可変容量セルの効率的なくず集めについて†

寺島元章†† 佐藤和美††*

可変容量セルに対する効率的なくず集め法の設計とその実現，および評価について述べる．このくず集めは古典的圧縮法を改良したもので，改良型圧縮法と呼ばれる．その特徴は，補正表をスタック領域の未使用部分に作成することによる記憶領域の効率的な利用と，ポインタ補正と圧縮の処理の高速化にある．FLA（完全遅延評価系）に具現された改良型圧縮法と既成の圧縮法の両者のくず集めの処理時間を比較し，改良型圧縮法の優位性を示した．また，複写方式のくず集めとの理論的な処理時間の比較を行い，くず集め時の現役セル比率が大きい場合，改良型圧縮法の処理時間が複写方式の2倍以内になることも示した．

1. はじめに

本論文は，可変容量セル（容量が固定でないセル）に対する効率的なくず集め（garbage collection）法の設計とその評価に関するものである．くず集めは，Lisp や Algol など，動的データ構造を有する言語処理系に必須の記憶管理機能である．そのため，くず集めに必要な時間計算量（time complexity）や領域計算量（space complexity）を改善するための新技法や，機能の質的変更である分散化や並列化などの論文[1]がこれまでに数多く発表されている．

本論文で提案するくず集めは，可変容量セルを対象とした古典的圧縮法[2]の改良版になる．この改良型圧縮法の特徴は，ポインタ補正表をスタック領域の未使用部分に作成するという記憶領域の効率的な利用と，ポインタ補正と圧縮に要する処理時間の短縮にある．この処理時間に関する定量的な解析を計算量の理論に基づいて述べる．

このくず集めは Sun 3/50 上で動作中の FLA（完全遅延評価系）に具現（インプリメンテーション）されている．その性能測定に基づいて，Morris の圧縮法[3]や可変容量セルのくず集めとして従来から定評のある複写法と比較した評価についても述べる．

2. くず集め

2.1 圧縮方式 vs 複写方式

可変容量セルのくず集めは，複写方式と圧縮方式とに大別される．前者は，セルの格納領域（heap）を2つの半領域（semi-space）に分割し，現役（使用中）のセルを交互に複写することで退役（使用済）のセルをふくむ格納領域を一括回収するという方式である．後者は現役のセルをそれらの配置順序を保存したままで同じ格納領域の一方に圧縮する方式である．表1はこの両者の特徴を対比させて示したものである．圧縮方式は格納領域量で複写方式より優位にあるが，処理時間では複写方式に劣る．後者の要因は格納領域の複数回の走査とポインタ補正による負担増である．

表1 圧縮方式と複写方式の特徴比較
Table 1 Comparison of characteristics between compactifying and copying methods.

	処理時間	格納領域
圧縮方式	セルの総容量（格納領域量）に比例	領域は1つでよい
複写方式	現役セルの総容量に比例	区別可能な2つの(同容量の)半領域が必要

2.2 圧縮方式

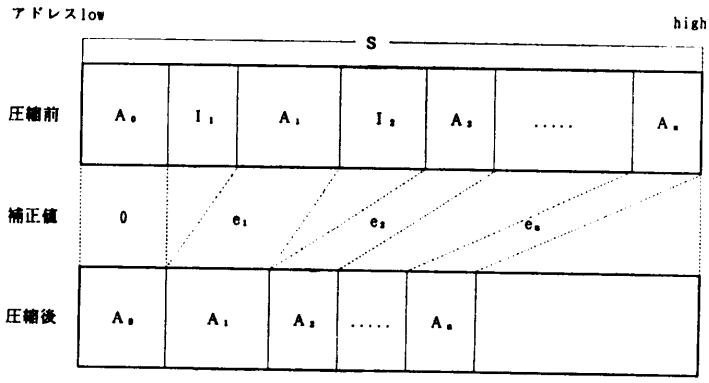
圧縮方式には，ポインタ補正と再配置（圧縮）とを別個の手順として行う方法と，それらを混ぜて1つの手順として行う方法とがある．前者には古典的圧縮法や Jonkers[4]の圧縮法が，後者には Morris の圧縮法がある．

2.2.1 古典的圧縮法

図1は，圧縮に伴うポインタ補正のしくみを模式的に示したものである．現役セル中の各フィールドの特定ビット（これをマークビットと呼ぶ）に印付けを行うマーキングの手続きが済むと，格納領域は現役セルが連続した1つの塊（A_i, $i=0..n$, 以下単に現役塊と呼ぶ）とそれらの塊の間に存在する退役セルの塊（I_j, $j=1..n$, 以下単に退役塊と呼ぶ）とに分割される．そこで，各現役セルをアドレスの小さいほう（low 側）に圧縮することを考えると，A_i は $-|I_1|-|I_2|-\cdots-$

† A Garbage Collector Efficient for Varisized Cells by MOTOAKI TERASHIMA and KAZUMI SATO (Department of Computer Science, University of Electro-Communications).

†† 電気通信大学情報工学科
* 現在 ソニー(株)スーパーマイクロ事業本部



S : 格納領域 $S=A_0 \cup I_1 \cup A_1 \cup I_2 \cup \cdots \cup A_n$
A_i : i 番目 $(0 \leqq i \leqq n)$ の現役セル塊
I_i : A_{i-1} と A_i $(1 \leqq i \leqq n)$ の間の退役セル塊
e_i : A_i $(1 \leqq i \leqq n)$ の補正値

図 1 現役セルの圧縮
Fig. 1 Compactifying of active cells.

