

A new head-normalization algorithm for λ -calculus

野口憲一 (Ken-ichi Noguchi)

東京工業大学 理学部 情報科学科

本稿で述べるLSアルゴリズムは、主に次の2つを行うアルゴリズムである。(1) λ 式のhead-normal-formの概形を高速に求める。(2) λ 式のhead-needed-redexをもれなく見つける。ここで言うhead-normal-formの概形とは、 $\lambda x_1 x_2 \dots x_n. y \square_1 \square_2 \dots \square_m$ 、つまりhead-variableに続く部分式の中身は明示せず、その個数だけを示した形である。またhead-needed-redexとは、head-normal-formへ到達するために、いつかは必ずreductionしなければならないredexである。ここでは、LSアルゴリズムの考え方、その性質、及び実行効率の計算機実験による評価について述べる。

A new head-normalization algorithm for λ -calculus

Ken-ichi Noguchi

Department of Information Science

Tokyo Institute of Technology

Ookayama, Meguro-Ku, Tokyo 152, Japan

In this paper, we present an algorithm named LS-algorithm, which do the following two things:(1)it finds the outline of the head-normal-form of the given λ -term quickly. (2)it finds all the head-needed-redexes of the given λ -term. By the outline of the head-normal-form we mean $\lambda x_1 \dots x_n. y \square_1 \dots \square_m$, the significant half up to the head-variable and the number of succeeding subterms. The head-needed-redex is a redex which is necessarily contracted in any reduction to its head-normal-form. We describe the idea of LS-algorithm and its properties, and evaluate its computational efficiency by computer experiment.

1. はじめに

λ -calculusは、関数の概念を抽象化した一種の論理体系である。LISPなどのいわゆる関数型言語のプログラムは、そこで使われる関数を入式の定数と見なせば一種の入式と考えることができる。従って、入式をリダクションするということは、関数型言語の世界では、プログラムの評価、実行を行なうことに対応する。

本稿は、この λ -calculusのリダクションに関する研究である。ここでは、 λ -calculusの β リダクションを、通常とは異なる手法で実行するアルゴリズムのアイデアを示し、その性質を述べ、さらにこのアルゴリズムの計算機実験による、実行効率の評価に関して述べる。

ここで述べるアルゴリズムは、LS (Left-Slope) アルゴリズムと言う。このLSアルゴリズムは、与えられた入式Mに対して、主に次の2つを行なうものである。

1. Mのhead-normal-formの概形を高速に求める。(図1. 1 参照)
2. Mのhead-needed-redexを高速にもれなく見つける。

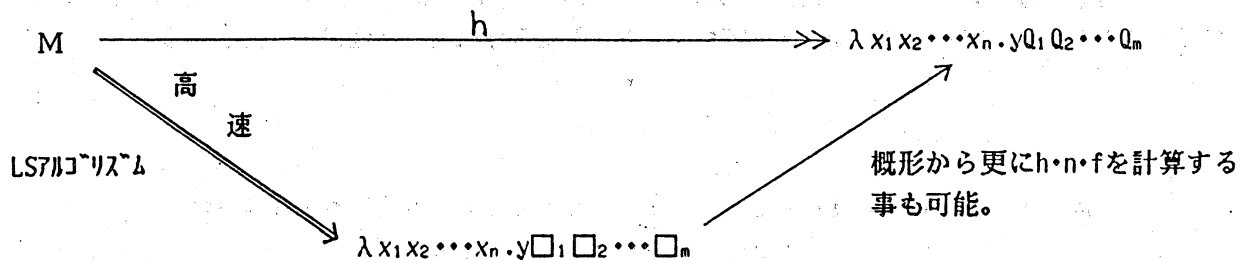


図1. 1 LSアルゴリズムのあらまし

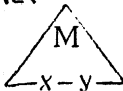
head-normal-form の概形とは head-normal-form $\lambda x_1 x_2 \dots x_n . y Q_1 Q_2 \dots Q_m$ に対し $\lambda x_1 x_2 \dots x_n . y \square_1 \square_2 \dots \square_m$ つまり、head-variableに続く入部分式 $Q_1, Q_2, \dots, Q_m$ に関しては、その中身を明示せず、その個数のみを示すものである。LSアルゴリズムは、より正確に言うと、この概形とこれらの部分式 $Q_1, Q_2, \dots, Q_m$ を求めるための情報を計算する。従って、LSアルゴリズムは、head-normal-formの概形を高速に求めることをまず先に行なうhead-normalizationアルゴリズムということになる。概形が高速に求まることによって、例えばある入式がhead-normal-formを持つか否かを知りたいとき、もし持つ場合には、head-normal-formを持つという情報を早く知ることが出来、また持たない場合でもLSアルゴリズムの実行過程から、リダクションに関する有用な情報や予想が読み取れる場合が多い(例えば、リダクションに関する周期性など)。従って、理論的にはhead-normal-formを持つか否かという問題は決定不能であるが、現実的な側面を考えると、ある入式がhead-normal-formを持つか否かをチェックするための、有効なアルゴリズムであるといえる。

