ソフトウェア基礎論 31 － 3
（１９８９．９．２０）

関数型プログラムにおける不要セル検出の最適化実現方法

井上克郎　　鳥居宏次
大阪大学基礎工学部情報工学科

筆者らは、関数型言語プログラムのソースプログラムを静的に解析して回収可能な不要セルの発生を検出し、実行時にそのセルを直ちに回収する方法を提案している(3)(4)。この方法の効果を確かめるために、実際にこの方法を実現したＬＩＳＰインタプリタを作成し、その上で６種類の例プログラムを実行してみた。その結果、３つについては発生する不要セルの全てをこの方法で回収できた。他の３つは、９９％、２４％、３％の不要セルを回収できた。更にこのうちの２つのプログラムについては、その関数定義を手で解析、変更することによって、回収率を２４％から１００％に、３％から５７％に向上させることができた。

An Implementation of Immediate Reclamation Method for Garbage Cells

Katsuro Inoue　　Koji Torii

Department of Information and Computer Sciences
Faculty of Engineering Science, Osaka University
Toyonaka, Osaka 560, Japan
inoue@osaka-u.ac.jp

We have proposed a method to predict generation of garbage cells, by analyzing a source text of functional programming languages(3)(4). The garbage cells whose generation was able to be predicted are reclaimed immediately without any overhead at the execution time. To investigate the effects of this reclamation method, an experimental LISP interpreter was implemented, and several sample programs were executed. We knew that for most programs, many of the garbage cells are predicted and reclaimed by this method. Programming techniques to improve the reclamability are also studied.

1．まえがき

最近関数型言語に関する感心が高まり、その処理系について種々の研究が行なわれている。関数型プログラムの実行効率を上げるための最適化の一つとして、実行時の不要セル（Garbage Cell）の発生を静的に予想し、実行時にそれを即時回収する方法が提案されている[3],[4],[5]。（不要セルとは、いわゆるヒープ領域中のあるメモリー領域のうち、ある実行時点以降使用されなくなるものをいう）。しかしその方法の実際の実現方法や効率についての報告はない。本論文では、文献(3),(4)の方法に基づいた不要セルの発生予測とその回収方法（ここではこれらを即時回収方式と呼ぶ）を実際に実現したLISPインタプリタについて述べる。

システムの実現手法としては、一般的なLISPの実現手法[7]を用いた。データ構造としては二分木リストのみを用い、即時回収方法に関するデータを集めるのに必要な基本関数のみを実現した。試作したシステム上でいくつかの例プログラムを実行し、使用セル数やゴミ集め（Garbage Collection）、実行時間などのデータを収集した。また、ソースプログラムを一部変更することによって即時回収できる不要セル数を向上させる方法についても考察した。

提案された即時回収方式は、副作用を含まない関数で、その引数がリストのものにしか適用できない。従って本システムでは、各定義関数が副作用を起こさないか、また、その引数がリスト型か調べるルーチンを設け、それらを満たすものだけを即時回収の対象とした。

２分木リスト中の各セルのうち、根のセル及び根からcdr部のポインタのみをたどって到達できるセルを背骨セルと呼ぶ（図１）。本システムでは、背骨セルのなかで不要なものの即時回収を行なう。以下２章では即時回収方式の概要、３章では本システムの概要、４章では例プログラムの実行結果、５章では回収率向上のための技法について述べる。

2．　即時回収方式の概要

2．1　非継承セルと生成セル

今 ...(F... x ...)... という部分式について考える。ここで、xはリスト型変数でFの第i引数、またFの値（出力値）もリストとする。

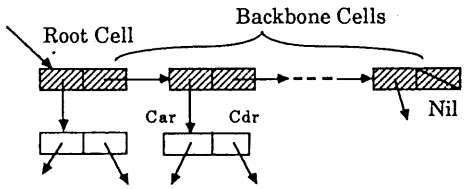

図１　２分木リスト中の背骨セル

Fig.1 Backbone cells in a binary list.

