計算機アーキテクチャ 93－7
(1992. 3. 23)

実時間応用のためのLispによるGCフリープログラミング

GC-free Programming in Lisp for Real-time AI

杉村 利明* 岸田 克己** 日比野 靖*
Toshiaki Sugimura Katsumi Kishida Yasushi Hibino

*NTT ヒューマンインタフェース研究所 **NTT インテリジェントテクノロジ(株)
NTT Human Interface Laboratories NTT Intelligent Technology Co., Ltd.

[あらまし] LISPにおけるGC-freeプログラミングの方法について論ずる。これは、LISPをリアルタイムシステムに適用するための現実的な努力である。ELIS-8200上のCommonlispで実現された方法を詳しく紹介するとともに、新しい別の方法を提案する。

[Abstract] Methods of GC-free programming in LISP environment are discussed. This is a realistic effort to expand the LISP application area into the real time AI's. An implementaion example for ELIS Commonlisp on ELIS-8200 is precisely explained and another new method is discussed.

1. はじめに

AI(人工知能)技術は、実用化の時代を迎え適用領域の拡大が望まれている。最近では、監視システムや制御システムなどの実時間処理を伴う分野で、AI技術の応用(リアルタイムAI)に対する重要性が認識されている。

Lispは強力な記号処理言語であり、またプログラミングがしやすい、などの理由から、多くのAIシステムがLispで実現されてきた。しかし、Lisp言語は、ガーベジコレクション(GC)のためにプログラムが停止してしまうことが重大な欠点であるとされてきた。特に、実時間処理を伴う処理では、致命的な欠点になる。

我々は、この欠点を解決しLispの利点を実時間応用に生かすために、「LispによるGCフリープログラミング」を提案し、このための支援機能を実現した[1]。本機能は、ELIS-8200[2,3]のELIS Common Lisp[4]に搭載されている。この提案は、Lispの実用的観点からの提案であり、Lispを実際に使えるものにするための努力が重要であると考えたからである。

本稿では、まず、並列GCとGCフリープログラミングの違いについて議論し、ELISで提供したGCフリープログラミングの支援機能について述べる。次に、ELISのカーネル自身がGCフリープログラミングによって実現されているので、ここで実施したGCフリープログラミングを分析し、LispによるGCフリープログラミングの可能性・実用性を議論する。また、GCフリープログラミングのための新たな提案についてもふれる。

2. 並列(実時間)GC

従来から、GCの影響を少なくするために、並列(実時間)GCと呼ばれる手法が開発されてきた。並列GCでは、メモリ資源を消費する応用プログラム(AP)と不要になったメモリ資源を回収するGCを同時に並行して実行することにより、GCによるAPの停止を防ぐことが可能である。これにより、APが長い時間(数秒、長い場合は数分から数時間も)停止するという欠点はなくなる。ターミナルから普通のインラクティブな使いかたをしている限りは、GCが生じたことを感じないですむ。

しかし、このAPとGCが並行して実行される様子を細かく見れば、APの実行中に割り込んでGCが実行されているので、当然APの実行は遅くなる。本当に緊急を要する処理には、このGCの割り込みによるオーバーヘッドは無視できない。さらには、この緊急処理の実行時間は、最悪の実行条件下において保証する必要があるので、この場合GCの割り込みによるオーバーヘッドはシステムの性能を大きく低下させることになる。また、メモリ資源の消費がGCによる回収よりも早くなった場合は、GCだけが実行されAPが停止(中断)してしまうという事態が生じる。緊急処理の実行が中断されない保証はない。以上の点から、この手法は実時間応用では十分とはいえない。我々と同様のアプローチを採用したG2でも、GCの

オーバーヘッドが許容できないことを理由にあげている[5]。

3. GCフリープログラミング

我々は、GCによるAPの停止を抑止するための方法として、GCが発生しないようにプログラムを書き分ける「GCフリープログラミング」を提案してきた[1]。

この方法は、「プログラムが必要なメモリ資源を事前に確保し、そのメモリ資源のみを消費して走行するように、プログラムを記述する方法」である。言い換えれば、「プログラマが消費したメモリ資源が不要になる時点を意識し、メモリ資源の再利用をコーディングに反映させる方法」である。　メモリ資源を考えないで容易にプログラミングができるという、プロトタイピング時のLisp言語の利点を放棄するのはとんでもないことだと思われるかもしれないが、この方法は、FortranやC言語で普通に行われているプログラミングの方法で、これらの言語に慣れたプログラマにはむしろ自然な方法といえよう。