head-needed-redexとは、簡単に言うと、head-normal-formに到達するために、いつかは必ずリダクションしなければならないredexである。関数型プログラミングの世界では、strictness-analysisにおけるstrictな引数に対応するものである(文献[1])。このhead-needed-redexをいくつか、高速に見つけることにより、最左最外戦略と異なる、効率の良い正規化戦略が実現可能となる。

2. λ グラフ

λ 式のλ グラフ表現は、LS アルゴリズムの説明に必要不可欠なものである。λ グラフは、λ 式の abstraction を unary-operator (abstraction-node)、application を binary-operator (application-node) として、木で表現したものである。(図2. 1 参照)

λ グラフに関して、ここでは次の記法を用いる。
最左変数・・・λ グラフの最左斜面の一番下にある変数 (λ 式の最左にある変数と同じ)。

特に、
・・・λ 式 M のλ グラフを表す。
特に、
・・・変数が明示してある場合、その変数 x, y は  において自由に現れている変数とする。

$M \equiv \lambda xy. (\lambda z. zz)x((\lambda y. y)y)$ の場合

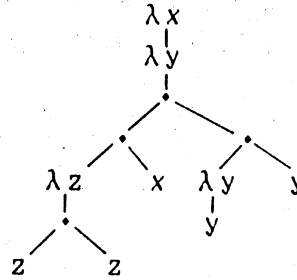


図2. 1 λ グラフの例

λ グラフにおいて head-redex、head-normal-form は次のように表される。

◦ head-redex

$\lambda x_1 x_2 \dots x_n. ((\lambda y. P)Q)Q_1 Q_2 \dots Q_m$

λ グラフでは、最左斜面上にある redex で一番上方にある redex である。(図2. 2)

◦ head-normal-form

$\lambda x_1 x_2 \dots x_n. y Q_1 Q_2 \dots Q_m$

最左斜面上に redex の無い λ グラフとなる (図2. 3)。

また、

head-normal-form の概形 $\lambda x_1 x_2 \dots x_n. y \square_1 \square_2 \dots \square_m$

は、最左斜面の node の列で表わされる。(図2. 3 の枠で囲まれた部分)

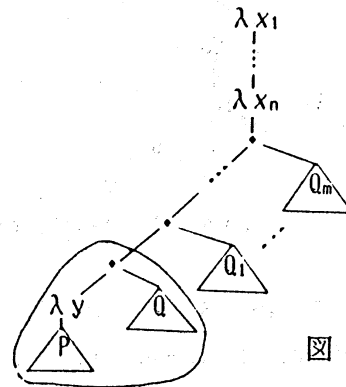


図2. 2

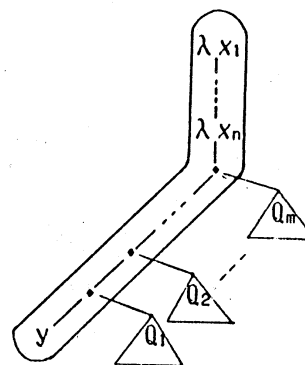


図2. 3

3. Left-Slope

定義 (λ 式の left-slope)

任意の λ 式 M に対して、M の left-slope は次の i)、ii) より定義される。(図3. 1 参照)

i) M の λ グラフの最左斜面 (を構成する node の列) は、M の left-slope である。

ii) M の λ グラフの各 application-node の、右側の部分木の最左斜面 (を構成する node の列) は、M の left-slope である。 □

図3. 1 の例では、実線部の node の列が left-slope であり、計 8 つの left-slope がある。最小の left-slope は 1 個の変数だけからなるものである。

定義 (入式のleft-slope-list)

left-slopeの定義から、最左のleft-slopeを除くすべてのleft-slopeは、その親のapplication-nodeと一対一に対応していることがわかる。(図3. 1 参照) そこで各application-nodeに互いに異なる正整数を割当て、各left-slopeに、