［定義１］　x中に含まれるセルのうち、Fの出力として現われないものを、Fの第i引数の非継承セル（Non-Inherited Cell）と呼ぶ。

［例１］　いまFがcarでxが図１のリストとすると、斜線部の背骨セルが非継承セルになる。

一般に非継承セルが直ちに不要セルとはならない。なぜならFの非継承セルであっても、x自身またはその一部が、式の他の部分の計算に用いられているかも知れないからである。

［例２］　(cons(car x) x)において、xの背骨セルはcarに対して非継承であるがconsの値の計算に用いられるため、不要セルではない。

［定義２］　Fの値のリスト中に存在するセルで、Fのどの引数中にも現われないセルをFの生成セル（Created Cell）と呼ぶ。

［例３］　(cons x1 x2)に対して、図２のように根のセルが生成セルになる。

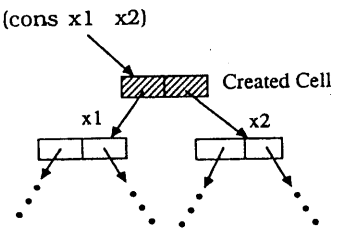

図２　生成セルの例

Fig.2 An example of created cell.

［性質１］　ある部分式　...(G ...(H ...)...)...に対し、関数Hの生成セルでかつGの非継承セルはGの計算終了後直ちに不要セルとなる。

［例４］　(car(cons　x1　x2))に対して、consの生成セルは不要セルになる。

これは、consの生成セルは、carの引数としてのみ使われ、他の部分で用いられないからである。（ここでは、共通な部分式の計算は一度求めた値を保存、再利用するのではなく、そのつど繰り返して行うことを仮定している。）　このような不要セルの出現は一般に動的にしか知ることができないが、一部静的に検出しうる場合もある。以下ではそのような静的な検出方法について議論する。

［定義３］　関数のある引数に含まれる各背骨セルが、入力データに依存せずどのような場合でも非継承セルであるとき、この引数を非継承引数と呼ぶ。

［例５］　carの第１引数は非継承引数である。cdrの第１引数、consの第１、第２引数は非継承引数でない。

［定義４］　ある関数値の背骨セルが入力データに依存せずどのような場合でも生成セルであるとき、この関数を生成関数と呼ぶ。

［例６］　関数　f(x)≡　(cons　x　nil)はただ１つの背骨セルを作りそれは常に生成セルなのでfは生成関数である。cons、car、cdrは生成関数でない。

［性質２］　ある部分式　...(G ...(H ...)...)...において、関数HはGの第i引数に位置するとする。いまHが生成関数でかつGの第i引数が非継承引数ならば、Hの値の背骨セルはGの計算終了後直ちに不要セルとなる。

［例７］　関数fを　f(x)≡　(cons　x　nil)とする。式(car(f　x))に対し、fは生成関数、carの第１引数は非継承引数なので、carの値の求った後にfの値の背骨セルは不要セルである。

ここでは以下で述べる非継承引数や生成関数を検出する方法、及び不要セルの回収方法を含めて即時回収法と呼ぶ。この方法で全て不要セルの検出をできるわけではないが、実用的に有効であることが４章で示される。

２．２　非継承引数の検出

プログラム中の各定義関数に対し、以下の操作を行う。そして、ステップ⑤で最終的に得られた解の値が真である変数に対応する引数が、非継承引数となる。ここでは文献(3)で提案された一般的な方法を、背骨セル専用に最適化した。後ろで例を用いた各ステップの説明を行うが、詳細な証明等は紙面の関係で省略する。

［ステップ①］　各リスト型引数に対してブール型の変数をそれぞれ設ける。ここでは便宜上、各関数名に引数の位置を示す数を付けたものを変数とする。また、car1=真、cdr1=偽、をブール型定数とする。

［ステップ②］　次に各定義本体において、リスト型変数にかかりうる、if－then－else以外のリスト型関数の系列を全て求める（空系列εも含む）。ただし、リスト型以外の値を持つ関数を含む関数列は求めない。

［ステップ③］　得られた系列を、引数の現れるところから順次右から左へ走査し、現われる各関数を対応するブール型変数や定数に置換える。そして次のいずれかの場合走査を停止する。