プロトタイピングの段階では、従来のLispのプログラミング通り、使う資源の管理はGCにまかせてアルゴリズムの設計にのみ集中し、その後にGCフリー化の段階でプログラムが使用している各資源の再利用のための宣言を入れる、というプログラミングの手順をとれば、Lispのプロトタイピング時の利点はそのまま生かすことができよう。

4. GCフリープログラミング支援機能

4.1 データタイプとメモリ資源再利用の可否

プログラムによるメモリ資源(データ)再利用の可否の観点から、Common Lispのデータタイプは次の5つに分類できる。

分類1 --- 小整数(fixnum)、文字、
　　短形式浮動小数点数
分類2 --- 配列、文字列
分類3 --- cons、list
分類4 --- 単形式浮動小数点数
分類5 --- 複素数、大整数(bignum)

分類1は、操作をしても新たにメモリ資源を消費せず、通常の記述が自然にGCフリーになっているもの。分類2は、あらかじめ必要な大きさのデータを作っておき、これに破壊的な操作を行いながら繰り返し使うことにより、すでにGCフリープログラミングが可能なもの。分類3は、Lispの最も重要な動的データであるが、プログラムによる再利用の手段が提供されていないもの。分類4は数値演算でメモリが消費されてしまうもの。分類5は実時間応用を含めてほとんどの場合使われないもの。

GCフリープログラミングを可能とするためには、これらの分類の中で、分類3と分類4に対してメモリ資源再利用の機能が新たに必要である。

4.2 ELIS提供の支援機能　ELISではLisp言語の特徴を損わず自然にGCフリープログラミングが行えるように、いくつかの便利な機能を提供している。

(1) 一括解放再利用型list処理関数 (work-cell-areaでのlist処理機能)

従来のLisp処理系では、セルなどの動的データは、GCによる回収が必要なメモリ資源を消費して生成しているために、「GCフリープログラミング」が困難であった。そこで、GCとは独立に、代表的な動的データであるセルのためのメモリ領域(work-cell-area)を管理する機能と、この領域を使ってlist処理を行う領域指定list処理関数を提供した。これらの領域指定list処理関数を用いてコーディングしておけば、途中でGCが生じることはない。

(1-1) work-cell-areaのメモリ管理機能

プログラムは、GCとは独立に、以下のGCフリーメモリ管理関数を使って必要なwork-cell-areaを自由に確保し再利用することができる。

○work-cell-areaの確保

プログラムは、関数allocate-areaにより事前に必要なwork-cell-areaを確保する。

以後、このwork-cell-areaのセルを使ってlist処理を行うようにプログラムをコーディングする。

○work-cell-areaの再利用

プログラムは、work-cell-area中に作ったデータが不要になったら、GCの助けを借りなくても、関数reinitialize-areaにより再度初期設定を行いwork-cell-areaの全てのセルを次のlist処理に再利用できるようにする。

○work-cell-areaの解放

プログラムは、全ての処理が終了し確保していたwork-cell-area自身が不要になった場合、関数deallocate-areaによりwork-cell-areaの解放を行うことができる。

実際には、この解放されたwork-cell-area自身は、

GCにより回収され、別の用途に再利用される。

(1-2) 領域指定list処理関数

セルを消費するlist処理関数についてのみ、新たに領域指定list処理関数を提供した。他の大部分のセルを消費しないlist処理関数(car、cdr等)は、何の変更も必要ないので、そのまま使えばよい。言い換えれば、GCフリー化の作業を行う場合、セルを消費するlist処理にだけ注意を向けて、本領域指定list処理関数を使えばよい。