- 最左のleft-slope・・・0番
- その他のleft-slope・・・親のapplication-nodeの番号

と名前を付け、抜きだして表にしたものをleft-slope-listと言う。(図3. 2) □

例) $M \equiv \lambda x. (\lambda zy. (\lambda xy. x(yv))y)((\lambda w. w)w)((\lambda xv. v)x)$ の場合、

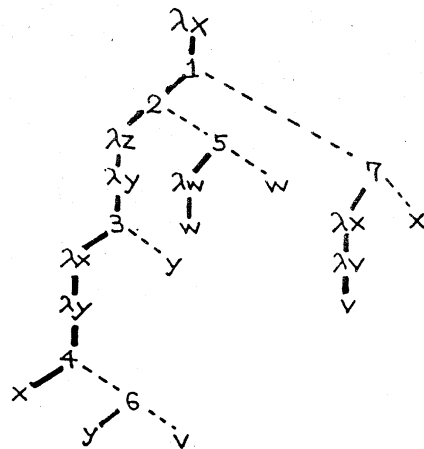


図3. 1 Mのleft-slope

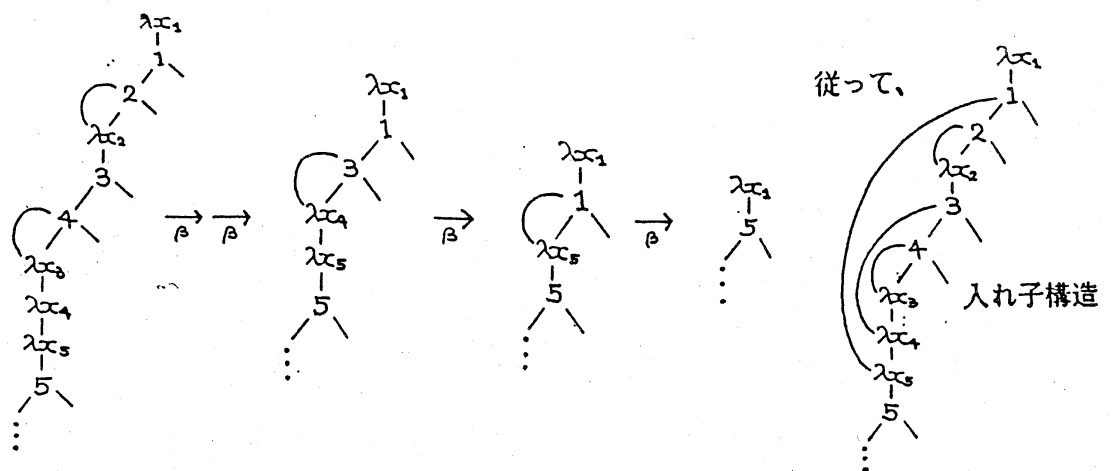
0	1	2	3	4	5	6	7
λx	γ	5	y	6	w	v	x
1	λx	λw		y			
2	λv	w					
λz	v						
λy							
3							
λx							
λy							
4							
x							

図3. 2 Mのleft-slope-list

4. Redexペア

入式のleft-slope上のnodeに対して、現在redexを構成しているか、または、あるreductionを行うことにより将来redexを構成するapplication-nodeとabstraction-nodeの対をredexペアと言う。

例えば、



以上の例でわかる様に、redexペアは元のleft-slopeにおいて、入れ子の関係にある。また、逆に入れ子の関係にある、application-nodeとabstraction-nodeの対はredexペアである。

5. LSアルゴリズムのアイデア

LSアルゴリズムのアイデアを通常のhead-reductionと比較をし、説明を試みる。図5. 1は、左にあるλグラフをhead-reductionして、そのhead-normal-formへ到達させたものを表わしている。

