

拡張 wp による記憶管理

寺 田 実†

ユーザとの対話を直接担当するプログラムのように多様な処理を長時間にわたり行う場合、利用されなくなった情報（関数定義など）を捨てることが重要になってくる。しかし通常のごみ集め（gc）では、到達可能なデータ（すなわち、少しでも再利用の可能性のあるもの）は回収できない。ここで注目したのが、いくつかの Lisp にある Weak Pointer (wp) というデータ型である。wp は単独では参照先を gc から保護しないという特徴を持つ。この機能を拡張し記憶管理のために利用しようというのが本研究である。拡張点は3点ある：捨てた後のポインタの値の個別指定、段階的なデータ保持能力、減衰によるデータ解放である。この拡張により、データの重要性や利用頻度に対応した保持期間などを設定できる。また、捨てた後での再ロードも容易に実現できるようになる。本論文では、拡張 wp の実現方式について、gc アルゴリズムの変更も含めて考察する。さらに拡張 wp を Unix 上での Lisp 処理系に実装し、アプリケーションを作成して効果やオーバーヘッドについて評価を行い、実用性を立証する。

Storage Management with Extended Weak Pointer

MINORU TERADA†

It is difficult for computers to discard unused information. In Lisp, a data type known as Weak Pointer can release its pointer by the help of garbage collector. We propose three extensions to this type and show the techniques for its implementation. These extensions are individual default value, variable strength wp, and decaying wp. The extended wp enables auto-reloading of discarded objects, controlling the resident time for cached objects. We implemented the extended wp on a existing Lisp system, and evaluated its performance.

1. はじめに

ヒープに対する自動記憶管理機構をそなえた言語システムが多く用途に対して利用されるようになってきた。例えば GNU Emacs¹⁾は汎用ツールとして、多様な作業（文書編集、プログラム作成、メール送受信、かな漢字変換など）の支援に利用されている。ところが、特に Lisp のように関数定義自身が記憶管理の対象となるシステムでは、多様な処理を行うためにロードした関数がヒープを占有することが多くなり、実行効率の悪化を招く。

関数定義のロードに関しては、実際の利用時まで遅らせてヒープ占有量を軽減するとともにシステム起動の高速化をはかる技法として、関数定義の自動ロードがある。1引数関数 f の定義がファイル“f.1”に格納されているとして、この関数を自動ロードするには以下のような定義を f に与えておけばよい^{*}。

```
(defun f(arg) (exfile "f.1") (f arg))
```

この自動ロード関数の実行によって、 f の定義が本来のものに書きかわり、再帰的に見える最後の呼出しによって、本来の定義が起動される。これ以降は f の定義は本来のものになったままで、再びファイルからロードされることはない。ここでの問題点は、一度ロードされてしまった関数定義は、それが intern されたシンボルから指されているために、不要になっても（自動的に）捨てることができないことである。通常のガーベジコレクション（以後 gc と呼ぶ）はルートから到達不可となったもののみを対象とするから、不要であっても到達可能なものは回収できない。ルートから到達可能な不要物を回収する方法が必要となるのである。

類似した状況がオペレーティングシステムにおけるメモリ管理部分に見られる。デマンドページング方式による仮想記憶では、実メモリ上にないページの参照時にページフォルトが発生し、メモリへの読み込みが行われる。一方それとは独立に実メモリを走査してい

† 東京大学工学部機械情報工学科
Department of Mechano-Informatics, Faculty of Engineering, The University of Tokyo

* 本論文では、Lisp 処理系として Utilisp を用いる。exfile はファイルの内容を順次評価する関数で、load に対応する。

るプロセス (pagedaemon) が存在し、最近使用されていないメモリページを解放していく。

本研究は、この pagedaemon のような機能の Lisp での実現を目指すもので、その方法は、Weak Pointer (以後 wp と呼ぶ) として知られているデータ型を拡張し、gc の協力のもとにヒープ中のデータを不要になった段階で捨てるというものである。