$|I_i|=e_i$ だけアドレスが変化する．絶対値記号はアドレスに換算した容量を表す．そこで，A_i 中のセルを指していたポインタは圧縮後も同じセルを指すように補正値 (e_i) が加えられる．この作業のことをポインタ補正と呼ぶ．

ポインタ補正は，セルの格納領域以外の記憶領域（これを補助記憶と呼ぶ）を使用すると，1個のポインタについて $O(1)$ で行える[2]．その原理は非常に単純で，格納領域を走査しながら，その $2^m \cdot i+\mathrm{low}$ 番地 $(m>0)$ の補正値を i 番目の成分 $(E[i])$ にもつ補正表を補助記憶に作成し，この表とセルの格納領域の印付け情報から p 番地 $(2^m \cdot j+\mathrm{low} \leqq p < 2^m \cdot (j+1)+\mathrm{low})$ の補正値を(2.1)式で算出することである．ただし，この「番地」とはセルの各フィールドを単位とするアドレスのことである．以下，「番地」はこの意味で用いられる．

$$E[j]-\|\{k \mid 2^m \cdot j+\mathrm{low} \leqq k < p \text{ かつ } \mathrm{unmarkedp}(k) \text{ が真}\}\| \quad (2.1)$$

ただし，$\|S\|$ は集合 S の要素数を表す．補正数は，$E[0]=0$, $E[i] \leqq 0$ $(i \geqq 1)$ である．なお，圧縮は low 側に行うものとする．

この方法の問題点は，補正表のために補助記憶が必要になること，(2.1)式の要素数の算出に特殊なハードウェアが使用できないとその算出に必要な時間計算量は厳密な意味で定数でなく 2^m に比例することである．ここで，mを小さくすることは補助記憶容量の増加を意味する．

2.2.2 Morris の圧縮法

Morris は，セルの格納領域を行きと帰りの2回の走査でポインタ補正と圧縮とが同時に行えるアルゴリズムを考案し，その正当性を証明した[1),3)]．これが，Morris の圧縮法と呼ばれているものである．この方法はアルゴリズムが簡潔であるという利点もあるが，具現上の問題点がいくつかある．すなわち，

（1） 1個のポインタについて，マークビットのほかに追加ビットが1つ必要である．Morris 自身が指摘しているように，処理速度を重視すると，このビットはセル内に確保してポインタと一緒に読み取らなければならない．これは，データの内部表現を決める上で大きな制約事項となる．

（2） 現役セルへのポインタに対して，ビット操作と「ポインタ交換」が必要である．また，共有セルの存在はその応分の「ポインタ交換」を必要とする．これは，処理時間が格納領域量以外にも比例項をもつことを意味する．

（3） データの内部表現がタグ付となる場合，ビット操作と「ポインタ交換」に関してアルゴリズムの手直しが必要である．この手直しは，代入と比較（ビット検査）演算を増やすことから，処理時間を増加させる．

（4） マーキングの手続きで，印付けしたセルの容量の累計を行う．これは，補正値 (e_n) をあらかじめ算出しておく必要があるからである．

Morris の圧縮法によるくず集めの処理時間とその評価については 4.3 節で述べる．

2.2.3 Jonkers の圧縮法

Jonkers は，追加ビットなしに格納領域の1回の走査でポインタ補正と圧縮とが行えるアルゴリズムを考案した[1),4)]．ただし，この圧縮法は，現役セルが非ポインタ成分を少なくとも1つもつという特殊な場合を仮定している．したがって，この方法は一般的な処理系への具現には不適当である．

2.3 改良型圧縮法

本論文で提案する改良型圧縮法は，2.2.1 項で述べた圧縮方式の問題点に次のように対処し，記憶領域の効率的な利用と処理の高速化とを実現したものである．

（1） 特別な補助記憶は不要

補正表はスタック用領域の未使用部分に確保する．スタックはマーキング終了後，くず集めが終るまで使

用されないので，この部分は安全かつ自由に使用できる．補正表の大きさはスタック未使用部の容量変動に対応して実行中に動的に決めることができる．

（2） 現役塊を連鎖にする

現役塊は2.2.1項で述べたように格納領域中に散在する．しかし，退役塊の先頭フィールドを利用してこれらを連鎖にすると，各現役塊は論理的に結合された状態になる．すなわち I_j $(j=1..n)$ の先頭に置かれた A_j への無印のポインタを検知することで A_{j-1} から A_j に進むことができる．これを利用すると，ポインタ補正や圧縮の対象であるすべての現役セルの検出は格納領域量ではなく，現役セルの総容量に比例する時間で行うことができる．なお，この連鎖の作成は補正表作成時にほとんど時間的負担なく行うことができる．

（3） 補正値の算出を高速化する

（a） 補正値が0である塊 (A_0) の早期検出

A_0 が low 番地から始まる場合，A_0 を指すポインタの補正値は0である．そこで，(2.2)式に示す neg 番地を補正表作成時に求めておき，補正対象のポインタ (p) と最初に比較する．$p<\mathrm{neg}$ ならば補正値は0で，その後の処理は不要である．そうでなければ(b)を行う．

$$\mathrm{neg}=\begin{cases}\mathrm{low}+|A_0| & A_0\text{ が low 番地から始まる場合}\\ \mathrm{low} & A_0\text{ が low 番地から始まらずに，その前に退役塊がある場合}\end{cases} \tag{2.2}$$

なお，neg 番地の算出も補正表作成時にほとんど時間的負担なく行うことができる．

（b） 二分走査

(2.1)式の要素数の算出は，2^m 番地単位に分割されたセル格納領域の1区画を，その先頭番地 $(2^m\cdot j+\mathrm{low})$ から p 番地（ただし $j=\llcorner(p-\mathrm{low})/2^m\lrcorner$）まで走査しながら無印のフィールドを数えることである．二分走査は，この区画をさらに二等分したその一方の半区画の走査だけで補正値を求める方式である．すなわち，$(p-\mathrm{low})\bmod 2^m<2^m/2$ ならば low 側の半区画を，そうでなければ high 側の半区画を走査する．二分走査により，各フィールドの探索回数の期待値は $2^m/2$ から $2^m/4$ になり，処理時間が 1/2 に短縮される．ただし，high 側の半区画を走査した場合の補正値は(2.3)式で与えられる．

$$E[j+1]+\|\{k\mid p<k<2^m\cdot(j+1)+\mathrm{low}\text{ かつ }\mathrm{unmarkedp}(k)\text{ が真}\}\| \tag{2.3}$$

走査すべき半区画は $p-\mathrm{low}$ を二進表現したときの下位から n 番目のビット状態で決まる．このビットは，アセンブリ言語を用いて，2^m の除算と $\bmod 2^m$ の剰余算とをシフト演算にしたときの剰余側の最上位ビットになるので，その検知は容易かつ高速に行うことができる．

3章で述べる FLA 処理系では，$m=3$ として具現した．これは多くの実験データに基づいた最適値ではないが，これより大きい値ではポインタ補正時間が格段に増加し，逆に小さい値では補正表の大きさとその作成時間が無視できないという経験的な値である．$m=3$ では，ポインタ補正時の探索回数の期待値は2に，補正表の容量はセルの格納領域量の 1/8 となる．この容量はスタック容量よりも小さい．ただし，（1）項で述べたように m の値は実行中に3より大きくなることがある．