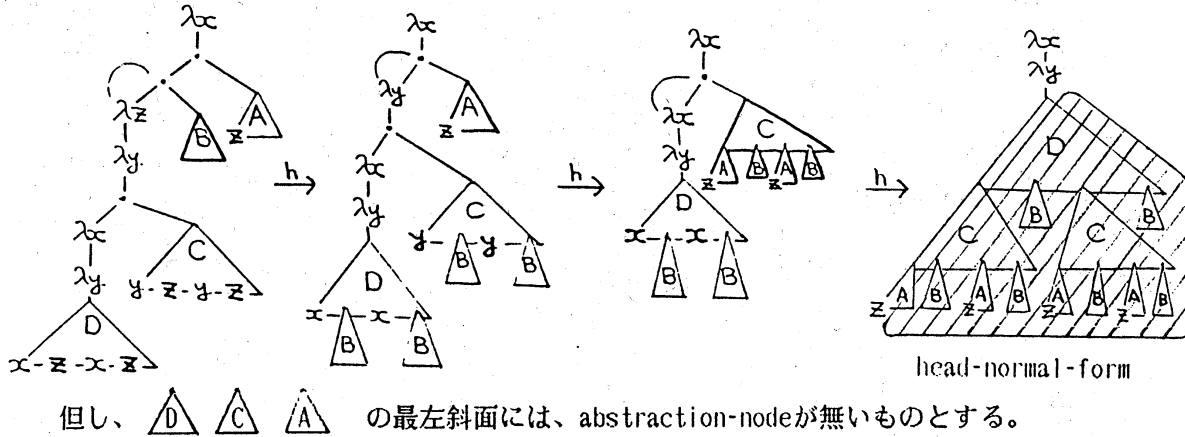


図5. 1 head-reductionの例

もし、head-normal-formの概形だけを求めたいのなら、head-normal-formの最左斜面のnodeだけがわかれば良いので、図5. 1の、head-normal-formの斜線部分への代入は無駄である。 $\triangle A$ $\triangle B$ $\triangle C$ $\triangle D$ が大きなλグラフで、更に α 変換が必要であるような場合は、斜線部への代入は非常に大きなタイムロスとなることは容易に理解できよう。そこで、LSアルゴリズムは、head-reductionのようにまずhead-redexを探しそれをreductionするというようなことはしないで、各ステップにおいて、まずλグラフの最左変数に注目する。そして、head-reductionにより将来この最左変数に代入されるλ部分グラフを見つけ、それを代入するというを行う。

最左変数に代入される部分式を見つけるためには、次の2つの情報が必要となる。

1. 最左変数を束縛するabstraction-node
2. 1で見つかったabstraction-nodeと将来redexを構成するapplication-node、つまりredexペアの関係にあるapplication-node

また、もし、1のabstraction-node、あるいは、2のapplication-nodeが無いときは、最左変数にもはや何も代入され得ないということになる。

LSアルゴリズムでは、バイパスを通すというアイデアを用いることにより、 α 変換の様な、やっかいな変数変換を行わずに、最左変数を束縛するabstraction-nodeを探すことを可能にしている。

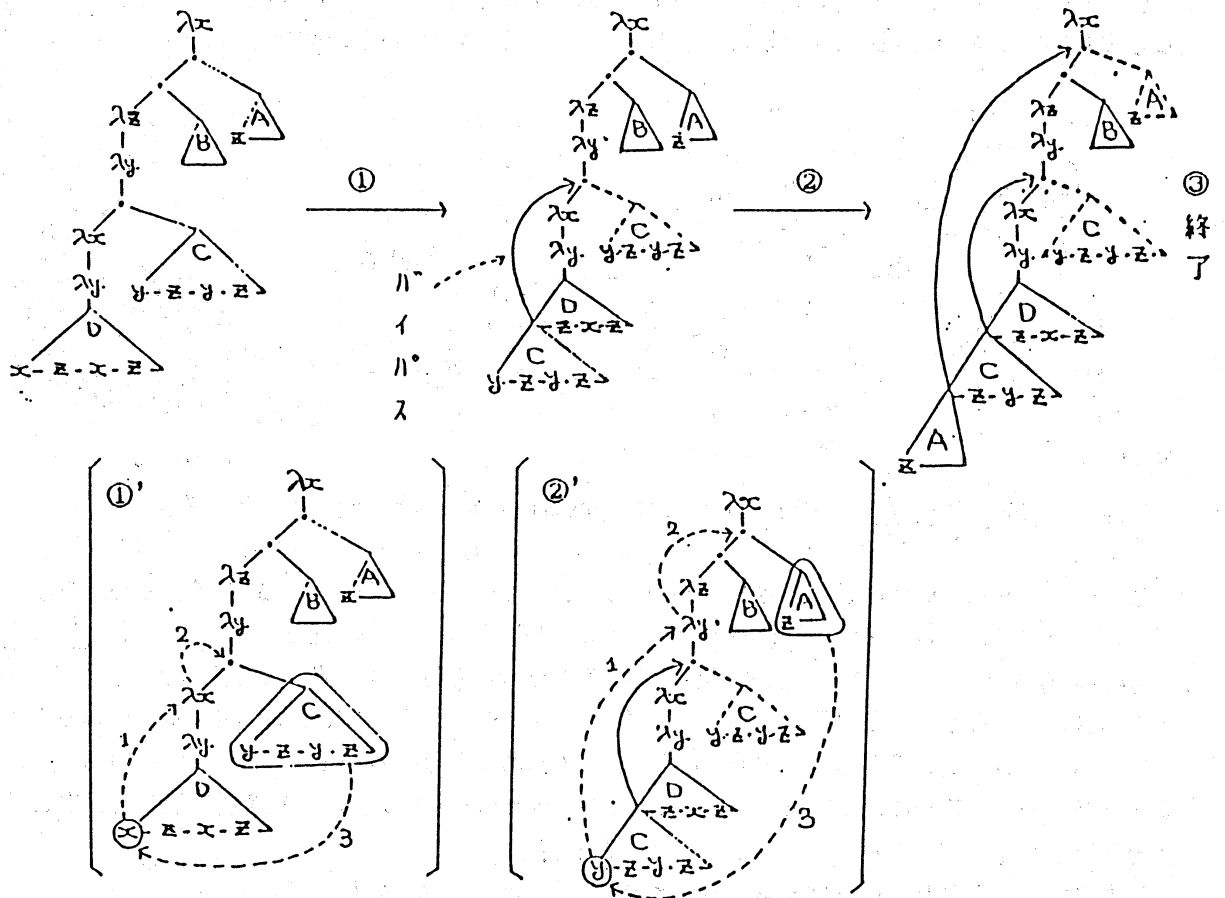
それでは、図5. 1の例について、具体的にこのアイデアを説明する。図5. 2を見て頂きたい。