wp とは参照先を gc から保護しないという特徴を持つポインタである。図 1 を用いてこれを説明する。左側が gc 前の状況であり、ポインタ p は通常のポインタとし、ポインタ w1, w2 が wp である。w1 と p は同一オブジェクト X を参照しているが、w2 の参照する Y には、他の通常ポインタによる参照はない。ここで gc が発生すると、状況は右ようになる。X は生存し、リロケーションを伴う gc の場合には w1, p とともに X のリロケーションに追従して更新を受ける。一方 w2 は Y に対する参照を失い適当な初期値 (nil, #f など) にリセットされる。その結果として Y はごみとして回収される。

wp については、generational gc に関する論文²⁾に実現法について言及があり、また Scheme 処理系 T³⁾、MIT Scheme⁴⁾に実装されている。用途として文献²⁾には以下のふたつがあげてある：

1. (現在使用中のオブジェクトの集合) 図 2 に示すように、現在使用中のオブジェクトを通常ポインタを用いて管理した場合、本来の使用 (図中の破線) が終了しても集合からの参照 (太線) が残り、回収できない。T では、wp を用いた weak set というデータ構造をそのために利用する。
2. (冗長性を持つ逆ポインタ) 親子関係における、

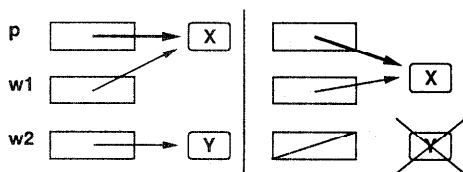


図 1 wp の動作
Fig. 1 Releasing pointer of wp.

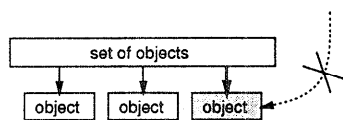


図 2 wp の利用例—オブジェクトの集合
Fig. 2 Managing set of objects.

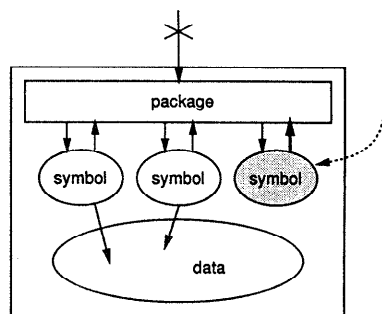


図 3 wp の利用例—逆ポインタ
Fig. 3 Reverse pointer retains unused package.

子から親へのポインタなど。一例として、package を持つ Lisp において、図 3 に示すように、不要になったパッケージに属するあるシンボルへ外部から参照 (図の破線) があると、逆ポインタ (太線) によりパッケージ自身が生存し、その結果としてそれに属する他の多数のシンボルも捨てることができなくなりヒープを圧迫する*。

本研究で注目するのは、ヒープをキャッシュと見た場合の自動的追出し機構としての応用である。必要に応じて外界からヒープにロードされた情報を、不必要になった段階で捨てるために wp が役立つ。外界からロードした情報は、wp で保持することにする。その情報を使用している間は通常ポインタからも参照を受けるので、回収される危険性がない。使用が終了した段階で、参照は wp によるものだけになるため、次の gc で回収される。次の gc をまたずに再度の使用が開始されれば、その情報は次の gc を生き残り、キャッシュとして働き続ける。

この方法により、これまで必要悪とされてきた gc を積極的に活用して、ヒープを外部記憶のキャッシュとして利用することが可能になる。

本論文の構成は、第 2 章で wp の拡張を提案し、第 3 章では wp の実現方式と、必要となる gc アルゴリズムの変更点を 2 種の代表的な gc アルゴリズムに対して示す。第 4 章では、拡張 wp を利用したアプリケーションの例を示す。第 5 章では、拡張 wp の Util-Lisp/C⁵⁾への実装について、処理系の変更点、オーバーヘッドの計測、アプリケーションにおける有効性データなどを示す。第 6 章では、実現法や問題点について議論を行う。

* NTT の竹内郁雄氏の指摘による。

2. wp の拡張

本章では、wp オブジェクトを三つの方法で拡張し、より高い機能を与える。

2.1 リセット値つきの wp

前章の wp の説明では、参照対象を失った wp は一律に特定の値にリセットされるとした。しかし、個々の wp についてリセット値を個別に設定することで、より柔軟な管理が可能になる。具体的には wp オブジェクト生成時に、参照対象の値だけでなくリセット値もユーザが引数として指定できるようにすればよい。このような wp をユーザから見ると、ある時点（一般ポインタからの参照を失って gc が起きた時点）で参照先が自動的に変化することになる。第4章で説明する関数定義の自動アンロードは、この機能を用いて実現する。