2.4 時間計算量

2.4.1 補正表の作成

補正表の作成では，セルの格納領域が1回走査される．その走査とは，各セルのマークビットを検査し，印無しセルの累計値を 2^m 番地 $(m>0)$ ごとに補正表に書くことである．この手順の時間計算量は1セルにつき $O(1)$ であることから，補正表作成の時間計算量は $O(M)$ となる．M はセルの総容量（格納領域量）である．

2.4.2 ポインタ補正

2.3 節の(3)項によると，ポインタ1個の補正に必要な時間計算量は(2.4)式で与えられる．以下，補正値が0である不動の A_0 のことを単に A_0 と記述する．

$$\begin{aligned}&O(1) && \text{ポインタが } A_0 \text{ を指すとき } (p<\mathrm{neg})\\ &O(2^m/4) && \text{それ以外 } (p\geqq\mathrm{neg})\end{aligned} \tag{2.4}$$

$O(2^m/4)$ と $O(1)$ の時間計算量をそれぞれ $d\cdot U$ と U とで表すと，ポインタ補正の時間計算量は(2.5)式で与えられる．d は比例定数 $(d>1)$ である．

$$\begin{aligned}&d\cdot U\cdot(N_P-N_{P_0})+U\cdot N_{P_0}+U\cdot N_T\\ &\quad=M_A\cdot U\cdot(d-(d-1)\cdot f)-U\cdot(d-1)\cdot N_{P_0}\end{aligned}$$

N_P ：現役セル中のポインタの総数

N_{P_0} ：A_0 を指すポインタの総数

N_T ：現役セル中のポインタ以外のデータ（非ポインタ）の総数

M_A ：現役セルの総容量 $(M_A=N_P+N_T)$

f ：非ポインタ比率 (N_T/M_A) (2.5)

ここでは，ポインタ補正の対象でないデータに対する

時間計算量も A_0 を指すポインタと同じく $O(1)$ とした．これは(2.5)式を単純化するための便法であり，このために一般性を失うことはない．また，現役塊をつなぐ連鎖の存在により，すべての現役セルの走査は現役セルの総容量に比例する時間で行うことができる．そこで，次の仮定を導入する．

H-1：f（現役セル中の非ポインタ比率）は定数である．

この仮定は現役セルの個数（あるいは容量）が十分大きいとき統計的に成り立つ．この仮定のもとで，(2.5)式は次のように書き直すことができる．

$$O(M_A - d_1 \cdot N_{P_0})$$
$$d_1 = (d-1)/(d-(d-1)\cdot f) : \text{定数} \qquad (2.6)$$

このようにO表記は，（ ）内の項に係る比例項と定数を省く意図で用いられる．この点が計算量の表記に用いられる通常の O 記法とは異なる．(2.6)式は，ポインタ補正の時間計算量が現役セルの総容量から N_{P_0} の寄与分を減じたものに比例することを示している．これが，古典的圧縮法のポインタ補正の時間計算量 ($O(M)$) と本質的に異なる点である．

そこで，(2.6)式の意味を直感的に理解するために，次の仮定を行う．

H-2：A_0 と他の現役塊との相互参照ポインタの総和は A_0 内の相互参照ポインタ数に比べて十分小さい．

この仮定はセルの共有や書換えの操作が少ない場合や現役塊中に占める A_0 の比が十分大きい場合に成り立つ．前者は現役塊間のポインタを少数に留め，後者は塊（特に A_0）内に相互参照ポインタ数を増大させるからである．一般に，くず集めが頻繁になると，現役塊中に占める A_0 の比は大きくなる．これは，現役期間（生成されてから退役までの期間）の長いセルほど A_0 に集積されることから立証される．

H-2 の仮定が成り立つとするならば，N_{P_0} を $(1-f)\cdot M_{A_0}$ (M_{A_0} は A_0 の容量) で置き換えることができる．そこで，ポインタ補正の時間計算量に関する(2.7)式が得られる．

$$O(M_A - d_2 \cdot M_{A_0})$$
$$d_2 = (d-1)\cdot(1-f)/(d-(d-1)\cdot f) : \text{定数} \qquad (2.7)$$

(2.7)式は，ポインタ補正の時間計算量が現役セルの総容量から A_0 の寄与分 ($d_2 \cdot M_{A_0}$) を減じたものに比例することを示している．これは，くず集めが頻繁で A_0 の容量比が大きくなるほど，A_0 の寄与分が正確かつ大きく時間計算量に反映され，ポインタ補正での時間短縮の効果が上がることを意味する．その量的評価については4.2節で述べる．

2.4.3 圧　　縮

圧縮とは，現役塊（セル）を順に走査しながら，マークビットの印をはずし，セルを補正値分再配置することである．この手順の時間計算量は1セルにつき $O(1)$ である．2.4.2 項で述べたように，すべての現役セルの走査は $O(M_A)$ の時間計算量で行うことができることから，圧縮の時間計算量は $O(M_A)$ となる．

2.5 マーキングと複写

マーキングはすべての現役セルの成分を1回調べることから，その時間計算量は現役セルの総容量に比例する．これは公知の事実であり，マーキングアルゴリズムの高速化とはその比例係数を小さくすること，すなわち，アルゴリズムの作業量を少なくすることにほかならない．**図2**の A-1 と A-2 は，C言語で記述されたマーキングアルゴリズムである．図中に使用されている型と関数は次の意味をもつ．

cell_field　：セルの各成分

```
A-1. marking algorithm

mark(p) cell_field *p;
{int i;
 if unmarkedp(p)
   for (i=0; i<cell_size(p); i++)
    {mark_field(p+i);
     if cellp(p[i]) mark(p[i]);};
};

A-2. fast marking algorithm

mark(p) cell_field *p;
{int i;
   push(-1);
M: if unmarkedp(p)
     for (i=0; i<cell_size(p); i++)
      {mark_field(p+i);
       if cellp(p[i]) push(p[i]);};
   pop(p); if (p>=0) goto M;
};

A-3. copy algorithm

cell_field *copy(p) cell_field *p;
{cell_field *q, *r; int i;
 if copiedp(p) return(p);
 if copiedp(*p) return(*p);
   else {q=(r=allocate(cell_size(p)));
         push(-1); goto C2;};
C1: if copiedp(*p) q[0]=*p;
      else
     {q=(q[0]=allocate(cell_size(p)));
C2:   for (i=0; i<cell_size(p); i++)
        if cellp(q[i]=p[i]) push(q+i);
      p[0]=q;};
    pop(q); if (q>=0) {p=*q; goto C1;};
 return(r);
};
```