図5. 2の解説：

①

1. まず、最左変数 x を束縛するabstraction-nodeを、最左変数から、λグラフの根に向かって探す。最初に見つかる λx が x を束縛するabstraction-nodeである。(①' - 1-→)
2. 束縛するabstraction-nodeが見つかったら、そのabstraction-nodeとredexペアとなるapplication-nodeを探す。(①' - 2-→)
3. 最左変数に、2で見つかったapplication-nodeの右側の部分木を代入する。(①' - 3-→)

図5. 2 LSアルゴリズムのアイデア



4. 3で代入した部分木の根から、その部分木の親であったapplication-nodeへバイパスを通す。

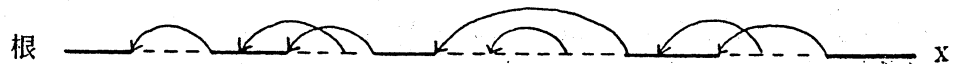
②

1. 最左変数 y を束縛する abstraction-node を、 λ グラフの根へ向かって、バイパスを経由して 探す。最初に見つかる λy が、最左変数 y を束縛する abstraction-node である。(②' - - $\rightarrow$) (注)

2. 3. 4. ... ①の2、3、4と同様。(②' 参照)

③ 最左変数 z を束縛する abstraction-node をバイパスを経由して、根に向かって探すが見つからない。従って、もはや最左変数 z には何も代入されないので終了。

(注) バイパスを経由して探す...もしあるnodeからバイパスが出ている時は、そのバイパスを経由して(すなわち途中のnodeを飛び越して)バイパスの終点から再び探し始めることを行う。例えば、次の様に、バイパスが通っている場合、実線部のnode上を探す。



以上のアイデアで、LSアルゴリズムは実行される。ここで、このアイデアで出来た、図5. 2の最終結果の λ グラフの、最左斜面上のnodeで、redexペアとなっているnodeをすべて取り除いてみ

但し、

- ・ L S-tableの最上段の数字の列は、その下にあるleft-slopeの番号の列を表わしている。
- ・ “/” が引かれている変数は、代入によりなくなった変数を表している。

L Sアルゴリズムは次のようになる。

L S アルゴリズム :

入力: λ 式 M 出力: MのL S-table

STEP 1) λ 式 M の left-slope-listを求める。

STEP 2) STEP 1 で作成されたleft-slope-listを参照して、次のようにL S-tableを作成する。

L S-tableの作成:

(0) 0番left-slopeをL S-tableに載せる。

(1) L S-tableの最左変数(注1)を束縛するabstraction-nodeをバイパスを経由して探す。
その時、 (注2)

(a) 束縛するabstraction-nodeが見つかる $\Rightarrow$ (2) へ

(b) 束縛するabstraction-nodeが見つからない $\Rightarrow$ 終了(作成されたL S-tableを出力)

(2) (1) で見つけたabstraction-nodeとredexペアとなるapplication-nodeを探す。
その時、 (注3)

