f else g は(3)の適用形を用いて次のように表現することができる．

$\Delta\Delta pfg$

すなわち，上記式では述部 p が先にリダクションされ真偽値（真，偽）が得られる．そして，真偽値が関数として適用されると真，偽がそれぞれ真部 f，偽部 g を選択してリダクションする関数として働く．すなわち，p が真のときは

$\Delta\Delta$真$fg\rightarrow\Delta(Kf)g\rightarrow f$

p が偽のときは

$\Delta\Delta$偽$fg\rightarrow\Delta Ig\rightarrow g$

となる*．

さらに，リダクション言語には，束縛変数の名前の衝突を解決するための反ラムダ記号や，再帰関数を実現するための名前付け記法や，$:af$ 型の適用構成子を用いたオペレータの中間置記法や手続き呼び出しのためのブロック表現，リストを用いた引数マッチングなどの機能が用意されている[12),14)]．

4.1.3 アーキテクチャ

RED マシンのアーキテクチャを図-4に示す．RED マシンの内部ではプログラムはシンボルの列として表現されている．ただし，このシンボルは論理的には2進木構造を表わす．そこで，RED マシンでは3本のスタック (E, M, A) を用いて式を走査するという方式を採用している．リダクションを実現するための各機能ユニットの動作は以下のとおりである．はじめに，スタックEにリダクションの対象となる式（シンボル列）が格納される．次に，TRANS ユニットはスタックMを補助スタックとして使用して，スタックAに式の2進木構造を保ちながら1シンボルずつ転送してゆ

---

* より厳密には，式 M 中の自由変数の集合を $FV(M)$ で表わすと，
$FV(X)=\{X\}$
$FV(\lambda X.M)=FV(M)-\{X\}$
$FV(MN)=FV(M)\cup FV(N)$

* 原論文では '真'，'偽'，'K'，'I' の代わりに別な記号が用いられているが，直観的な理解を助けるために，K，I という結合子 (4.2.1) の記号を用いた．

く．このとき REDREC (REDex RECognition)ユニットはリデックスがあるかどうかの検査を行い，リデックスを検出すると REDEX (REDuction EXecution) ユニットおよび ARITHmetic ユニットに制御を渡して，それぞれ，式の書き換え，数値演算および論理演算を行う．そして，すべてのシンボルがスタックAに移ると，今度はスタックAからスタックEに向けてシンボルの転送を行い，同様にしてリデックスの検出，書き換えを行う．この動作をあらかじめ指定された回数またはリデックスが検出されなくなるまで繰り返し，そのときにスタックEまたはスタックAに残されたシンボルを結果として表示する．

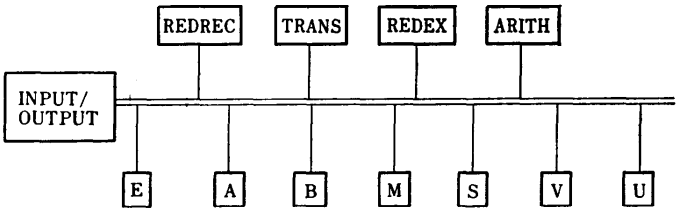


図-4 GMD リダクションマシン 参考文献8)より転載

4.1.4 RED マシンに対するコメント

RED マシンのアプローチは逐次処理が基本であり，リデックス検出のコストも非常に高い．したがって，マシンの性能としては高速とは言い難いが，リダクションに基づく計算機が実際に構築できることを示した功績は大きいと言えよう．

4.2 SKIM

SKIM[18]は結合子を利用した逐次型グラフ・リダクションマシンであり，Clarke らにより，1980年にケンブリッジ大で作製された．SKIM のユーザ言語は SMALL と呼ばれる数式処理用に設計された関数型言語であり，SMALL で書かれたプログラムを結合子を利用した関数表現のひとつである Turner グラフ[16]にコンパイルして実行する．多くの関数型言語はラムダ式および結合子式（Turner グラフ）で表現できるので，SKIM はこの意味で汎用の関数型言語マシンと言えよう．SKIM の特徴はモデルとなった Turner のグラフ・リダクション・モデルに依存している．すなわち，結合子式をコピー無し外側リダクション戦略でリダクションすることにより，効率のよい引数の遅延評価を実現している点にある．Turner のグラフ・リダクション・モデルとの相違点としては，次に実行する結合子の引数をスタックに積む代わりに逆転ポインタを用いて引数のアクセスを可能にした点にある．