図2 マーキングアルゴリズムと複写アルゴリズム
Fig. 2 Marking algorithms and copy algorithm.

unmarkedp : セルが印付けされていないか
cell_size : セルの容量（成分の個数）
mark_field : セルの該当フィールドに印付けする
cellp : セルを指すポインタか
copiedp : 複写されたセルか
allocate : 領域の確保
push : スタックへプッシュ（マクロ）
pop : スタックからポップ（マクロ）

ただし，ポインタはセルの先頭フィールドのみを指し，遅くともそこではセルの容量が読み取れるものとする．

A-1 は可変容量セルに対する再帰的アルゴリズムで，公知のものである．A-2 は A-1 の再帰を除去したもので，こうした再帰の除去は公知の技法である．当然，A-2 は A-1 よりも高速である．ポインタ値は非負数であり，最初にプッシュされる負数（−1）は空スタック（stack empty）に対する番兵である．A-2 はスタックを使用しない Schorr & Waite のアルゴリズムを可変容量セル用に変更したもの[5]より高速である．理由は，スタック代わりに用いたセルを復元するという負担増が後者に存在するからである．

複写アルゴリズムは，マーキングアルゴリズムに似せて作ることができる．それは，両者ともセル（の各成分）を再帰的にたどることで目的が達成されるからである．前者の目的はセルの各フィールドに印を付けることであり，後者の目的はセルの各成分を別の領域に複写し，その行き先番地（forwarding address）を元のセルに残すことである．

A-3 は，A-2 に類似させた可変容量セル用の複写アルゴリズムである．この両者によるくず集めの処理時間を得るには，それらを具現したくず集めの処理時間を測定すればよい．しかし，この方法では，両者の具現が必要である．そこで，A-2 と A-3 の構文要素が変換される対象計算機命令語の実行時間を基に両者の（理論的な）処理時間比を算定し，対象計算機でのマーキング(A-2)の処理時間の実測値から複写(A-3)の処理時間，すなわち複写方式のくず集めの処理時間を算定する．この方法では，A-3 の具現が不要である．対象計算機を Sun 3/50，その命令語を MC 68020 としたときの 1 個のセルに対する処理時間比は(2.8)式で与えられる．**表 2** は，算定の基礎となった構文要素と命令語の対応関係である．なお，MC 68020 の命令

表 2 構文要素と対応命令語
Table 2 Program elements and their corresponding instruction sets.

構文要素		MC 68020 命令語	clock 周期	IBM 370 系命令語	
代入文	p[c]=q	move.l Dn, d(Am)	[7]	ST	Rn, d(Rm)
	p=q[i]	move.l (An₁, Dn₂), Dm	[12]	L	Rm, (Rn₁, Rn₂)
	p[i]=q[j]	move.l (An₁, Dn₂), (Am₁, Dm₂)	[17]	L	Rw, (Rn₁, Rn₂)
				ST	Rw, (Rm₁, Rm₂)
引用文	push	move.l #n, −(An)	[8]	S	Rn, #n
		move.l (Am₁, Dm₂), −(An)	[14]	LA / L	Rw, n / Rw, (Rm₁, Rm₂)
				ST	Rw, Rn
	pop	move.l (An)+, Dm	[7]	LA	Rm, (Rn)
				LA	Rn, d(Rn)
	allocate	addq #n, An	[3]	LA	Rn, d(Rn)
	cell_size	move.l d(An), Dm	[7]	L	Rm, d(Rn)
	mark_field	ori #n, (An, Dm)	[20]	L	Rw, (Rn, Rm)
				OR	Rw, #n
				ST	Rw, (Rn, Rm)
条件文	if(p>=0)	cmp.l #n, Dn	[8]	LTR	Rn
	if(copied)	beq.s d	[5, 9]	BE	d
	if(unmarked)	btst #n, Dn	[5]	C	Rn, #n
	if(cellp)	beq.s d	[5, 9]	Bxx	d
goto 文		bra.s d	[9]	B	d
for 文		cmp.l d(Dm), Dn	[9]	C	Rn, d(Rm)
		bgt.s d₁	[5, 9]	BG	d₁
		.....			
		addq #1, Dn	[3]	LA	Rn, d(Rn)
		beq.s d₂	[9]	B	d₂

Am, Am₁, An, An₁: アドレスレジスタ，Dm, Dm₂, Dn, Dn₂: データレジスタ，Rm, Rm₁, Rm₂, Rn, Rn₁, Rn₂, Rw: 汎用レジスタ

語体系が特異なものでないことは，そのクロック周期と併記したメインフレーム計算機（IBM 370 系）の命令語とを対比すれば明らかである．

$$r_{A_3/A_2}=\frac{74n_P+64n_T+90}{62n_P+52n_T+24} \quad (2.8)$$

n_P：セルを指すポインタの個数

n_T：それ以外の成分の個数

数値はクロック周期を表す．

実際のマーキングや複写では，その対象である現役セルの個数（容量）は十分に大きいとしてよい．そこで，n_P を $(1-f)\cdot M_A$ に，n_T を $f\cdot M_A$ に置き換え，$M_A \gg 1$ とすると，処理時間比は(2.9)式となる．

$$R_{A_3/A_2}=\frac{(74-10f)\cdot M_A}{(62-10f)\cdot M_A}=1+12/(62-10f) \quad (2.9)$$

f（非ポインタ比率）は 0〜1 であるので，処理時間比（R_{A_3/A_2}）は，1.19〜1.23 の範囲となる．すなわち，複写方式のくず集めの処理時間はマーキングの処理時間の約 1.2 倍と算定される．ただし，この比率は対象計算機を Sun 3/50 に規定した場合であり，対象計算機が異なればこの比率も変化する．