1. いま見ている関数記号が左端のものであるとき。
2. 次に見るべき左隣の関数がconsのとき。
3. いま見ている関数が定義関数で、その定義内で、consを着目する引数に施しうるか、又はそれを施しうる別の定義関数を呼びうる場合。

そして得られた各変数、定数の間を論理和記号｜で結び、一つのブール式にする。空系列εに対しては定数、偽を式の値とする。（このブール式は、その値が真のとき引数の背骨セルが関数値に現れうることを表す。）

［ステップ④］　一つの定義関数の一つのリスト型変数に対する各関数列から変換したブール式をそれぞれを論理和＆で結び、ブール型方程式を作る。そして、各定義関数の各変数に対して得られた方程式をまとめて一つの連立方程式とする。

［ステップ⑤］　得られた方程式の、偽＜真　に関する最大解を求める。（解が真である変数に対応する関数の引数は非継承引数。）

クイックソートの例

図３のクイックソートのプログラムについて考える。（ステップ①の例）qsの第１引数に対しqs1、highの第１引数にhigh1、lowの第１引数にlow1、appendの第１引数、第２引数にそれぞれappend1, append2を設ける。high、lowの第２引数はlist型ではないことがわかるの

```
(de qs (x)
  (cond((null x) nil)
    (t  (append(qs(low (cdr x)(car x)))
               (cons (car x)(qs(high (cdr x)(car x]
(de high (x i)
  (cond((null x) nil)
       ((not (lessp (car x) i))
              (cons (car x)(high (cdr x) i)))
       (t (high (cdr x) i]
(de low (x i)
   (cond ((null x) nil)
     ((lessp (car x) i)(cons (car x)(low (cdr x) i)))
     (t (low (cdr x) i]
(de append (x y)
  (cond  ((atom x) y)
     (t   (cons (car x)(append (cdr x) y]
```

図３　クイックソートプログラム

Fig.3　Quicksort program.

で変数を設けない。一連の操作が終了した時点で、この変数の値が真ならば非継承引数、偽ならばそうでないことを示す。

（ステップ②の例）　引数のリストがどの様な関数が施されて最終的な関数値のリストとなるかを後のステップで調べるために関数列を求める。例えば、qsの第１引数xに対して、

(append (qs (low (cdr x) ...)) ...)

(append ... (cons (car x) ...))

(append ... (cons ... (qs (high (cdr x) ...))))

が得られる。(null x)や

(append (qs (low ... (car x)))

(append ... (cons ... (qs(high ...(car x)))))

もxに対する関数列であるが、途中の値がブール値や整数になるため、ここでは除外する（(car x)の値は整数）。他の定義関数についても同様。

（ステップ③の例）　例えば上記の関数列 (append (qs (low (cdr x) ...)) ...)に対し、まず、cdrがcdr1（＝偽）に置き換えられ、次にlowがlow1に置き換えられる。そして、lowは、その定義にconsを持つのでそこで走査をやめる。結局ブール式として、 low1 | cdr1 が得られる。他の関数列についても同様。

この関数列の例では始めxにcdrが施される。このときxの各背骨セルは、根を除きcdrの値になる。すなわちcdrはその引数の背骨セルを出力に出す。（ブール式中の項cdr1はこれを表す。）

次にlowが施されるが、その第１引数が非継承であればcdrの値の背骨セル（すなわちxの背骨セル）はlowの出力とはなりえない。一方もし非継承でないなら出力になりうる。（low1がそれを表す。）後者の場合は、後に続くqsとappendに依存してxの背骨セルがappendの出力に現われるかどうかが決まる。この判定は、lowの中に含まれるconsと、appendに含まれるcarやcdrが打ち消し合う場合があり、その判定は容易ではない。ここでは簡単化のためそれらの引数のセルがそのまま出力に現れうると（安全側に）考え、走査を停止する。すなわちlowの第１引数が非継承でないならappendの出力、すなわちqsの値としてxの背骨セルが現れうると考える。

（ステップ④の例）　qsの第１引数xに対しは、以下のような式が得られる。

qs1 = (low1 | cdr1) & (car1) & (high1 | cdr1)