(a) redexペアとなるapplication-nodeが見つかる $\Rightarrow$ (3) へ

(b) redexペアとなるapplication-nodeが見つからない $\Rightarrow$ 終了(作成されたL S-tableを出力)

(3) (2) で見つけたapplication-nodeの番号のleft-slopeを、left-slope-listから抜き出し、L S-tableの最左変数に代入する(注4)。さらに、代入したleft-slopeの頭から、(2) で見つけたapplication-nodeへバイパスを通す。

(1) へ戻る。

□ {L S アルゴリズム}

注1) L S-tableの最左変数・・・L S-tableにおいて最左にあるleft-slopeの(一番下にある)変数(図6.1では、4番left-slopeの変数“y”)

注2) バイパスを経由して探す・・・§1の説明と同様

注3) redexペアとなるapplication-nodeを探す・・・L S-table上の各left-slopeを、変数を取り除き、つなぎ合わせたものを一つのnodeの列と見なしredexペアを探す。

注4) 代入する際、代入するleft-slopeの番号を上段に書く。

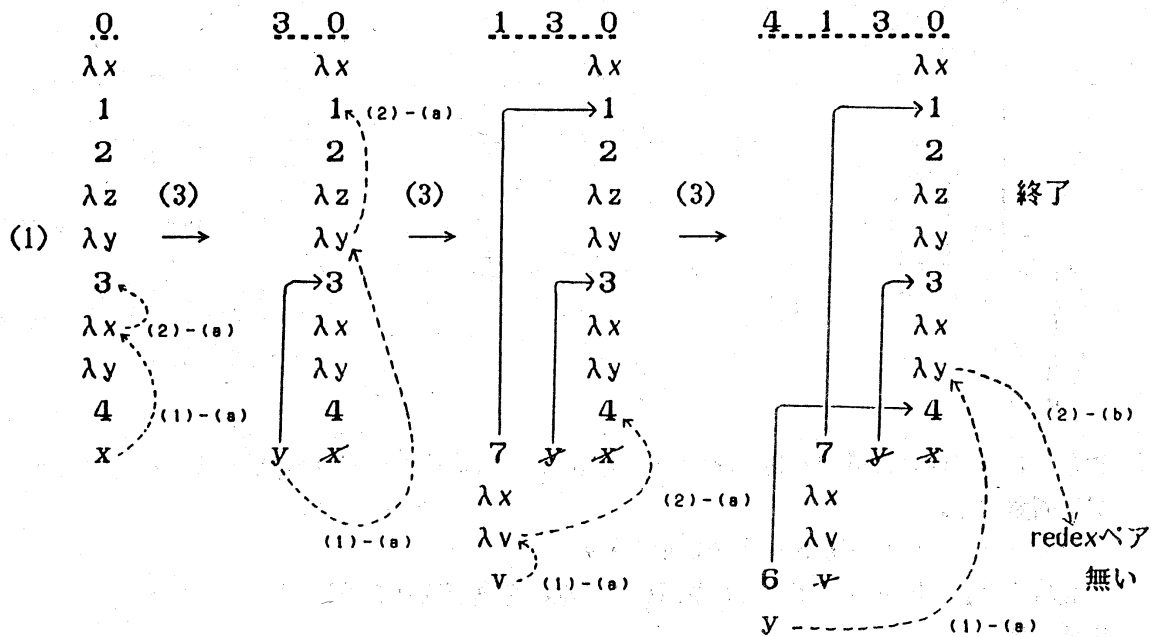
LSアルゴリズムが終了するのは、(1) - (b)、または、(2) - (b) の条件が成り立つ場合、つまり、LS-tableの最左にあるleft-slopeの変数に、もはや、何も代入され得ない時に終了する。それでは、LSアルゴリズムが、実際にどの様に実行されるか、図6.1の場合の実行例を示す。

実行例：

入力： $M \equiv \lambda x. (\lambda zy. (\lambda xy. x(yv))y) ((\lambda w. w)w) ((\lambda xv. v)x)$

STEP 1) 図6.1のleft-slope-listを作成する。

STEP 2)



head-normal-formを持たない入式 $(\lambda x. x x) (\lambda x. x x)$ に対して、LSアルゴリズムを実行すると図6.2のようになる。LSアルゴリズムは、head-normal-formを持たない入式に対しては止まらないアルゴリズムである。