3. FLA 処理系

FLA 処理系に関する詳細な記述は本論文の主題ではないので，ここでは処理系の概要と処理系が扱うデータの内部表現であるデータ構造について述べることにする．

3.1 処理系

FLA 処理系は環境スタックを扱う仮想 Lazy SECD 計算機[6]に基づいている．それは，関数型言語の一方言である Lispkit-lisp[7]で記述されたプログラムを仮想計算機の機械語に翻訳し，それをさらに対象計算機である Sun 3/50 の機械語（MC 68020 命令語）に変換するという方式の処理系である．一般に，遅延評価はクロージャ（閉包）を用いて容易に行うことができるが，それより制約の強い完全遅延評価はクロージャでは不十分である．そこで，完全遅延評価から遅延評価へのプログラム変換を行う lambda-hoisting[8]方式が考案されており，FLA 処理系もそれを採用している．

lambda-hoisting と仮想機械語を生成する翻訳系は，Lisp で記述されている．仮想機械語を MC 68020 命令語に変換するコード生成系はC言語で記述されている．コード生成系は仮想機械語を MC 68020 命令語に変換する場合に，生成される命令語が単純かつ短いものはその命令語を，そうでないものは実行時評価ルーチンへの呼出し命令語を生成する．この実行時評価ルーチンはアセンブリ言語（MC 68020 命令語）によって記述されている．

3.2 データ構造

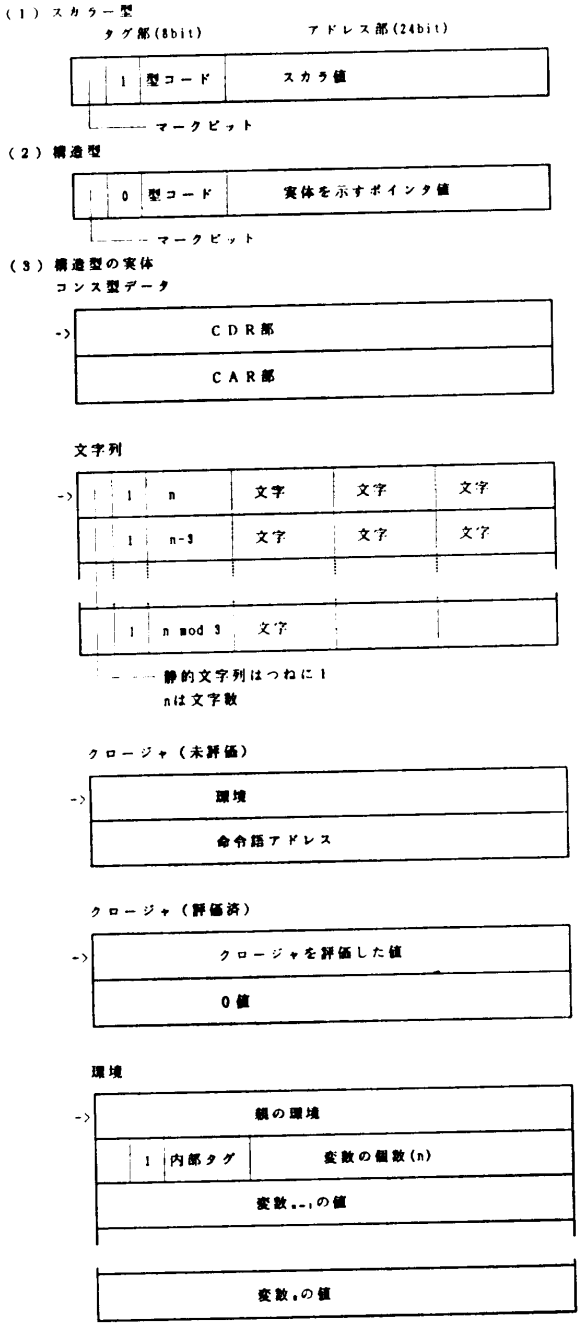
FLA のデータ構造は，その言語仕様で規定されたデータ型に対するものと処理系専用のものとがある．前者のデータ型には，整数，文字，論理値，コンス型データ，文字列やシンボルがある．後者には環境とクロージャがあり，これらは FLA が仮想 Lazy SECD 計算機に基づくために必要となるものである．

データ型は対象計算機の1語を用いて表現される．その1語は8ビットのタグ部と24ビットのアドレス部とからなる．タグ部の先頭ビット（b_0）はマークビットである．次の1ビット（b_1）はそのアドレス部がポインタ補正の対象となる（$b_1=0$）か否（$b_1=1$）かを決める．この2ビットを除いた残りの6ビットが型コードと呼ばれるポインタタグで，そのアドレス部の値の型を規定するのに用いられる．

整数，文字，論理値のスカラ型データに対するアドレス部はそれらの内部表現の値である．その値はデータと既成の一対一対応となることから，b_1 は1となる（**図3**（1））．

コンス型データ，文字列，シンボルなどの構造をもつ型に対するアドレス部はそれらの実体を指すポインタ値である．それらの実体がポインタ補正の対象となるヒープ上に作られるとき，b_1 は0となる（図3（2））．実体はセルで構成され，固定容量セルと可変容量セルとがヒープ中に混在する（図3（3））．コンス型データとクロージャの実体は2つのフィールド（2語）をもつ固定容量セルとして作られる．その容量は型コードで判別される．容量が翻訳時か実行時に決まるシンボル，文字列，環境の各実体は可変容量セルとして作られる．その容量の情報は実体中に置かれる．

ここで注目すべきことは，プログラマが使用するデータと処理系がその評価過程で使用する環境やクロージャがヒープ中に混在することである．後者は，値渡し（call-by-value）やアドレス渡し（call-by-reference）による評価方式の言語処理系ではスタックに対応するものである．こうした性格の異なるデータの混在という「論理的」問題と相異なる容量をもつデータ（セル）の混在という「物理的」問題を内在したのが FLA 処理系のくず集めである．



内部タグは処理系が使用

図 3 データ表現
Fig. 3 Data representation of FLA.

4. 評　価

4.1 理 論 値

改良型圧縮法によるくず集めは，マーキング，補正表の作成，ポインタ補正，圧縮の4つの行程からなる．**表 3** は3章で述べたそれらの時間計算量を古典的圧縮法と対比して示したものである．

表 3 改良型と古典的圧縮法の時間計算量の比較
Table 3 Comparison of time complexities of two compactifying methods.