（car1=真、cdr1＝偽。）ここで、&で結ぶのは各関数列に対応したブール式全てが真のとき、qs全体としてxの背骨セルを関数値に含まないことを意味する。全体としては次のような連立方程式が得られる。（簡単な式の簡略化を行なっている。）

```
qs1 = low1 & high1
high1 = high1
low1 = low1
append1 = append1
append2 = append2
```

（ステップ⑤の例）　この方程式を満たす解の組は一般に複数個存在し、何れの場合も真の値を持つ変数に対応する引数は非継承引数である。我々はできる限り多くの非継承引数を発見したいので、ここでは最大解を求める。この方程式の最大解はqs1、high1、low1、append1が真，append2が偽になる。すなわち、qs、high、low、appendの各第１引数が非継承引数である。

２．３　生成関数を求める式

プログラム中の各定義関数に対し、以下の①から⑥の操作を行なう。

［ステップ①］非継承引数のステップ①と同じ。ただし、ブール型定数としてcons1＝真、cons2＝偽を設ける。

［ステップ②］非継承引数のステップ②と同じ。

［ステップ③］得られた系列を、左端の関数記号から

右へ順次走査し、現われる関数をブール型変数や定数と置き換える。そして次のいずれかの場合走査を停止する。

1. 次に見るべき右隣の記号がcarかcdr、又は右端の変数記号。
2. いま見ている関数が定義関数で、その定義内でcarかcdrを着目している引数に施しううるか、又はそれらを施しうる別の定義関数を呼びうる場合。
3. 次に見るべき右隣の関数が重複セルを生成する可能性のある関数。

そして得られた各変数、定数間を論理和記号｜で結び、一つのブール式にする。空系列εに対しては定数、偽を式の値とする。

ここで重複セルとは、リストの根から一つ以上のcar部のポインタをたどって到達できる背骨セルのことをいう。一般にＬＩＳＰプログラム実行中には、ほとんど重複セルは発生しないことが知られている[9]。ここでは文献(3)で示された条件を用いて重複セルを発生しうる関数を見つける。例えば (de f (x) (cons x x)) のように一つの引数に対しconsの第１、第２引数を同時に施しうる関数はこの条件に含まれる。クイックソートプログラムにおいて、重複セルを生成しうる関数は、存在しない。

［ステップ④］非継承引数のステップ④と同じ。

［ステップ⑤］非継承引数のステップ⑤と同じ。

［ステップ⑥］各定義関数が持つそれぞれのリスト型引数に対応するブール型変数の値の論理和をとり、その結果が真の定義関数が生成関数になる。

クイックソートプログラムの例

得られる連立方程式は、

qs1 = append1 & (append2 | qs1)

high1 = high1

low1 = low1

append1 = append1

append2 = ε　(= 偽)

となり、その最大解はqs1、high1、low1、append1が真、append2が偽である。qs、high、lowは、それぞれ唯一つのブール型変数を持ちそれらが真であるため、生成関数であることがわかる。appendはappend1、append2二つの変数を持ち、append2が真ではないため、生成関数ではない。

3．試作システム

3．1　主な特徴

前章で述べた方法を実際に実現したLISPシステムをNEWS-801ワークステーション上に作成した。表１に作成した組み込み関数を示す。次章で述べるように即時回収方式のための、基本関数rclm0、rclm1、...、rclm31、hold、及びoptがある。一方通常よく用いられる不要セル回収プロセス（Ordinary Garbage Collection、OGCと呼ぶ）も作成した。これは利用可能なセルが自由リスト中になくなるかまたは基本関数gcによって起動される。まず、①現在使用中のセルを順次たどり印をつけ、②ヒープ領域を走査して印がないセルを自由リストにつなぎ（回収する）、③再度走査して各セルの印を消す。

3．2　ｒｃｌｍ関数とｈｏｌｄ関数

関数holdはシステム中に設けられた特別なスタックにその引数を保存する（このスタックは他の用途のものと兼用できるがここでは別に設けた）。引数がリスト型のときは、そのリストの根をさすポインタが保存