主な領域指定list処理関数を表1に示す。これらの関数は、対応するCommon Lispのlist処理関数を拡張し、第1引数にwork-cell-areaを指定できるようにしたもので、指定した領域の中のセルを使ってlist処理を行う。これに加えて、個別に領域指定list処理関数を指定しなくてもよいように、within-work-cell-areaマクロを用意した。このマクロで囲んでおけば(レキシカルスコープ)、通常のlist処理関数で記述しておいても、自動的に領域指定関数として解釈される。図1に、同じ処理についてwithin-work-cell-areaで実現した場合と、個別に領域指定list処理関数を使った場合を示す。

表1 主な領域指定list処理関数
(セルを消費するlist処理関数についてのみ追加)

@append, @cons, @copy-list, @list,
@list*, @make-list, @mapcar, @merge,
@multiple-value-list, @reverse, ...

(2) 逐次解放再利用型list処理関数 プールしているセルを使って処理を行うlist関数と、プールに不要になったセルを解放するための関数を提供した。キュー操作を実現するために使うセルのように、不要になった時点で、個別にセルを逐次解放する必要がある場合に、本機能を使う。これは、大きな繰り返し処理の単位で使ったセルを全て解放するwork-cell-areaの方法とは、別のセルの解放・再利用の方法である。付録には、セルに対するGCフリープログラミングの例として、画面を動き回る虫のデモプログラムを、一括解放再利用型list処理関数と逐次解放再利用型list処理関数とを用いて実現したプログラムのソースリストを載せた。

○cons、list生成

プールしているセルを取り出して、consを作る関数cons-spとlistを作る関数list-sp、を用いてlist処理を行う。

○セルの解放

関数recycle-cellは、不要になったconsやlistをセル単位でプールに解放する。

```
(1) within-work-cell-areaマクロを使った場合
 (within-work-cell-area wc-area
   (setq x (cons x y))
   (setq y (car y))
   (setq z (list x y) ))

(2) 個別に領域指定list処理関数を使った場合
   (setq x (@cons wc-area x y))
   (setq y (car y))
   (setq z (@list wc-area x y))
```

(セルを消費するlist処理についてのみ、領域指定list処理関数を使用関数carは、セルを消費しないので従来のlist処理関数をそのまま使用)

図1 領域指定list処理関数の使用例

(3) 資源管理ライブラリ

同種の配列・文字列・構造等をプールしておき、必要になったら取り出して使い、不要になったらプールに戻し再利用するための資源管理機能を、標準ライブラリとして提供した。この機能は、いわゆるメモリ管理機能とは次元が異なり、GCによるメモリ管理機能の配下で各資源(データ)をプールして再利用することで、GCの発生を抑止するものである。後でも述べるように、デバイスドライバの実現に用いるバッファとしての配列等をプールして共有して使うのに、本機能は有効である。

○資源管理テーブルの生成

プログラムは、関数make-resource-tableにより資源をプールを行う資源管理テーブルを生成する。このとき、プールする資源を必要に応じて生成するためのconstructor関数を指定する。

○資源の確保

プログラムは、関数allocate-resourceまたはマクロ

using-resourceを使ってプールされている未使用の資源を割当ててもらい処理を行う。

○資源の解放

確保した資源が不要になったら、関数deallocate-resourceにより元のプールに解放し、再利用できるようにする。

また、関数deallocate-whole-resourceを呼べば、資源管理テーブルでプールされていて、現在は取り出して使われている全ての資源が解放される。

○資源のクリア

全ての処理が終了したら関数clear-resouceにより資源管理テーブルをクリアし、プールを空にする。

実際には、GCが実行された時点で、これらのプールされていた全ての資源が回収され、別のようとに使われることになる。

(4) 破壊版浮動小数点数演算関数

単形式浮動小数点数の計算の際に、引数自身のメモリ領域を破壊し再利用することで、新たにメモリ資源を消費しない破壊版数値演算関数を提供した。言うなれば、これはFortranのcall by referenceによる引数渡しの方法に対応するものである。

5. GCフリープログラミングの実際

GCフリープログラミングのための機能を提供しているELISのカーネル自身も、GCフリープログラミングによって実現されている。そこで、これを例題として、メモリ資源(データ)の解放と再利用の仕方を分類すると以下の2つになる。