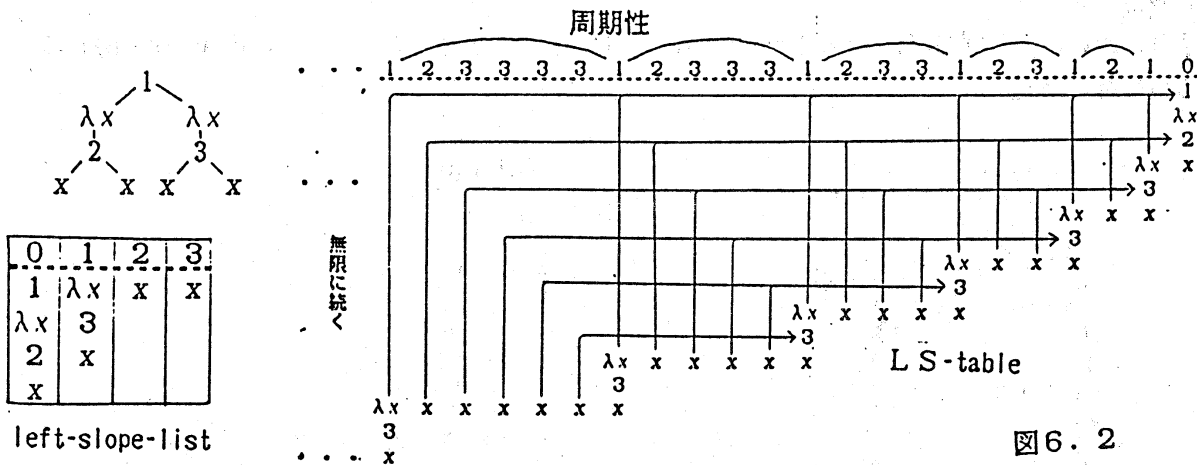


図6.2

7. LSアルゴリズムによって得られる情報

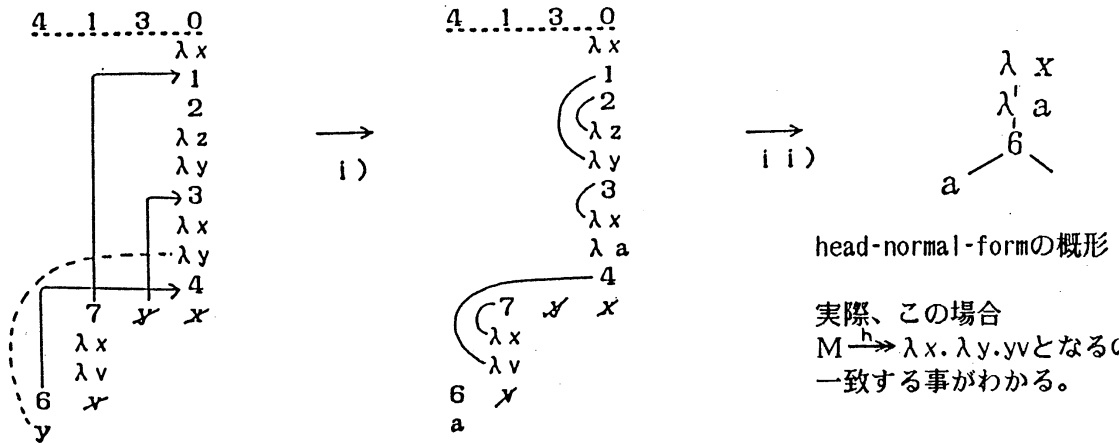
(1) head-normal-formの概形

“5. LSアルゴリズムのアイデア”で説明したように(図5.4)、LSアルゴリズムで作られたLS-tableから、次のi), ii)を順に行う事により求めることが出来る。(図7.1参照)

i) LS-tableの最左変数と、それを束縛するabstraction-node(バイパス経由で見つけたもの)の変数を、新しい変数に換える(図7.1では、a)。