4.2.1 結合子と結合子計算

SKIM および Turner のグラフ・リダクション・モデルのアプローチの特徴は関数の表現方法に結合子式を用いた点にある．この結合子は Curry により発展させられた関数表現の一つであり，S, K, I といった記号の組み合わせだけで関数を表現しようとするものである．(SKIM の名前の由来もこの結合子の名前からきている)．結合子 S, K, I は以下のリダクション規則を持つ．

$$Sfgx \rightarrow fx(gx)$$
$$Kxy \rightarrow x$$
$$Ix \rightarrow x$$

結合子式の表現能力はラムダ式と同等であり，任意のラムダ式は結合子式に変換できることが知られている．この，結合子式による関数表現の特徴は以下のとおりである．

（1） 変数を使わずに関数を表現できる．

（2） リデックスの検出が容易である．

変数を使わない関数表現というのは奇異に感じるかもしれないが，たとえば，同一値を返す関数 $\lambda X.X$ は結合子式では SKK という3つの結合子の適用形で表現できる．実際，

$$SKKx$$
$$\rightarrow Kx(Kx)$$
$$\rightarrow x$$

となり，SKK が $\lambda X.X$ と同じ働きをすることが確かめられる．関数の表現に変数がないということは極めて重要である．すなわち，ラムダ式のリダクションでは変数を実引数に置き換えるときに実引数中の自由変数が束縛されないようにするための判断が必要であるが，結合子のリダクションではこのような判断が不要になることを示している．たとえば，以下のラムダ式のリダクションを考えてみよう．

$$(\lambda X.\lambda Y.X+Y)\ (Y+1)$$

ここで実引数 $Y+1$ に現われる変数 Y は自由変数である．したがって，上記のラムダ式で変数 X をそのまま $Y+1$ で置き換えてしまうと

$$\lambda Y.(Y+1)+Y$$

となり，実引数中にあった自由変数 Y が束縛変数に

なり，関数の意味が変わってしまう．これを避けるためにラムダ計算では変数の置換において，自由変数が束縛変数になる恐れがある場合には名前が衝突した束縛変数の名前を変更するという方式を用いている．この方法にしたがって束縛変数の名前を X から Z に変更すれば，上記式は次のように正しくリダクションされる．

$\lambda Z.(Y+1)+Z$

しかしながら，このような変数の名前の書き換えを実行時に行うことは非効率であるし，ハードウェア的に実現しにくいのは明らかである．一方，結合子式では関数表現にもともと変数がないため，このような問題は生じない．したがって，結合子のリダクションは単純な式の書き換えで実現することができる．

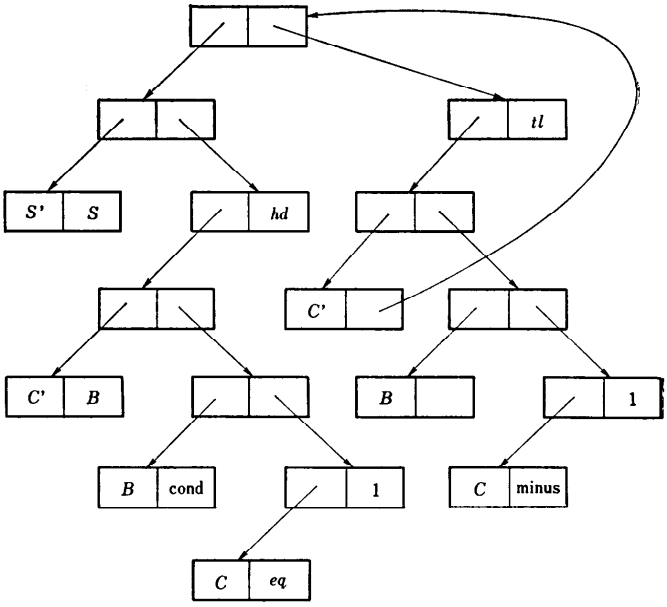


図-5 関数 select を表わす Turner グラフ 参考文献42)より転載

結合子式のもう一つの特徴は，最左端の結合子が必ずリダクション可能な結合子になっているということである．したがって，結合子式を用いた場合にはリデックスを見つけるために式を走査する必要はなく，最左端の結合子からリダクションすればよい．さらに，最左端の結合子からリダクションした場合には外側リダクション戦略となり，再帰呼び出しや条件式があっても（その関数が止まる場合には）リダクションが停止することが保証されている[3]．

4.2.2 Turner グラフ

前節で述べたようにラムダ式に比べて，結合子式はリダクションモデルとしてより有用な性質を備えている．しかしながら，Curry らの提案した結合子計算は原理的な計算能力を示したものであり，現実の計算機のモデルとしては以下の問題点を持つ．