(1) 一括解放再利用型

--- データの集まり(メモリ領域)を一括解放し、再利用する方法 ---

- 場合1-1 かな漢字変換の形態素解析処理 [分類3を対象]
- 場合1-2 ストリームでのバッファを用いた入出力 [分類2を対象]

(2) 逐次解放再利用型

--- 個々のデータを逐次解放し、再利用する方法 ---

- 場合2-1 プロセススケジュールキューなどのキュー操作 [分類3を対象]
- 場合2-2 多様なバッファの再利用操作 [分類2を対象]

場合1-1は、入力文の解析に必要なlist(セルの集まり)の再利用をwork-cell-areaの機能で実現している。

場合1-2は、配列(配列要素の集まり)のインデックスを先頭に戻すことで、配列の再利用を行っている。

場合2-1は、キュー操作でのセルの再利用を行っている。セルが不要になった時点で個別に逐次解放する。このために、逐次解放list処理機能を用いた。

場合2-2は、同種の配列・文字列・構造をプールして再利用するために資源管理ライブラリを用いた。

6. 分析・評価

(1) 場合1-1は、プロトタイピングの段階では、従来のLispのプログラミング通り、使う資源の管理はGCにまかせてアルゴリズムの設計にのみ集中できた。GCフリー化の段階でプログラムが使用している各資源の量とライフタイム(データが生成されてから不要になるまでの期間で、データの解放の時点を決定する)を調べ、work-cell-areaのための宣言を入れた。これは、「後から必要に応じて宣言を入れ効率を上げる」というCommon Lispの宣言の考え方とも合致する。

GCフリー化での重要なことは、再利用するデータのライフタイムと容量を見極めることである。一般には、かなりの負担になると考えられるが、大きな処理単位で各データのライフタイムをとらえることにより、この負担は低減できよう。実際、GCフリー化の作業が全体のプログラミングの中で占める割合はわずであった。以上のことから、一括解放再利用型のlist処理にたいして、work-cell-areaの手法により、Lispの特徴を損うことなくGCフリー化が行えると言えよう。

(2) 場合2-1の逐次解放再利用型のlist処理の中身は、各種のキュー操作である。プログラミング上は、一括解放再利用型のlist処理に比べて、セルの解放の時期を決めるのが大幅にむずかしい。理由は、処理を大まかにとらえてライフタイムを決めることができず、個々のセルのライフタイムを決定し、細かな解放制御をする必要があるからである。しかし、キュー操作を資源再利用の観点から見ると、Lispに限らず他の言語でも同様の配慮が必要で、プログラマへの負担は大きいものがある。従って、むしろキュー操作のための資源再利用型の標準ライブラリを提供するのが有効な方策といえよう。

(3) 場合1-2の配列・文字列の再利用の方法は、従来のFortranやC言語とLispで違いはない。また、場合2-2の再利用の方法も同様である。資源再利用のために、Lispにおいても、従来の言語と同様のプログラミングが行える。

7. 新たな提案（今後の機能）

7.1 新言語機能の提案

現在ELISで提供している、一括解放再利用型list処理関数と逐次解放再利用型list処理関数の機能を統一し(現在は、それぞれの機能のために別の名前の関数を用意している)、更にlistデータ以外にも資源の解放・再利用ができるように一般化を行う。そのために、言語の基本メカニズムの中に、どこのメモリ領域にデータを生成するかを指定するための統一した機能を提案する。具体的には、言語仕様として、メモリ領域指定のために必要な構文を提案し、評価機構として何が必要になるかを議論する。また、そのためのメモリ管理機能としては、何処までを用意する必要があるかを明らかにする。

(1) メモリ領域指定機能

メモリ領域の指定は、新たに提案するスペシャルフォーム within-areaで統一的に行う。この中で呼ばれた関数が、データを生成する場合は、ここで指定されたメモリ領域にデータを生成するものとする。この時、within-areaによるメモリ領域指定のスコープは、レキシカルスコープとダイナミックスコープによる方法の2つが考えられる。