ii) redexペアとなっているapplication-nodeとabstraction-nodeをすべて取り除く。

出来上がったnodeの列が、head-normal-formの概形を表す。図6.1の例に対しては、図7.1のようになる。



また、LS-tableの情報とleft-slope-listから、head-normal-formも求めることが出来るが、ここでは省略する。

(2) head-needed-redex

文献[1] Theorem 3.6-(ii)を用いて、次の定理が成り立つ。

定理 7.1

Rは、λ式 M の head-needed-redex である

$\iff$ Rは、MのLS-tableに現れるleft-slope上のredexである

証) 略□

つまり、図6.1の例の場合、head-needed-redexは、図7.2のλグラフで、abstraction-nodeに*の付いたredexとなる。

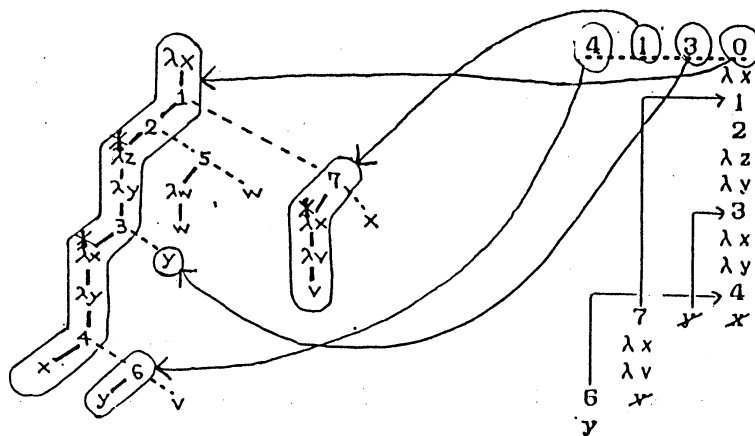


図7.2

8. LSアルゴリズムの実行効率に関して

一般のグラフィダクションと比較して、以下のようなことが言える。

LSアルゴリズムの特徴：

1. グラフィダクションにとって、必要不可欠で厄介な α 変換に対応することはやらない。
2. head-normal-formの概形を得るのに不必要な部分はいじらない。

LSアルゴリズムのインプリメントに関して：

LSアルゴリズムは、(1) left-slope-listを作るルーチン、(2) LS-tableを作るルーチン、の2つからなるが、それぞれに対して

(1) left-slope-listを作るということは、基本的には構文解析を行うことである。従って、λグラフを作るのと同程度の実行効率のインプリメントが出来る。

(2) LS-tableは、left-slopeの番号の列と、バイパスの情報をうまくコード化したもので表現できる。従って、LS-tableをコードで表現することにより、いちいちleft-slope-listからleft-slopeを抜き出し、コピーするということが必要ないため、高速にLS-tableを作成することが出来る。

LSアルゴリズムのオーダーに関して：

このLSアルゴリズムのオーダーは定まらない、その理由は

1. head-normal-formを持たない入式に対しては止まらない
2. アルゴリズムが止まる場合、つまりhead-normal-formを持つ入式だけに限ってもオーダーは定まらない。それは、もしオーダーが定まってしまうと、head-normal-formを持つか否かという問題が決定可能となってしまう、有名なSCOTTのUndecidability-Theorem (文献[3]、Theorem 5.6) に矛盾してしまうからである。

計算機実験に関して：

LSアルゴリズムは、オーダーが定まらないアルゴリズムであるが、現実的には、graph-head-reductionと比較すると、かなり効率の良いアルゴリズムであることが計算機実験により、確かめられた。いくつかの入式に対して、次のA、B2つの実験を行った。

A. graph-head-reductionにより、head-normal-formを求める。

B. LSアルゴリズムにより、LS-tableを作成する。

λグラフとleft-slope-listは、ほぼ同時間で作成出来るので、A、Bからそれぞれ、λグラフ、left-slope-listをつくる時間を、除いた実行時間を調べたところ。BはAより、2倍から280倍速く実行されるという結果がでた。LS-tableが求まれば、ただちにhead-normal-formの概形、head-needed-redexが求まるので、LS-アルゴリズムは有効なアルゴリズムであるといえるであろう。実際、LSアルゴリズムのほうが、いくらでも速くなるような入式を、人工的に作ることが出来る。

謝辞:

本研究において、非常に貴重な御意見、御助言をいただいた東京工業大学の實来正子先生に深く感謝致します。

参考文献

- [1] H.P.Barendregt, J.R.Kennaway, J.W.Klop, M.R.Sleep, Needed reduction and spine strategies for the lambda calculus (Information and computation vol.75 pp.191-231,1987)
- [2] H.P.Barendregt, The Lambda Calculus (North-Holland,1981)
- [3] J.R.Hindley & J.P.Seldin, Introduction to Combinators and λ -calculus (Cambridge University Press,1986)
- [4] S.L.P.Jones, The Implementation of Functional Programming Languages (Prentice Hall,1987)
- [5] 野口憲一, A new head-normalization algorithm for λ -calculus (東京工業大学 修士論文 1990)
- [6] G.Revesz, Lambda-Calculus , Combinators and Functional Programming (Cambridge tracts in theoretical computer science; v.4,1988)