（1） 変換された結合子式が元のラムダ式に比べて非常に大きくなってしまう．

（2） 最左端の結合子からリダクションしてゆくと与えられた引数をそのままコピーすることになり，引数中のリデックスを何回もリダクションしなければならなくなる．

このような問題点を解決し，より効率的な結合子のリダクションを実現したのが Turner のグラフ・リダクション・モデル[16]である．Turner はまず，変換後の結合子式をよりコンパクトなものにするために従来から知られていた S, K, I, B, C を用いた変換アルゴリズムにさらに S', C' を加えた変換アルゴリズムを考案した[17]．このアルゴリズムを用いると，元のラムダ式と同程度の大きさの結合子式に変換される．たとえば，リスト s の n 番目の要素を選ぶ select という関数は以下の結合子式に変換される．

ラムダ式

select $=\lambda n.\ \lambda s.$ if $n=1$ then hd s
else select $(n-1)$ $(tl\ s)$

結合子式

select $=S'S$
$(C'B(B$ cond $(C$ eq $1))$ hd$)$
$(C'B(B$ select $(C$ minus $1))$ $tl)$

ただし，hd, tail, cond はそれぞれリストの先頭，先頭を除いた残りのリスト，条件式を実現する関数であり，B, C および S', C' は以下のリダクション規則を持つ結合子である．

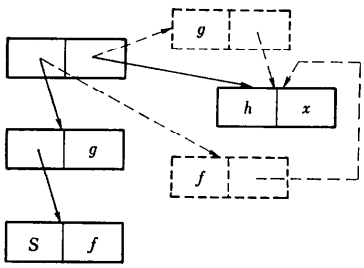


図-6 $Sfg(hx)$ のリダクション

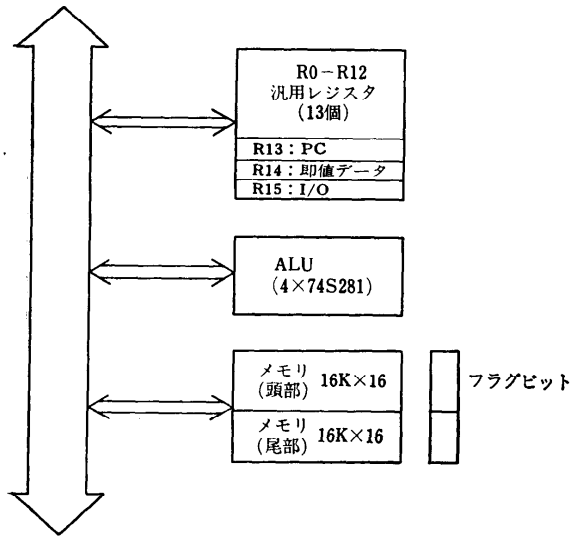


図-7 SKIM システム構成 参考文献18)より転載

$Bfgx \rightarrow f(gx)$
$Cfgx \rightarrow (fx)g$
$S'kfgx \rightarrow k(fx)(gx)$
$C'kfgx \rightarrow k(fx)g$

なぜ，S' や C' を加えると結合子式がコンパクトになるかについてはここでは詳しく述べないが，引数が2個以上になったときに，S' および C' があると式の関数適用の構造を変えずに結合子式に変換できる点が挙げられる．また，結合子式 select の Turner グラフを図-5に示す．Turner グラフでは各セルは関数適用を表わしている．そして，グラフの左側のポインタを辿っていった先の結合子（あるいは関数）が次にリダクション可能な結合子（関数）を表わしている．左端の結合子はその結合子のリダクションに必要な引数が揃うとグラフの書き換えを行う．

Turner グラフのもうひとつの特徴は結合子式を表現するのにグラフ構造を用いて，コピー無しリダクションを実現した点である．$Sfg(hx)$ をリダクションしたときのグラフの変化を図-6に示す．結果は $f(hx)(g(hx))$ であるが (hx) がポインタで共有されるため (hx) のリダクションは一度で済むようになっている．

4.2.3 アーキテクチャ

SKIM のシステム構成を図-7に示す．SKIM のアーキテクチャは結合子のリダクション規則がマイクロプログラムで定義されていることを除くと汎用計算機と特に変わった点はない．プロセッサは16個の内部レジスタを持ち，13個が汎用レジスタに，残りが，マイクロプログラムのアドレスレジスタ，即値および分岐アドレスの格納レジスタ，入出力制御に使用されている．また，メモリは16ビット幅で32K語あり，セルの頭部と尾部の2バンクに分かれている．マイクロ命令は32ビット幅で4K語あり，600 ns で動作する．性能的には IBM 370/165 上の LISP インタプリタの約半分（コンパイラの8分の1）の速度で動作するという．また，最近の報告[19]によるとマイクロコードを書き直した SKIM 11 では 50% の性能向上がみられたという．