(1-1) レキシカルスコープによる方法

within-areaの中で陽に呼ばれているデータ生成関数(consやlistなど)だけが、ここで指定したメモリ領域の中にデータを生成する。within-areaの中で呼ばれる関数からネストして間接的に呼ばれる場合は、ここでの指定は有効でない(スコープの外)。GCフリー化において使用しているメモリ領域を意識する必要のあるコードには、陽にそれが記述されているのが、プログラムを理解する立場から有用で、自然な指定法と考える。プログラムを変更する場合は、領域を指定している部分をそれぞれ変更する必要が生じ作業量が増大するといるデメリットもあるが、誤った領域指定の範囲を限定できるので、領域指定は慎重に行うことが重要であるという観点から、本方法を有力視している。　本方法では、関数はwithin-areaによって指定される「第2の引数」を持つことになったと考えることができる。呼びだし側は、within-areaで関数呼び出しを囲むことでこの「第2の引数」を指定する。呼び出された側は、以下のようにメモリ領域指定を行うことで、呼びだし側の領域指定を参照し、これを次の関数呼び出しに引き渡すことができる。これにより、呼びだし側で指定したメモリ領域に、呼び出され側がデータを生成することが可能になる。なお、呼び出された側が陽に、指定したメモリ領域を参照しない場合は、従来のGCが管理している共通メモリ領域にデータが生成されるものとする。また、Common Lispで提供しているデータ生成関数(cons、list、make-arrayなど)は、within-areaで陽に指定したメモリ領域にデータを生成するように作られているものとする。

```
(defun foo (x y)
  (within-area (:area *area-1*)
    ;; 関数listは、スペシャル変数*area-1*で
    ;; 指定したメモリ領域にlistデータを生成
    ;; する。
    (list x (bar y)) ))

(defun bar (x)
  (within-area (:area :caller)
    ;; 関数consは、:caller指定により、関数
    ;; barの呼びだし側で指定したメモリ領域
    ;; にconsデータを生成する。
    (cons x 1) ))
```

実現上の観点からは、メモリ領域指定のための引数が関数呼び出しの時に内部的に渡されるように、評価機構を変更する必要がある。この引数は、必ずしも渡す必要はないので、コンパイル時の最適化により省力することにより、このオーバヘッドを低減することが可能である。

(1-2) ダイナミックスコープによる方法

within-areaの中から呼び出されている(直接・間接に関係なく)すべてのデータ生成関数が、ここで指定したメモリ領域の中にデータを生成する。このように、ダイナミックスコープによる方法は強力ではあるが、逆に、この指定の影響を防ぐための対処が必要になるので(レキシカルスコープによる方法では、領域指定を意識する必要のある部分だけに陽にそのコードを書けばよいが)、GCフーリ化の対象外のプログラムにも、この対処が必要となるというデメリットがある。

実現上の観点からは、スペシャル変数を使うことにより実現でき、評価機構のそのものを大きく変える必要はないというメリットがある。

(2) メモリ領域の種類とメモリ管理機能

5章で分類したように、データ資源の再利用の仕方には、一括解放再利用型と逐次解放再利用型の2つがあり、これに対応したメモリ領域を提供する。更に、一時的な処理に便利なメモリ領域を提供する。

○一括解放再利用型メモリ領域

work-cell-areaの汎用版である。このメモリ領域で

は、任意のデータを生成することができる。この中の全データが不要になったら、一括して解放・回収を行う。一括して、データを解放するので、フラグメンテーションの問題は発生しない。

このメモリ領域のデータの解放はreinitialize-areaで行う。

(現在のELISはポインタタグ方式であるので、汎用の一括解放再利用型メモリ領域を提供することはできなかったが、データタグ方式ではこのメモリ領域の中のデータの解放・参照に誤りがあっても、システムダウンに至らないようにできるので、本メモリ領域の提供が可能になる)

○逐次解放再利用型メモリ領域

逐次解放再利用型list処理の汎用版のためのメモリ領域である。このメモリ領域にも任意のデータを生成することができ、不要になったら、逐次そのデータを解放することができる。しかし、長さの異なるデータを同じメモリ領域で生成し解放すると、フラグメンテーションが発生するという問題がある。前もって見積もった数のデータを、所定の時間内に確実に生成できる、という実時間の要求条件を保証させるためには、1つのメモリ領域からは同一サイズのデータを生成するという制限を課する必要がある。