表１　作成した組み込み関数

判定	atom eq equal null < >
リスト操作	car cdr cons list length
演算	and not or + - * /
入出力	close closeall dirout load open print read terpri
制御	cond do
プロパティリスト	get proplist putprop remprop
即時回収	hold rclm0 ... rclm31 opt
その他	eval de gc getd quote quit set setq

される。関数rclm1は、そのスタック最上の要素をポップし、それがリストを指すポインタであれば、そのリストの背骨セルを自由リストに戻す（回収する）。hold、rclm1ともに引数を１つ取り、その関数値は引数そのままである。

いま部分式 ...(F (G ...))... において、Fの第1引数が非継承引数、Gが生成関数ならば、Fの関数値が求まった後にはGの関数値の背骨セルは不要である。そこでrclm1-holdの組を ...(rclm1 (F (hold (G ...)))... のように挿入して不要な背骨セルを回収する。rclm2はリストのcar部でたどれる各セルを回収し、rclm3はそれらのセルと各背骨セルの両方を回収する。同様にrclm31まで種々のパターンで回収する関数を用意した。rclm0はスタック最上要素をポップスするだけで何も回収しない。

rclm-holdの組は直接LISPのソースプログラムに手で入れることもできるが、rclm1-holdの組については次に述べるように関数optの実行によって自動的にその定義に挿入できる。

3．3　rclm1－hold組を挿入する関数

関数optは定義関数名を引数として取り（この定義関数を主関数と呼ぶ）、2．2、2．3で述べた各ステップを含む以下のことを順に行なう。

(1)主関数から直接及び間接的に呼びうる定義関数名の表を作成する（主関数もこの表に含まれる）。

(2)表中の各関数から副作用を起こしうるものを捜しそれらを表から除く。

(3)表中の各定義関数の引数のうち、リスト型でないものを捜す。その引数の型が明らかに非リスト型と分かっている基本関数(plus、times、greaterp等）をもとに順次各関数の引数の型を求める。

(4)求めたリスト型の各引数に応じた変数に対し、かかりうる関数系列（condを除く）を2.2や2.3で述べたように作成する。

(5)表中の関数がcarまたはcdr、及びconsを施しうるかを調べる。

(6)上述の(4)、(5)から非継承引数及び生成関数のためのブール式を作成する。

(7)得られた連立式の最大解を求める。この手間は定義関数の個数nに比例する（各式の長さや一つの定義関数の引数の個数はnとは独立で、ある定数以下であると仮定する）。

(8)求った非継承引数、生成関数からrclm1-holdの組を挿入できる部分式を表中の各定義関数の定義中から捜し、その定義を挿入したものに変更する。

```
(de qs (x)
  (cond  ((null x) nil)
     (t (rclm1(append(hold
          (rclm1(qs(hold(low (cdr x)(car x))))))
          (cons (car x)
            (rclm1(qs(hold(high (cdr x)(car x]
```

図４　rclm1-holdを有する関数ｑｓの定義

Fig. 4 Definition of qs with rclm1 and hold.

4．例プログラムの実行

次に示すプログラムを作り、作成したシステム上で実行した。各プログラムは、rclm1-holdを挿入していないものとoptを実行して自動的に挿入したものをそれぞれ実行し、OGCの起動回数、OGCに要した時間、総実行時間等を比較してみた。

(a)　クィックソートプログラム（qs）（４定義関数５行）：図３に示すプログラムで、入力は１０００個の整数とする。表２にヒープ領域の大きさが５０００セルの時、表３に１００００セルの時の実行結果を示す。このプログラムは、主関数qsにoptを施すことによってqsの定義中に図４のように３組のrclm1-holdが挿入される（他の定義は変わらない）。表から分かるように即時回収によって全不要セルが回収される。表２の値で、即時回収しないときにOGCに要する時間13.6秒は、総時間の差56.1-42.6=13.6秒と等しいことがわかる。また表３に示すように、ヒープ領域を大きくすることによって、即時回収しなくてもOGC時間が小さくなることがわかる。この場合、余分な関数実行の手間等のために総実行時間が、即時回収しないものより大きくなっている。