4.2.4 SKIM に対するコメント

SKIM のアーキテクチャについては特に言うべき点はないが，結合子を機械語として持つ最初の計算機として評価されよう．また，Norman によると結合子のリダクションの研究に SKIM を実際に使用しているとのことである．

ここでは，結合子を用いたリダクションマシンのアプローチ全体についてコメントしておく．結合子はラムダ式に比べて書き換え規則が単純であり，いくつかの結合子を追加することにより，比較的コンパクトに関数を表現することができる．さらに，結合子式の表現能力の高さを考慮すると，リダクションマシンのアプローチとしては非常に魅力的である．Turner や SKIM のアプローチをさらに発展させたものとしてラムダ式をより一般化された結合子 (super combinator) に展開する Hughes の研究[20]や結合子列をスタック命令にコンパイルする Jones の研究[21]がある．これらは確かに，より高速な逐次型外側リダクションを実現するためには有効であろうがアーキテクチャ的にはますます汎用計算機（ノイマン型計算機）に近づいてゆくように思える．また，単に，逐次型の外側リダクションを実現するだけならば，結合子を使ったリダクション方式よりも，クロージャを使ったリダクション方式の方が速いという報告[22]もある．

また，結合子を用いた並列リダクションマシンの研究としては Sleep および Burton による ZAPP[23),24)]の研究がある．ZAPP は巡回的に結合したコンピュータ・ネットワークのなかで Turner グラフを並列リダクションしようというものであるが，結合子を単純に並列リダクションしたときに生ずる無限ループの発生をどう防ぐかなど，未解決な問題点が多い．

4.3 ALICE

ALICE (Applicative Language Idealized Com-

パケット識別名	関数ポインタ	引数リストまたは値	パケットの状態	参照カウンタ	結果通知先リスト

図-8 ALICE のパケット表現

puting Engine)[25] は汎用的な並列処理を目的とした並列グラフ・リダクションマシンである．ALICE の主言語は HOPE と呼ばれる項書き換えシステムを基礎とした関数型言語であるが，そのほかにも OR 並列 PROLOG や PARLOG, Portable Standard Lisp などの言語もサポートしている．これらの言語は CTL (Compiler Target Language)[26]と呼ばれる ALICE の機械語に翻訳され，さらに内部表現としてはパケットと呼ばれるデータの集まりに変換される．パケットは図-8 に示すように命令部とデータ部を持ち，プログラムやデータはこのパケットをポインタで結合したグラフで表現される．そして，このパケットを書き換えることによりリダクションが実現される．

ALICE は INMOS の Transputer を使ってプロトタイプ機が 1985 年の 6 月に稼働する予定であるが，Transputer の供給が遅れているため完成が遅れているとのことである．また，マンチェスタ大で開発したデータ・フローマシンをベースにした ALICE II の開発がアルベイ計画の下で進められている．

4.3.1 HOPE

HOPE は項書き換えシステムを基礎にした関数型言語である．HOPE によれば階乗プログラムは以下のように表わされる．

```
Factorial(N) <= factB(O, N)
factB(i, i) <= i
factB(i, i+1) <= i+1
factB(i, j) <=
        factB(i, mid)*factB(mid, j)
        where mid=(i+j)/2
```

プログラムの意味はほとんど自明と思われるが，左辺の呼び出しがあると，右辺の式に変換されることを示している．たとえば，4 の階乗は以下のようにして計算される．

```
  Factorial(4)
→factB(0, 4)
→factB(0, 2)*factB(2, 4)
→(factB(0, 1)*factB(1, 2))
  *(factB(2, 3)*factB(3, 4))
→(1*2)*(3*4)
→2*12
→24
```

ここで注意すべき点は，項書き換えシステムでは，変数の結合は書き換えの過程の中で自明の手続きとして与えられていることである．結合子の役割が変数の分配にあることを考えると，項書き換えシステムのリダクションは非常に高度なリダクションを実現していることになる．また，リデックスの位置が固定的でないため，リデックスの決定には式の走査が必要である．

4.3.2 アーキテクチャ

ALICE のマシン構成を図-9 に示す．マシンはパケットを格納するパケット・プール・セグメント (PPS) およびパケットの書き換えを行う実行ユニット (Ajent) とこれらを結ぶデータ転送用の結合ネットワーク (Interconnection Network) とパケットの識別番号などを管理するための分配ネットワーク (Distribution Network) から構成される．各ポートにはどのユニットを接続してもよく，また，パケット・プール・セグメントと実行ユニットの個数の組み合わせ方も自由である．結合ネットワークは 64 のポートを持つ 100 Mbit の転送速度を持つ 3 段の 4*4 のクロスバ・スイッチで構成されている．また，分配ネットワークはリングバスで実現されている．