このメモリ領域への個別のデータの解放は、release-objectで行う。

○局所型メモリ領域

数値計算などのような局所的な計算結果を生成するためのメモリ領域である。

within-areaのブロック構造の中で使ったデータを、このブロックを出る時に解放するという使い方である(スタック的な使い方)。

○共通メモリ領域

通常のLisp処理系で提供しているヒープに相当する。なお、GCによって回収されないことが分っているデータは、本メモリ領域を更に分化したGCの対象にならないメモリ領域(静的メモリ領域)に生成するということも可能である。

<例>

```
(defun foo (a b c d &aux x y)
  (within-area (:area :tmp-area)
    ;; かけ算、わり算の結果は局所型メ
    ;; モリ領域に生成される
    ;; この構文を出ると、ここで使用し
    ;; た資源は解放される
    (setq x (* a b))
    (setq y (/ c d)) )
    (within-area (:area *area-1*)
      ;; 足算の結果は、*area-1*で指定
      ;; したメモリ領域に生成される
      (+ x y) )))
```

7.2 並列GCとGCフリープログラミングの共存

現在のELISで提供しているGCフリープログラミングのための支援機能では、APを含めてシステム全体をGCフリー化してはじめて、GCフリー化の効果が発揮できるが、更に並列GCとGCフリープログラミングとを組み合わせ補完させることにより、GCフリープログラミングの必要な範囲を最小限にすることが考えられる。すなわち、緊急処理などのクリティカルなものだけをGCフリープログラミングし、それ以外の処理は、並列GCにまかせる方法である。GCフリー化されていない他の部分がGCの処理の最中であっても、緊急処理が割り込んで実行可能となる。そして、この緊急処理は既に確保した資源を使って処理を行うように記述されているので、資源が不足してGCが発生し緊急処理の実行が停止することはなく、期待した時間内に緊急処理を完了することができる。また、並列GCとの組み合わせにより、緊急処理以外の処理のレスポンスの向上(長い待ちをなくす)を同時に達成することもできる。

7.3 GCフリープログラミングのためのデバッグ機能 GCフリー化を行う場合のバグの原因は、大きく分けて次の2つに分類できる。

○参照誤り

早過ぎた解放により、必要なデータが正しく参照できない場合。

解放済みのデータを誤って参照しようとした。

○解放忘れ

不要になったデータの解放忘れ。

(1) 参照誤りの検出処理

参照誤り検出のためのデバッグ機能の基本メカニズムは、以下の通りである。デバッグ時には、GCの機能を利用して参照誤りを

検出する。

1) メモリ領域の指定を無視して、共通メモリ領域にデータを生成する。
2) 解放は、そのデータの存在していた場所(メモリ)に解放済みの印を付け、後から誤った参照が行われた場合は、エラーが検出できるようにする。
3) 解放済みの印の付いた領域は、参照がなければ、GCにより回収され、次のデータ生成に再利用される。

(2) 解放忘れの検出

大量に生成されるデータ(メモリ消費が高い)の解放忘れは、メモリ不足として検出される可能性が高い。また、room等により使用量を測定することにより、生成はわずかではあるが、解放忘れのために単調に増加する場合も検出可能である。しかし、希にしか実行されない、異状処理等での解放忘れの検出はむずかしい(そもそも、希にしか実行されない部分のデバッグはGCフリー化に関係なくむずかしい)。また、解放忘れの検出は、Lispだからむずかしいということはなく、他の言語でも再利用を行うためのプログラムは、同様にむずかしいものがあろう。解放忘れの検出に関して、特効薬的なデバッグ機能はなさそうである。しいて言えば、Lispは一般のデバッグ機能が充実しているというのが強味の一つである。

8. まとめ

実時間応用をねらいとして、LispによるGCフリープログラミングを提案し、このためにELISで提供した支援機能機能について報告した。次に、LispにおけるGCフリープログラミングが実際にどのように行われているかを分析・評価し、work-cell-areaの機能などにより、Lispの特徴を生かしながらGCフリープログラミングが実現できることを確認した。