(b)　n個のnilから成るリストの２分（div2）（４定義関数、１７行）：始めn個のnilを背骨セルに持つリストを作り次にそれをもとに長さを半分にしたリストを作る。これを１２００回繰り返す[10]。このプログラムには４組のrclm1-holdが挿入できる。表４、５にそ

表２　ｑｓの実行結果（５０００セル）

データ：整数１０００個

1	2	3	4	5	6	7	8	9
N	5000	17144	0	-	15	13.6	56.1	-
Y	5000	17144	17144	100%	0	0.0	42.5	0.1

1 即時回収するか否か

2 ヒープ領域中の全セル数

3 発生した全不要セル数

4 即時回収によって回収できたセル数

5 回収率（3÷4）

6 OGCが起動された回数

7 全OGCに要した時間

8 総実行時間（7も含む）

9 optに要した時間

表３　ｑｓの実行結果（１００００セル）

データ：整数１０００個

1	2	3	4	5	6	7	8	9
N	10000	17144	0	-	2	1.0	43.1	-
Y	10000	17144	17144	100%	0	0.0	44.0	0.1

れぞれヒープ領域が５００セル、５０００セルの実行結果を示す。表４では、即時回収しなければOGCが繰り返し起動されそのために総実行時間が大きくなるが、即時回収することによってOGCの手間が不要となり総実行時間が大幅に小さくなることを示している。また、表５に示すように、この差はqsと同様ヒープ領域を増すことによって小さくなる。

(c) フルリバース関数（fullr）（３定義関数１１行）：リストの各階層中の各要素を逆順にする[11]。表６にデータとして５００個のnilを（nil ... nil）の形のリストで与えられた場合、表７に（（(... (nil) nil) ...) nil）の形のリストで与えた場合を示す。不要セルはいずれの場合も即時回収によって全て回収される。

表４　ｄｉｖ２の実行結果（５００セル）

データ：長さ２００のリスト

1	2	3	4	5	6	7	8	9
N	500	120206	0	-	599	113.8	256.5	-
Y	500	120206	120200	99%	0	0.0	133.9	0.1

表５　ｄｉｖ２の実行結果（５０００セル）

データ：長さ２００のリスト

1	2	3	4	5	6	7	8	9
N	5000	120206	0	-	25	8.0	141.0	-
Y	5000	120206	120200	99%	0	0.0	133.4	0.1

表６　ｆｕｌｌｒの実行結果

データ：(nil nil ... nil)の形の５００nilのリスト

1	2	3	4	5	6	7	8	9
N	5000	125250	0	-	31	57.2	196.9	-
Y	5000	125250	125250	100%	0	0.0	139.6	0.1

表７　ｆｕｌｌｒの実行結果

データ：((...(nil)nil)...)nil)の形の５００nilのリスト

1	2	3	4	5	6	7	8	9
N	5000	499	0	-	0	0.0	2.9	-
Y	5000	499	499	100%	0	0.0	3.1	0.1

表８　ｐｒｉｍｅの実行結果

データ：２００以下の素数発生

1	2	3	4	5	6	7	8	9
N	1000	777	0	-	1	0.1	29.1	-
Y	1000	777	777	100%	0	0.0	28.9	0.3

(d) 素数発生（prime）（８定義関数２８行）：エラトステネスのふるい方によって素数の系列を作る。このプログラムには３組のrclm1-holdが入る。表８にヒープ領域が１０００セルの結果を示す。即時回収によって全て不要セルが回収されることがわかる。しかし、

表９　ｉｎｔｅｒｐの実行結果

データ：　ｔａｒａｉ(７,５,３)

1	2	3	4	5	6	7	8	9
N	1000	3128	0	-	6	1.2	18.8	-
Y	1000	3128	106	3%	6	1.4	19.1	0.3

表１０　ｒｃｌｍ２４を挿入したｉｎｔｅｒｐの実行結果

データ：　ｔａｒａｉ(７,５,３)