パケット・プール・セグメントはパケット管理ユニット (PMU)，ロード・シェアリング・ユニット (LSU)，外部要求ユニット (ERU) の 3 つのユニット

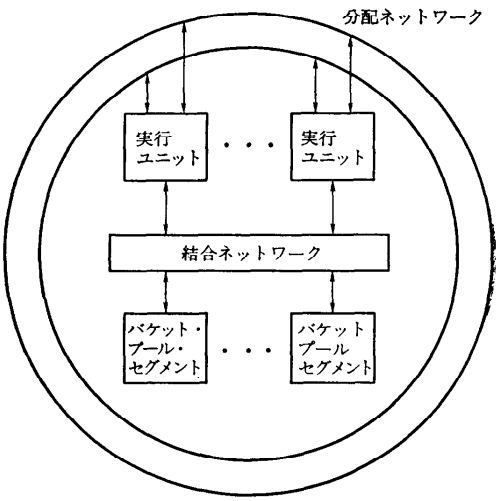


図-9 ALICE マシン構成 参考文献 26)より転載

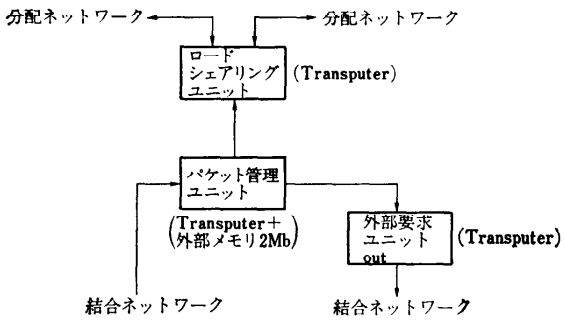


図-10 パケット・プール・セグメントの構成

から構成され (図-10), それぞれ独立の Transputer で実現されている. PMU は, 2M バイトの外部メモリを持ち, 生成されたパケットを格納する. 外部メモリは固有のアドレスを持ち, システム全体で1つの線形空間を構成している. LSU はメモリの未使用領域を管理しており, パケット生成時にメモリ領域を確保して識別番号を与えるために使用される. ERU はパケットを結合ネットワークに転送するときのバッファとして使用される.

実行ユニットは LSU, 入力用 ERU, 出力用 ERU および2つのパケット書き換えユニット (PRU) と関数定義キャッシュ・マネージャ (FDCM) から構成され (図-11), それぞれ独立の Transputer で実現されている. PRU はそれぞれ 64K バイトのメモリを持ち, そこに, 書き換え規則を格納している. そして,

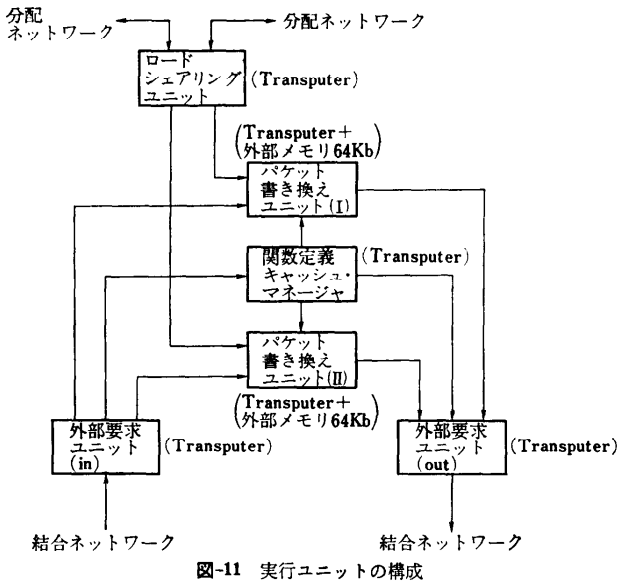


図-11 実行ユニットの構成

入力用 ERU から入力されたパケットを受け取ると FDCM に格納されている関数定義および局所メモリ中の書き換え規則を用いてリダクションし, 結果を出力用 ERU に転送する. 新しいパケットを生成するときは, LSU を使ってパケットの格納領域と識別番号の確保を行う.

4.3.3 ALICE に対するコメント

ALICE はまだ稼働していないが, 詳細なシミュレーションから彼らは, ALICE のアプローチに対するいくつかの教訓を得ている.

(1) 多くの並列性を望んではいけない. (処理単位が小さいとバッファ管理などに必要な通信のオーバヘッドが大きくなりすぎる).