本機能を搭載したELIS AIエンジンボード[6]は、ネットワークの警報情報の選択処理に適用するために、伝送路網運用保守システムへ導入された。このシステムの実現においてGCフリープログラムによる完全無停止走行の実現がキーポイントになっている。

今後は、(1)listが対象であった機能を、他の動的データに拡張すること、(2)メモリ資源制御のための言語構造とメカニズムの洗練化、を行っていきたい。

<謝辞> 日頃御指導いただくマルチメディア研究部 遠藤 隆也部長に感謝致します。

<参考文献>

[1] 杉村,岸田他: ELIS Common LispのGCフリーコーディング機能, 情報処理第42回全国大会, 1990.

[2] 渡邊,川村,日比野: AIエンジン用LSIの開発, NTT R&D Vol.39.No.9,1990.

[3] 鈴木,菅原,杉村: リアルタイムAI用OSカーネル, NTT R&D Vol.39.No.9,1990.

[4] 杉村,岸田: ELIS Common Lispのマルチプログラミング機能,情報処理学会第39回全国大会,5Q-1,1989.

[5] JAMES R. ALLARD他: Real-Time Programming in Common Lisp, CACM, Vol.34, No.9, 1991, pp.65-69.

[6] 渋谷,小平,鈴木: 機器組込み用ELIS-VMEボードのハードウェア,情報処理学会第42回全国大会, 1989.