1	2	3	4	5	6	7	8	9
Y	1000	3128	1811	57%	2	0.5	17.5	0.3

表１１　Ｇｏの実行結果

データ：　黒白の手順２４手

1	2	3	4	5	6	7	8	9
N	3000	23144	0	-	19	2.8	265.2	-
Y	3000	23144	5757	24%	17	2.6	265.2	1.6

表１２　複写関数を有するＧｏの実行結果

データ：　黒白の手順２４手

1	2	3	4	5	6	7	8	9
N	3000	24764	0	-	22	3.3	268.1	-
Y	3000	24764	24764	100%	0	0.0	263.2	1.7

即時回収しなくてもOGCは１回しか起動されず、総実行時間の差は小さい。

(e)　LISPインタプリタ（inter）（７定義関数３９行）：文献[12]に示されるLISPの実行系の定義。１組のrclm1-holdが挿入できる。この上で長さ７、５、３のリストをデータとするタライ関数[10],[13]を実行する。表９に示すように約３％のセルしか即時回収出来なかった。そこで手でこのプログラムを解析すると[3]、さらにrclm24-holdが一つ挿入できることがわかった。rclm24は、根のセルと、根のセルのcar部が指すセルを回収する。これを挿入することによって表１０に示すように５７％の不要セルが回収できるようになった。

このようにプログラムによっては、背骨セル以外の不要セル回収が重要になる場合がある。

(f)　碁盤プログラム（Go）（４３定義関数１６６行）：二人のプレイヤーが入力する打ち手に従って、盤面を更新し、勝者を判定するプログラム[14]。５×５の盤面で２４手まで入力した。この比較的大きなプログラムに対してoptの実行時間は１．７秒しか要さなかった。これによって、１４組のrclm1-holdが自動的に入った。このプログラムは、データ構造として背骨セルのみから成るリストしか用いないため多くの不要セルが即時回収できることが期待されたが、表１１に示すように２４％の不要セルしか即時回収できなかった。そこでプログラムを解析すると、図５に示す差集合を求めるsetdiffのために多くの不要セルが即時回収できないことがわかった。このsetdiffの定義では、c2がnilのときにc1が直接出力値として出てしまうために非継承引数や生成関数のための条件が満たさなくなってしまう。そこで図６に示すようにそのまま出力値として出さずにc1を複写したリストを出すように変更すると、４組のrclm1-holdが更に挿入できるようになり表１２に示すように１００％回収できた。しかしこの変

```
(de setdiff (c1 c2) (cond
      ((null c2)    c1)
     (t     (setdiff (remv c1 (car c2)) (cdr c2]
(de remv (l e)(cond
      ((null l)  nil)
      ((equal (car l) e) (remv (cdr l) e))
      (t  (cons (car l) (remv (cdr l) e]
```

図５　差集合を求める関数setdiffの定義

Fig. 5 Definition of setdiff.

```
(de setdiff (c1 c2) (cond
      ((null c2) (copy c1))
      (t     (setdiff (remv c1 (car c2)) (cdr c2]
(de remv (l e)(cond
      ((null l)  nil)
      ((equal (car l) e) (remv (cdr l) e))
      (t  (cons (car l) (remv (cdr l) e]
(de copy (l) (cond
            ((null l) nil)
            (t(cons (car l) (copy (cdr l]
```

図６　複写関数を有するsetdiffの定義

Fig.6 Definition of setdiff with copy.

更は複写用関数copyを使うため、全体のセル使用量は増す。

5．即時回収向上のための手法

5．1　複写関数の利用

前章のGoのプログラムでの例のように、引数の値をそのまま複写して出力に出す関数を用いることによって非継承引数でないものを非継承引数に、生成関数でないものを生成関数にすることができる。しかし、複写関数を用いると一般に消費されるセルの数が増し、また総実行時間が大きくなる場合がある。従って、即時回収による効率向上が顕著な場合のみ用いるべきであろう。

5．2　補助関数の導入

いま ...(g (f exp const))...という式があったとする。ここでexpはある式、constは変数を含まない定数式とする。またgの第1引数は非継承引数、fが生成関数か調べる式がf1=真、f2=偽であったとする。今までの方法ではf2が偽であるためfが生成関数とはならずこの部分式にはrclm1-holdを挿入することはできない。

そこで、f'を以下のように定義する。

(de f' (x_1) (f x_1 const))