(2) 遅いプロセッサを多数つなげて高速なマシンを作製するのは難しい. (単体でも実行したときにも高速なプロセッサを用いないと性能はでない).

(3) (不用意に並列実行するよりも) コンパイラによって性能改善できる点が多い. 上記の教訓は並列マシンを作製するときに常にいわれることであるが, リダクションマシンも例外ではないことを示唆している.

4.4 そのほかのリダクションマシン

そのほかのリダクションマシンとしては,

(1) ノースキャロライナ大 (Mago') の FFP マシン

(2) ユタ大 (Keller) の AMPS

(3) ニューキャッスル大 (Treleaven) のストリング・リダクションマシン

(4) 日本電気 (筆者) の ARM

がある.

Mago' の FFP マシン[27),28)]は Backus の提案した無変数型の関数型言語 FP を直接実行する並列ストリング・リダクションマシン (図-12) である. このマシンの特徴は木状に結合された多数のセルの最下層にプログラムの断片(関数名, アトム, デリミタなど)を一つずつ格納し, 各セルの内容を近隣のセルに移動させることによりリダクションを実現している点にある. リデックスの検出や移動するデータの指示は木構造のネットワークを用いた巧妙なアルゴリズムにより実現されている. 木構造アーキテクチャは VLSI 化にも適しており, 実マシンは作成されていないが注目すべきアーキテクチャの一つである.

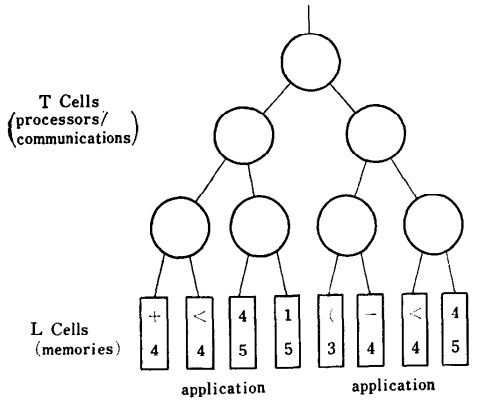


図-12 FFP マシン 参考文献8)より転載

ユタ大の AMPS[31]はデータ・フロー・グラフを要求駆動でリダクションする並列グラフ・リダクションマシンとみなすことができる．AMPS 自身は木構造のネットワークを持つ並列マシン構想が発表されただけでその後特に具体化されていない．また，対象言語は初期の FGL[32]から等式系に基づいた FEL (Function-Equation Language)[33]に移っている．実行モデルとしては要求駆動で動作する機械語シミュレータが PASCAL で作成されており，最適化の研究[34]などが行われているが，ハードウェア化の話は当面ない様子である．

ニューキャッスル・リダクションマシン[35]は一列に並べられたプロセッサを両方からアクセス可能なキューでつなぎ，プロセッサ間で一文字単位の転送を行いながら式の書き換えを行う並列ストリング・リダクションマシンである．各プロセッサは現在の状態と隣のプロセッサから送られてきた文字によって次の状態に遷移し，次のプロセッサに転送する文字を決定する．しかしながら，このような形でリダクションできる関数型言語は限られており，方式自体にも多くの問題点を抱えている．

ARM[36]は筆者らによって検討が進められた LISP プログラムの並列実行を狙った並列グラフリダクションマシンである．ARM の特徴は組込命令をノードとした木構造グラフに LISP プログラムを展開し，グラフの各ノードにリデュースオペレータと呼ばれる変数結合用のオペレータを付加した点にある．リデュースオペレータのリダクションは結合子のリダクションと同様に実引数の分配先を規定している．また，各ノードの組込命令は単純なデータ駆動ではなくリデュース・オペレータにより命令実行を制御することができる．これにより，基本的な駆動メカニズムとしてはデータ駆動を用いながらも，遅延評価や関数引数などの実現が可能となっている．また，アーキテクチャとしては処理単位を大きくするためにグラフを機能単位にブロック化する方法や，より効率的に実引数を分配するリデュース・オペレータのコンパイル方法やリダクションマシン向きのリストの表現方法や，ガーベージ・コレクション方式の検討が行われている[37]．

5. 終わりに

実際にハードウェアが試作された RED マシン，ケンブリッジ大の SKIM，インペリアル・カレッジの ALICE を中心にこれまでに発表されたリダクションマシンのアプローチ，言語，アーキテクチャについて述べた．関数型言語の実行モデルを考えたとき，式の書き換えによって計算を実現するというリダクションマシンの考え方は確かに魅力的である．しかしながら，リダクションマシンが本当に認められるためには単なるアーキテクチャ的なおもしろさだけではなく，その有効性を実証できるものでなければならない．