	改良型	古典的
マーキング	$O(M_A)$	
表作成	$O(M)$	
ポインタ補正	$O(M_A-d_1\cdot N_{P_0})$	$O(M)$
圧縮	$O(M_A)$	$O(M)$

(注) M: セルの総容量（格納領域量）
M_A: 現役セルの総容量
N_{P_0}: A_0 を指すポインタの総数
d_1: 定数

4.2 実 測 値

表 4 と**表 5** は，FLA 処理系に具現された改良型圧縮法の処理時間（実測値）である．表4は，セルの格納領域量を変えて2種の完全遅延評価プログラム（P-1 と P-2）を実行したとき，ポインタ補正に要した CPU 時間を A_0 の早期検出の有無別に示したものである．

表 4 ポインタ補正時間
Table 4 Processing time for pointer adjustment.

P-1. 7 Queen（8王妃問題の7王妃版）

格納領域量		64K語	32K語	16K語
		x=0.105	x=0.212	x=0.463
		y=0.023	y=0.074	y=0.268
		GC 18回	GC 43回	GC 136回
ポインタ補正	有	1.06〈 8.55〉	2.18〈 7.29〉	6.04〈 5.86〉
A_0 の検知	無	1.30〈10.48〉	3.00〈10.0〉	10.46〈10.2〉
d_2 の値		0.84	0.78	0.73

P-2. リストの長さに基づくフィボナッチ数 (fb(17)) 計算

格納領域量		64K語	48K語
		x=0.188	x=0.259
		y=0.128	y=0.174
		GC 16回	GC 24回
ポインタ補正	有	1.06〈5.38〉	1.54〈5.03〉
A_0 の検知	無	1.84〈9.33〉	2.86〈9.35〉
d_2 の値		0.62	0.69

注（1） 左側の数値は処理時間（CPU 時間，単位は秒）を表す．
〈 〉内の数値は，処理時間÷(GC の回数×現役セル容量）で正規化された値（単位は m 秒/K 語）である．
（2） x は現役セルの占有率，y は A_0 の占有率を表す $(0\leq y\leq x<1)$.

表 5 処理時間
Table 5 Processing time.

P-1. 7 Queen (8王妃問題の7王妃版)

格納領域量	印付け	表作成	補正	圧縮	Morris 法	計	評価全体*
64K語	0.48	1.70	0.80	0.22	—	3.20	14.27
$x=0.098$	⟨6.22⟩	(2.16)	⟨10.4⟩	⟨2.85⟩	—	(4.07)	
$y=0.0095$	0.42	—	—	—	5.36	5.78	17.77
GC 18回	⟨5.45⟩	—	—	—	(6.82)	(7.35)	
32K語	1.18	2.20	1.54	0.50	—	5.22	16.65
$x=0.202$	⟨6.38⟩	(2.18)	⟨8.33⟩	⟨2.70⟩	—	(5.69)	
$y=0.070$	1.28	—	—	—	7.08	8.36	20.28
GC 28回	⟨6.92⟩	—	—	—	(7.71)	(9.41)	
16K語	3.74	2.94	3.88	1.46	—	12.0	23.16
$x=0.420$	⟨6.54⟩	(2.16)	⟨6.79⟩	⟨2.45⟩	—	(8.84)	
$y=0.225$	3.74	—	—	—	13.66	17.40	30.23
GC 83回	⟨6.54⟩	—	—	—	(10.1)	(12.8)	

P-2. リストの長さに基づくフィボナッチ数 (fb(17)) 計算

格納領域量	印付け	表作成	補正	圧縮	Morris 法	計	評価全体*
64K語	1.02	1.70	1.04	0.40	—	4.16	15.35
$x=0.186$	⟨6.98⟩	(2.16)	⟨7.10⟩	⟨2.73⟩	—	(5.29)	
$y=0.095$	0.98	—	—	—	6.08	7.06	19.00
GC 12回	⟨6.69⟩	—	—	—	(7.73)	(8.98)	
48K語	1.16	1.88	1.10	0.72	—	4.86	15.88
$x=0.237$	⟨5.86⟩	(2.25)	⟨5.56⟩	⟨3.64⟩	—	(5.82)	
$y=0.147$	1.30	—	—	—	6.70	8.00	19.88
GC 17回	⟨6.57⟩	—	—	—	(8.02)	(9.57)	
32K語	3.76	2.70	2.50	1.38	—	10.34	21.73
$x=0.429$	⟨7.44⟩	(2.29)	⟨4.95⟩	⟨2.73⟩	—	(8.77)	
$y=0.314$	3.70	—	—	—	11.72	15.42	28.32
GC 36回	⟨7.32⟩	—	—	—	(9.94)	(13.1)	

注 (1) 数値は処理時間 (CPU 時間, 単位は秒) を表す. ただし, ( ) 内の数値は, 処理時間÷(GC の回数×格納領域量) で, ⟨ ⟩内の数値は, 処理時間÷(GC の回数×現役セル容量) でそれぞれ正規化された値 (単位はm秒/K語) である.
(2) *評価時間は GC の時間 (「計」欄の数値) を含む.
(3) xは現役セルの占有率, yは A_0 の占有率を表す ($0 \leqq y \leqq x < 1$).

A_0 の検出を行わない場合, ポインタ補正の時間計算量は N_{P_0} の寄与分がなく $O(M_A)$ になる. したがって, 処理時間は(4.1)式で表すことができる.

$$T_P = a \cdot x \cdot M \qquad (4.1)$$

a : 定数
x : 現役セルの占有率 (M_A/M, $0 \leqq x < 1$)
M : 格納領域量 (セルの総容量)

(4.1)式の妥当性は, 表4の下段の正規化値が同一プログラムに対してほぼ同じ (P-1 は $a=10.2\pm0.3$, P-2 は $a=9.34\pm0.01$) であることからもわかる. P-1 と P-2 で a の値に違いがあるのは現役セル中のポインタ比率 (2.4 節参照) が異なるためである.

A_0 の早期検出を行うと, ポインタ補正の計算時間量は $O(M_A - d_1 \cdot M_{P_0})$ である. このため, 処理時間は(4.1)式からずれ, 時間が短縮される. これは, 表4の上段と表5のポインタ補正の正規化値が x の増加に伴って減少することからも明らかである. なお, 表5のポインタ補正時間は A_0 の早期検出を行ったときの実測値 (CPU 時間) である.