[付録] GCフリープログラミングの例
(画面を動き回る虫のプログラムwormを例として)

```
;;;     データ、変数の定義       ;;;
;;;     queueの定義
(defstruct queue top)
;;;     wormの定義
(defstruct (worm (:include queue)))
;;;     work-cell-areaを指す大域変数の定義
(defvar *worm-wk-cell* nil)


;;;     逐次解放再利用型list処理関数の定義       ;;;
(defvar *cell-pool* nil)

(defun cons-sp (x y &aux top-cell)
    (cond (*cell-pool*
           (setq top-cell *cell-pool*)
           (setq *cell-pool* (cdr *cell-pool*))
           (setf (car top-cell) x (cdr top-cell) y)
           top-cell )
          (t (cons x y)) ))

(defun recycle-cell (cell)
    (setf (cdr cell) *cell-pool*)
    (setq *cell-pool* cell) )


;;;     queue操作関数(資源再利用型)の定義       ;;;
(defun enqueue (queue entry)
   (if (queue-top queue)
       (nconc (queue-top queue) (cons-sp entry nil))
       (setf (queue-top queue) (cons-sp entry nil)) ))

(defun dequeue (queue &aux work)
    (progi (car (queue-top queue))
           (setq work (queue-top queue))
           (setf (queue-top queue) (cdr (queue-top queue)))
           (recycle-cell work) ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;          worm プログラム本体            ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun worm (&aux *old* pos worm last-pos counter)
    (declare (special *old*))
    (erase-all-display *terminal-io*)          ; 画面クリア
    (setq worm (make-worm))
    ;; work-cell-areaの確保
    (setq *worm-wk-cell* ($rt:allocate-area $rt:work-cell-area 500))
```

```
    (setq *old* 0 counter 0)
    ;; 初期wormの生成 (逐次解放再利用型list処理)
    (setq pos (cons-sp 12 20))                    ; 初期座標
    (dotimes (i 20)
       (enqueue worm (setq pos (next-position pos)))
       (put-worm pos) )
    ;; wormの移動
    (loop (when (zerop (mod (incf counter) 2500))
            ;; 一定期間移動したら画面をクリア
            (erase-all-display *terminal-io*)
            (setq counter 0) )
          ;; work-cell-areaのクリア (一括解放再利用型list処理)
          ($rt:reinitialize-area *worm-wk-cell*)
          ;; wormの最後尾を消す (資源再利用型queue操作)
          (setq last-pos (dequeue worm))
          (remove-worm last-pos)
          ;; cellの解放 (逐次解放再利用型list処理)
          (recycle-cell last-pos)
          ;; wormの頭を一つ進める (資源再利用型queue操作)
          (enqueue worm (setq pos (next-position pos)))
          (put-worm pos) ))

;;;   座標pにwormのbodyを表示
(defun put-worm (p)
    (cursor p)
    ;; 色をランダムに選んでbodyを表示する
    (with-color (random-color) *terminal-io*
     (write-char #\● *terminal-io*) ))

;;;   座標pのwormのbodyを削除
(defun remove-worm (p)
    (cursor p)
    (write-char #\・ *terminal-io*) )

;;;   カーサの位置を設定
(defun cursor (p &aux x y)
    (write-byte-seq *terminal-io* '(#o33 #o133))
    (write-byte-seq *terminal-io* (decompose (car p)))
    (write-byte-seq *terminal-io* '(#o73))
    (write-byte-seq *terminal-io* (decompose (cdr p)))
    (write-byte-seq *terminal-io* '(#o110)) )

;;;   カーサの位置指定パラメタの分解
(defun decompose (x)
    ;; 一括解放型list処理関数の実行 (一括解放型list処理関数の利用)
    (within-work-cell-area *worm-wk-cell*
      (list (+ (floor x 10) #o60) (+ (mod x 10) #o60)) ))

;;;   wormの次の先頭位置を求める (逐次解放再利用型list処理関数の利用)
(defun next-position (p &aux r column line ncolumn nline)
    (declare (special *old*))
    (setq line (car p) column (cdr p) r (random 3))
    (case (setq *old* (mod (+ *old* r 7) 8))
      (0 (setq nline line) (setq ncolumn (+ column 2)))
      (1 (setq nline (1+ line)) (setq ncolumn (+ column 2)))
      (2 (setq nline (1+ line)) (setq ncolumn column))
      (3 (setq nline (1+ line)) (setq ncolumn (- column 2)))
      (4 (setq nline line) (setq ncolumn (- column 2)))
      (5 (setq nline (1- line)) (setq ncolumn (- column 2)))
      (6 (setq nline (1- line)) (setq ncolumn column))
      (7 (setq nline (1- line)) (setq ncolumn (+ column 2))) )
    (cond ((= nline 1) (setq nline 3) (setq *old* (mod (+ *old* 2) 8)))
          ((= nline 24) (setq nline 22) (setq *old* (mod (+ *old* 2) 8))) )
    (cond ((= ncolumn 2) (setq ncolumn 4) (setq *old* (mod (+ *old* 2) 8)))
          ((= ncolumn 80) (setq ncolumn 78) (setq *old* (mod (+ *old* 2) 8))) )
    ;; 逐次解放再利用型list処理関数の実行
    (cons-sp nline ncolumn) )
```

```
;;;     color primitive の定義      ;;;
(defconstant COLOR-ATTRIBUTE-LIST
    '((:off . "0")
      (:赤 . "31") (:緑 . "32") (:黄色 . "33") (:青 . "34")
      (:マゼンダ . "35") (:シアン . "36") (:白 . "37") ))

(defconstant NUM-TO-COLOR
    '((0 . :赤) (1 . :緑) (2 . :黄色) (3 . :青)
      (4 . :マゼンダ) (5 . :シアン) (6 . :白) ))

(defconstant MAX-COLOR (length NUM-TO-COLOR))

(defmacro with-color (color stream &body body)
    `(unwind-protect
       (progn
        (set-color-attribute ,color ,stream)
        ,@body)
       (set-color-attribute :off ,stream) ))

(defun set-color-attribute (color stream)
    (let ((color-attribute (cdr (assoc color COLOR-ATTRIBUTE-LIST))))
      (when color-attribute
       (write-char #\escape stream)
       (write-char #\[ stream)
       (write-string color-attribute stream)
       (write-char #\m stream) )))

(defun random-color ()
    (cdr (assoc (random MAX-COLOR) NUM-TO-COLOR) ))


;;;     その他の下請け関数の定義      ;;;
;;;     バイト列の出力
(defun write-byte-seq (stream codes)
    (for i codes (write-byte i stream)) )
;;;     画面クリア
(defun erase-all-display (stream)
    (terminal::print-escape-sequence stream "[2J") )
```