リダクションマシンのアプローチはいろいろあるが，大きくわけて，逐次型マシンと，並列型マシンに分けられる．逐次型マシンの場合，外側リダクションをいかに実現するかが焦点となっているようであるが，外側リダクションであろうと内側リダクションであろうと式の書き換えで計算を実現するよりは現在のハードウェア技術で実現が容易なスタック命令などにコンパイルする方式のほうが高速な処理系が実現できるように思える．一方，並列型マシンのアプローチは，さらに ALICE のようなグラフ・リダクション系と Mago' の FFP マシンのようなストリング・リダクション系に分けられる．ALICE のようなグラフ・リダクションマシンの場合，ポインタが多用されているためどうしてもメモリ・アクセス・ネックになりやすい．したがって，高並列マシンよりも比較的高速なプロセッサを数台あるいは数十台つなげた中規模な並列マシンが適していると思われる．一方，FFP マシンのようなストリング・リダクションマシンは現在のハードウェア技術で実現するにはまだ困難が多い．しかしながら，Backus が指摘するように，真のボトルネックがメモリアクセス・ネックにあるとすればグラフ・リダクションマシンではノイマン型計算機を超えることはできないかも知れない．メモリ共有が無いという意味

でノイマン型計算機を凌駕する可能性があるとすればむしろ FFP マシンのようなセルラ・アーキテクチャではないだろうか．1つの問題を解くために多くの断片にわけ，各断片が局所的な情報を基に独立に動作するという発想は傾聴すべき価値がある．

謝辞 本稿を作成するにあたって有用なコメントを頂いた ICOT の上田和紀氏および田中二次郎氏に感謝の意を表します．

参考文献

1) Backus, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, CACM Vol. 21, No. 8, pp. 613-641 (Aug. 1878).
2) Barendregt, H. P.: The Lambda Calculus: Its Syntax and Semantics, North-Holland (1981).
3) Curry, H. B. and Feys, R.: Combinatory Logic, Vol. 1, North Holland (1958).
4) Dennis, J. B.: First Version of a Data Flow Procedure Language, Lecture Notesin Computer Science, Vol. 5, pp. 187-216 (1972).
5) Henderson, P.: Functional Programming Application and Implementation, Prentice-Hall International Series in Computer Science, London (1980).
6) Manna, Z.: Mathematical Theory of Computation, Mc-Graw Hill (1974).
7) Stoy, J.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT press (1977).
8) Treleaven, P. C., Brownbridge, D. R. and Hopkins, R. P.: Data-driven and Demand-driven Computer Architecture, Computing Surveys, Vol. 14, No. 1, pp. 93-143 (Mar. 1982).
9) Treleaven, P. C., Hopkins, R. P. and Rautenbach, P. W.: Combining Data Flow and Control Flow Computing, Computer Journal Vol. 25, No. 2, pp. 207-217 (1982).
10) 飯塚：VLSI コンピュータアーキテクチャ，電子通信学会誌，Vol. 68 No. 2, pp. 180-186 (1985).
11) 二木，外山：項書き換え型計算モデルとその応用，情報処理学会誌，Vol. 24, No. 2, pp. 133-146 (1983).
12) Berkling, K. J.: Reduction Languages for Reduction Machines, Proc. Second Int. Symp. Computer Architecture, pp. 133-140 (1975).
13) Kluge, W.: The Architecture of a Reduction Language Machine Hardware Model, Geselschaft fur Mathematik und Datenverarbeitung MBH Bonn, Teck. Report ISF-Report 79 (Aug. 1979).
14) Hommes, F. and Schlutter, H.: Reduction Machine System User's Guide, Gesellschaft fur Mathematik und Datenverarbeitung MBH Bonn, Tech. Report ISF-Report 79 (Dec. 1979).
15) Kluge, W. and Schlutter, H.: An Architecture for Direct Execution of Reduction Language, Proc. Int. Workshop on HLL Comp. Archi., pp. 174-180 (June 1980).
16) Turner, D. A.: A New Implementation Technique for Applicative Languages, Software-practice and experience, Vol. 9, No. 1, pp. 31-49 (1979).
17) Turner, D.A.: Another Algorithm for Bracket Abstraction, Journ. of Symbolic Logic, Vol. 44, No. 2, pp. 267-270 (June 1979).
18) Clarke, T. J. W., Gladstone, P. J. S., Maclean, C. D. and Norman, A. C.: SKIM-The S, K, I Reduction Machine, Proc. 1980 LISP Conf. pp. 128-135 (1980).
19) Stoye, W. R., Clarke, T. J. W. and Norman A. C.: Some Practical Methods for Rapid Combinator Reduction, Proc. ACM Symp. LISP and Functional Programming, pp. 159-166 (Aug. 1984).
20) Hughes, R. J. M.: Super Combinators: A New Implementation Method for Applicative Languages, Proc. ACM Symp. LISP and Functional Programming, pp. 1-10 (Aug. 1982).
21) Muchnick, S. S. and Jones, N. D.: A Fixed-Program Machine for Combinator Expression Evaluation, Proc. ACM Symp. LISP and Functional Programming, pp. 11-20 (Aug. 1982).
22) Ida, T. and Konagaya, A.: Comparison of Closure Reduction and Combinarory Reduction Schemes, 京都数理解析研究所ソフトウェア科学，工学における数理的方法研究集会 (1984).
23) Sleep, M. R.: Applicative Language, Dataflow and Pure Combinatory Code, Proc. COMPCON 80 spring, pp. 112-115 (Feb. 1980).
24) Burton, F. W. and Sleep, M. R.: Execution Functional Programs on a Virtual Tree of Processors, Proc. Functional Programming Languages and Computer Architecture, pp. 180-187 (Oct. 1981).
25) Darlington, J. and Reeve, M.: ALICE: A Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages, Proc. Functional Programming Languages and Computer Architecture, pp. 65-75 (Oct. 1981).
26) Reeve, M.: An Introduction to the ALICE Compiler Target Language, Imperial College Research Report(1981).
27) Mago', G. A.: A Network of Microprocessors to Execute Reduction Languages, JCIS Vol. 8,