するともとの式はf'を使って、...(g (f' exp))...というふうに置き換えられる。ここで生成関数に対する方程式の解は、f'＝真となるのでrclm1-holdを挿入できるようになる。これは、今までの方法が引数として定数しか持たないものに対しても、変数を含む式と同様の条件を適用していたためで、関数optを改良することによって、このような補助関数を導入しないでも、挿入できるようになろう。

6．　むすび

本システムは、まず、基本となるLISPシステムを作成し、次にrclmとholdを基本関数として追加、最後にoptを追加して開発した。基本システムとrclm、holdの実現に一人で約３カ月、optの追加に約１カ月半を要した。基本システムは約５０の基本関数を持ち、言語cで約３２００行である。そのなかで関数optの部分はcで１２００行である。

本システムで、自動的に即時回収するのは、背骨セルの一部である。リスト中の他の部分の不要セルは自動的に回収しない。他の部分についても同様な方程式を立てて、解くことによって、それが即時回収できるか否かを判定できる[3]。それぞれを解くには、４章で示したようにわずかな時間でできるので、他の部分の回収ルーチンをシステムに追加すれば、より実用性が増すであろう。

本システムでは即時回収方式をLISPインタプリタ上で実現したが、LISPコンパイラや他の関数型言語システム上に実現することも可能である。特に、コンパイラシステムで実現する場合は、種々の最適化の一つとして、即時回収できるか否かの判定を行い、その結果をrclmやhold等の関数を用いずに直接目的コードに組み込むことが出来よう。

謝辞　本システムの作成にご協力いただきました川瀬淳氏（現住友電気工業㈱）に感謝します。

文献

(1) J.Cohen :" Garbage collection of linked data structure", Computing Surveys, **13**, 3, pp. 341-367 (Sep. 1981).

(2) J.Barth :" Shifting garbage collection overhead to compile time", Comm. ACM. **20**, 7, pp. 512-518(July 1977).

(3) K.Inoue, H.Seki and H.Yagi:"Analysis of functional programs to detect run-time garbage cells", ACM Trans. of Programming Languages and Systems, **10**, 4, pp.555-578 (Oct. 1988).

(4) 井上、関、八木：”関数型言語で用いられるリストの効率的実現法について”、信学技報，**ＡＬ８５－２６**（１９８５－０７）．

(5) 太田、吉田、福村：”ＬＩＳＰ関数によって生成されるリスト構造の解析”、信学論（Ｄ）1、**Ｊ６９**、６、pp. ８７８－８８５（昭６１－０６）．

(6) 井上、川瀬、鳥居：”ガーベジセルの直接回収を行なうＬＩＳＰシステムの試作”、情報学ソフトウェア研報、**５７**、４、（昭６２－１１）．

(7) J.Allen:" Anatomy of LISP", McGraw-Hill, New

York (1978).
(8) 関、井上、谷口、嵩：”関数型言語ＡＳＬ／Ｆのコンパイル時における最適化、信学論（Ｄ），Ｊ６７，１０、ｐｐ．１１１５－１１２２、（昭５９－１０）．
(9) D.W.Clark and C.C.Green:"An empirical study of list structure in lisp", Comm. ACM, 20, 2, pp.78-87(Feb. 1977).
(10) R.P.Gabriel:" Performance and evaluation of lisp systems", MIT Press, Massachusetts (1985).
(11) 黒川利明：”ＬＩＳＰ入門”、培風館、p. ９４（昭５７－０６）．
(12) J.MaCarthy, P.W.Abrahams, D.J.Edwards, T.P. Hart and M.I.Levin:" LISP 1.5 programmer's manual", MIT Press, Massachusetts (1965).
(13) 竹内郁雄：”ＬＩＳＰ処理系コンテストの結果”情報学記号処理研報、５、３（昭５３－０８）．
(14) 井上、鳥居：”囲碁のルールの代数的記述について”、情報学ソフトウェア研報、５２、８、（昭６２－０２）．
(15) 井上、川瀬、鳥居：”ガーベジセルの直接回収を行うＬＩＳＰシステムの試作”情報学ソフトウェア工学研報、５４、４（昭６２－１１）．