表4の「d_2 の値」は, その欄上の2つの正規化値から算出した(2.7)式の d_2 に相当する値である. 2.4 節で述べた H-2 の仮定が成り立つならば, d_2 は実行プログラムによらず一定になる. しかし, 同一プログラムで d_2 の値に違いがあることは H-2 の仮定が普遍的でないことを示している. しかし, y(A_0 の占有率, M_{A_0}/M) が大きくなるにつれて P-1 と P-2 の d_2 の差が小さくなるという表4の結果は, y が十分に大きいときに H-2 が成立する可能性のあることを

示している．そこで，d_2 は y が大きいときの P-1 と P-2 の平均値である 0.7 とし，表5の正規化値から改良型圧縮法によるくず集めの処理時間を表す式を求めると(4.2)式が得られる．

P-1：$(2.17+20x-7.70y)\cdot M$

P-2：$(2.23+20x-7.23y)\cdot M$　　(4.2)

表4と表5の x と y の値が同じ格納領域量で異なるのは FLA 処理系が異なるためである．表4の結果を得た処理系の改良版が表5の処理系であり，後者で実行したほうが「ごみ」の量が少ない．

4.3 改良型圧縮法 vs Morris の圧縮法

Morris の圧縮法によるくず集めを FLA 処理系に具現し，その処理時間を改良型圧縮法と対比して示したのが表5である．具現上の問題であった追加ビットは（ポインタの）アドレス部の最上位ビットで代用した．これは，

（1） タグ部を変更しない

（2） 処理速度を低下させない

の2点を保証する現実的な解決策である．

表5から明らかなように，改良型圧縮法は処理時間でも Morris の圧縮法より優位にある．特に，現役セルの占有率 (x) が小さいほど優位性は顕著である．たとえば，P-1 で $x=0.098$ のとき，改良型圧縮法は Morris の圧縮法の約55%の処理時間しか要しない．図5は，両者の正規化値をグラフ表示したものである．

Morris の圧縮法の処理時間を定量的に評価するために，表5の正規化値に古典的くず集めの処理時間を表す(4.3)式をあてはめると(4.4)式が得られる．(4.4)式はマーキングの処理時間も含む．

$$T_O=(c+b\cdot x)\cdot M \qquad b,\ c:\text{定数} \qquad (4.3)$$

$$T_M=(5.78+16.7x)\cdot M \qquad (4.4)$$

(4.2)式と(4.4)式の M に係る項を比較すると Morris の圧縮法の特徴はその定数部分が大きいことである．これは格納領域を2回走査する作業の寄与分である．改良型圧縮法はこの定数部分が 2.20（平均値）であることから，改良型圧縮法は x が小さいほど処理時間で優位になる．

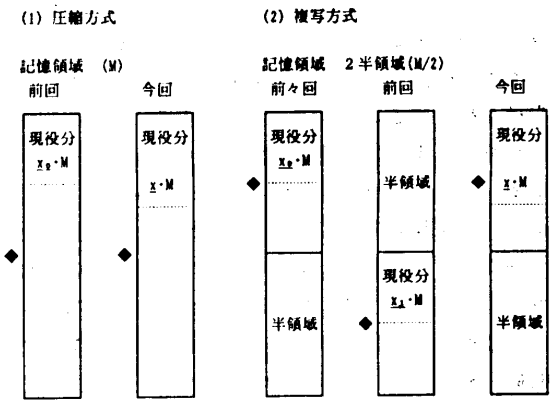
4.4 改良型圧縮法 vs 複写方式

2.4 節の結論と表5のマーキングの正規化値から，複写に必要な時間 (T_C) は(4.5)式で与えられる．

$$T_C=7.88\cdot x\cdot M \qquad (7.88\fallingdotseq 1.2\times 6.57) \qquad (4.5)$$

x：現役セルの占有率 (M_A/M, $0\leqq x<1$)

6.57 はマーキング処理の正規化値の平均．



◆は使用対象の領域を指す．
記号式は該当する領域の領域量を表す．

図4 圧縮方式と複写方式の作業比較

Fig. 4 Comparison of process between compactifying and copying methods.

改良型圧縮法と複写方式の両者のくず集めの処理時間を比較する場合，セルの格納に用いる領域量（厳密に言うならば，回収により使用できた領域量）は同じにするのが原則である．格納領域量を M とすると，改良型圧縮法のくず集めは M がすべて使用し尽くされたときに引用される．複写方式では $M/2$ の半領域が交互に使用されるため，くず集めは $M/2$ が使用し尽くされたときに引用される．そこで，前者が M の格納領域で $x\cdot M$ の現役セルを圧縮する作業は，後者が2回呼ばれ，もとの半領域に $x\cdot M$ の現役セルを複写する作業に相当する（図4参照）．ただし，使用できた領域量は(4.6)式に示すように違いがある．

圧縮法：$(1-x_0)\cdot M$

複写方式：$(1-x_0-x_1)\cdot M$　　(4.6)

x_0：前回圧縮後，または前々回複写後の現役セル容量比

x_1：前回複写後の現役セル容量比

総容量が $x_1\cdot M$（前回）と $x\cdot M$（今回）の現役セルを複写するのに必要な時間は(4.5)式の x を単に x_1+x で置き換えることで得られる．しかし，(4.6)式で示されるように改良型圧縮法と複写方式のそれぞれのくず集めでは回収により使用できた領域量が異なる．同じ領域量で両者の処理時間を比較するためには，(4.6)式に基づいた補正が必要である．その補正結果は(4.7)式で示される．

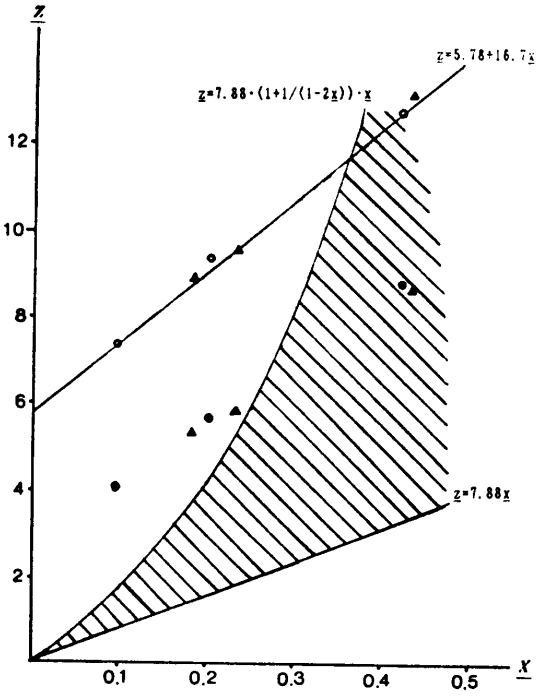
$$T_C=7.88\cdot(x_1+x)\cdot M\cdot(1-x_0)/(1-x_0-x_1) \qquad (4.7)$$