No. 5 and Vol. 8, No. 6 (1979).
28) Mago', G. A.: A Cellular Computer Architecture for Functional Programming, Proc. IEEE COMPCON 80, pp. 179-187 (Feb. 1980).
29) Mago', G. A.: Copying Operands versus Copying Results: A Solution to the Problem of Large Operands in FFP's, Proc. Functional Programming Language and Computer Architecture, pp. 93-98 (Oct. 1981).
30) Mago', G. A.: Data Sharing in an FFP Machine, Proc. 1982 ACM Symp LISP and Functional Programming, pp. 201-207 (Aug. 1982).
31) Keller, R. M., Lindstrom, G. and Patil, S.: A Loosely-coupled Applicative multi-processing System, Proc. AFIPS Conference, pp. 861-870 (June 1979).
32) Keller, R. M., Jayaraman, B., Rose, D. and Lindstrom, G.: FGL (functional graph language) Programmers' Guide, AMPS Technical Memorandum No. 1, Univ. Utah (July 1980).
33) Keller, R. M.: FEL: An Experimental Applicative Language, 情報処理ソフトウェア基礎研究会資料, pp. 73-79 (Dec. 1982).
34) Tanaka, J.: Optimized Concurrent Execution of an Applicative Language, Ph. D Thesis University of Utah (Mar. 1984).
35) Treleaven, P. C. and Mole, G. F.: A Multiprocessor Reduction Machine for User-Defined Reduction Languages, Proc. 7th Int. Symp. on Computer Architecture, pp. 121-130 (May 1980).
36) 小長谷, 山本, 北森, 内藤: リダクションマシン "ARM" の基本アーキテクチャ, 信学技報 EC 82-28, pp. 69-78 (June 1982).
37) 北森, 内藤, 小長谷, 山本: リダクションマシン ARM におけるリスト処理とガーベジコレクション, 信学技報 EC 82-29, pp. 79-88 (June 1982).
38) 尾内, 麻生, 清水, 益田, 松本: 並列推論マシン PIM-R のアーキテクチャ, 情報処理第 30 回全国大会論文集, pp. 195-196 (Mar. 1985).
39) 柴山, 村山, 富田, 萩原: 並列リダクション・モデルに基づく Prolog マシン(1)-アーキテクチャ, 情報処理第 28 回全国大会論文集, pp. 187-188 (Mar. 1984).
40) Vegdahl, S. R.: A Survey of Proposed Architectures for the Execution of Functional Languages, IEEE Trans. on Computers, Vol. c-33, No. 12, pp. 1050-1071 (Dec. 1984).
41) Davis, A. L. and Keller, R. M.: Data Flow Program Graphs, IEEE Computer, Vol. 15 No. 2, pp. 28-41 (Feb. 1982).
42) 井田: 新世代プログラミング 13 アーキテクチャ(3)リダクション・マシン, bit Vol. 16, No. 9, pp. 60-65 (1984).
43) Landin, P. J.: The Mechanical Evaluation of Expression, Computer J., Vol. 6, No. 4, pp. 308-320 (1964).

(昭和 60 年 4 月 26 日受付)