(4.7)式は複写方式によるくず集めの処理時間を表す．

式中の x_0 と x_1 は実行プログラムと M とに依存して決まる値である．しかし，くず集めの回数が多く，しかもその何回目であるかを特定しなければ，それらの関係を推定することができる．くず集めごとに現役セルが増える場合は，明らかに $0<x_0<x_1<x<1/2$ である．逆に，現役セルが減る場合は $x_0>x_1>x$ であるが，くず集めで常に現役セルが減少していくことは一般的なプログラムの実行ではまずありえない．現役セルの減少があれば，それに見合うかたちで増加があったと考えるべきである．こうした場合，くず集めの回数が多ければ $x_0 \sim x_1 \sim x$ となる．以上のことから，Tc に関する(4.8)式が得られる．

$$7.88x \cdot M \leqq Tc \leqq 7.88 \cdot (1+1/(1-2x)) \cdot x \cdot M \tag{4.8}$$

なお，上限は $x_0 \leqq x_1 \leqq x$ の関係式，下限は $0 \leqq x_0 \leqq x_1$ の関係式から得られる．(4.8)式は複写方式によるくず集めの処理時間（推定値）を表す．この式の値を改良型圧縮法の正規化値と比較すれば，両者の処理時間に関する評価ができる．**図5**には Tc の取り得る範囲が図示されている．これから言えることは，改良型圧縮方式のくず集めの処理時間が複写方式のそれの2倍以内になるのは，上限に対して $x>0.12$ である．くず集めが頻繁になれば，$x_0 \leqq x_1 \leqq x$ でかつ x が大きくなる．こうした場合，Tc はその上限近くの値を取るので，改良型圧縮法の処理時間が複写方式のそれの2倍以内になることは十分保証される．そこで，従来から指摘されていた圧縮方式の処理時間に関する欠点は解消されると考える．



注(1) 斜線部は複写方式の正規化処理時間（理論値）の範囲を示す．
(2) 記号（実測値）の意味
●: P-1 改良型圧縮法
○ Morris の圧縮法
▲: P-2 改良型圧縮法
△ Morris の圧縮法
(3) x は現役セルの占有率 (M_A/M) を表す．
(4) 正規化処理時間（z）の単位はm秒/K語である．

図5 正規化処理時間
Fig. 5 Normalized processing time.

圧縮方式は，その使用領域量の利点のほかにも，現役セルの配置に関する局所性を保存するという利点がある．これは，データの割り付け (allocation) に関する局所性が重要視される仮想記憶やキャシュ記憶での使用に大きな効力をもつことになる．

5. まとめ

本論文が提示した2つの技法，不動塊（A_0）を指すポインタの早期検知と格納領域の二分走査はその原理自体決して目新しいものではない．しかし，これらの技法はポインタ補正の高速化に大きく寄与した．また，前者の技法は A_0 に対応する補正表の一部の構築を不要にし，記憶領域の節約にも寄与した．FLA（完全遅延評価系）に具現された改良型圧縮法のくず集めの処理時間の実測値から，可変容量セルに対するくず集めとして，改良型圧縮方式がソフトウェアインプリメンテーションでも十分実用に耐え得ることを示した．

くず集めを含む FLA 処理系は比較的小さく，しかもキャシュメモリ等の高速記憶のない Sun 3/50 上で動作しているため，主記憶上のセル配置に関する局所性がその処理時間にほとんど反映されていない．こうしたことから，高速記憶と大容量の仮想記憶を有する計算機で大規模な処理系を動作させたときの処理時間はさらに向上するものと考える．

圧縮方式自体の改良としては，ビットカウント用命令語を有する計算機で，マークビットをフィールドから分離し単独の表にすることが考えられる．その場合，ポインタ補正と圧縮の処理時間がさらに短縮できる．

圧縮方式を可変容量セルのくず集めとして定着させるには，その処理時間を複写方式に近づけることが必要である．それが可能か否かを含めて，今後の検討課

題である.

謝辞 有益なコメントを頂いた査読者に感謝します.

参考文献

1) Cohen, J.: Garbage Collection of Linked Data Structures, *ACM Comput. Surv.*, Vol. 13, No. 3, pp. 341-367 (1981). (寺島訳: つなぎのあるデータ構造のくず集め, コンピュータサイエンス '81 (ビット別冊), 共立出版 (1983).)
2) Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, *Inf. Process. Lett.*, Vol. 7, No. 1, pp. 27-32 (1978).
3) Morris, F. L.: A Time- and Space-Efficient Garbage Compaction Algorithm, *Comm. ACM*, Vol. 21, No. 8, pp. 662-665 (1978).
4) Jonkers, H. B. M.: A Fast Garbage Compaction Algorithm, *Inf. Process. Lett.*, Vol. 9, No. 1, pp. 26-30 (1979).
5) Knuth, D.E.: *The Art of Computer Programming*, Vol. 1; *Fundamental Algorithms* (2nd ed.), Addison-Wesley, Reading, Mass. (1973). (米田ほか訳: 基本算法/情報構造, サイエンス社 (1980, 1981).)
6) Henderson, P.: *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, N. J. (1980).
7) Henderson, P. et al.: A Lazy Evaluator, *Proc. of 3rd Symp. on Principle of Programming Languages*, pp. 95-103 (1976).
8) Takeichi, M.: Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs, *New Generation Computing*, Vol. 5, No. 4, pp. 377-393 (1988).

(昭和63年11月18日受付)
(平成元年6月13日採録)

寺島 元章 (正会員)

昭和23年生. 昭和48年東京大学理学部物理学科卒業. 昭和50年同大学院修士課程, 同53年博士課程修了. 理学博士. 昭和53年より電気通信大学計算機科学科助手として勤務して現在に至る. 記号処理, プログラミング言語, 人工知能などに興味をもつ. ACM, AAAI 各会員.

佐藤 和美 (正会員)

昭和39年生. 昭和62年電気通信大学計算機科学科卒業. 平成元年同大学院修士課程修了. 工学修士. 現在, ソニー(株)スーパーマイクロ事業本部勤務, ワークステーションのソフトウェア開発に従事. 関数型言語とその処理系の作成に興味をもつ.