2.2 強さを持つ wp

wp と通常ポインタの違いは、参照先のオブジェクトに対する保護能力の強弱と考えることができる。この強弱を、2段階だけでなく中間的な段階にまで拡張する。強さ0の wp が最強で、通常のポインタに相当する。これ以下、1, 2, 3 と次第に弱くなっていく。これに対応して、gc にも強さを与える。0を最強とし、通常ポインタ以外をすべてリセットの候補とする。一般に強さ i の gc によって、強さ i をこえる（つまり i より弱い）wp がリセットの候補になる。

これを用いると、保持するデータの重要性にあわせて適当な強さの wp を利用することが可能となる。入力データなど再現不可能なものは通常ポインタ（強さ0の wp）で保持し、再計算可能なものはそれより弱い wp にする。外部ファイルの内容のキャッシュのような再計算のコストが少ないものはさらに弱い wp とする。

こうして利用者がデータの重要性を指示できれば、有限の資源であるヒープを有効利用できるようになる。ヒープ不足の致命的エラーが起きた場合には強さ0の最強 gc を行うことによって本当に重要なデータだけを残して、そのデータのファイルへの退避に必要な資源を確保することができる。

2.3 減衰する wp

wp のリセットについて、参照先が通常ポインタからの参照を失ってすぐ次の gc で行うのではなく、一定回数の gc という時間的余裕を与えることを考える。wp ごとにカウンタを持たせ、gc が起きるたび

にそれを減らしていく。カウンタが0でない間はその wp は通常ポインタとして扱い、0になった段階でリセットの候補とする。つまり、しだいに忘れることを可能にする機構である。

カウンタに設定する値を調整することで、その wp が保持するデータがヒープに残留できる時間が制御できる。再計算のコストが大きいデータについては大きなカウンタ値を設定することで、再利用までの時間をのばすことができる。一方外部ファイルのデータのキャッシュなどでは初期値0を与えることで、最初の gc ですぐにリセットの候補になる。

さらに、wp の保持するデータが実際に再利用された場合には、カウンタを元の（大きな）値に（プログラムから明示的に）戻してやることによって、LRU (Least Recently Used) 方式の管理が可能になる。こうすると、適当な間隔で再利用されるデータは常にヒープ中に保持され続ける。

なお、カウンタの減算については、上に述べたような無条件方式のほかに、通常ポインタからの参照がない場合に限って減算を行う方式も考えられる。

2.4 3拡張の併用

以上の3方式は相互に独立であるため、併用することができる。ただし、強さと減衰との関係については、以下のように定めるのが適当であろう。

- (wpの強さ < gcの強さ) その wp は、減衰カウンタの値にかかわらず、ただちにリセットの候補となる。
- (wpの強さ = gcの強さ) 減衰カウンタを1減じて、0になった場合のみリセットの候補とする。
- (wpの強さ > gcの強さ) 減衰カウンタも減らず、リセットの候補にもしない。

3. wp の実現方式

3.1 wp オブジェクトの導入

文献2)にあるような拡張のない wp の実装には、あるポインタが wp であるかどうかの判定のためにポインタ中に1ビットの場所を確保すればよい。しかし、この wp ビットの付加による wp の実現は、実際には困難であることが多い。第一に、汎用ハードウェア上での Lisp 処理系においては、ポインタ中に新たに1ビットを確保するのは困難である。豊富なタグビットが利用可能な Lisp 専用のアーキテクチャと異なり、汎用アーキテクチャでは、ポインタ内の余裕はデータ型判別用のタグとしてすでに利用済である場合

が多い。第二に、前章で述べた拡張 wp の実現にあたっては複数のフィールドが必要になり、この方法は不可能になる。

以上の理由から、wp を保持する専用のオブジェクトを導入するのが現実的である。前節の拡張をすべて行った場合には、wp オブジェクトの構造は以下のようなフィールドからなる。

- (1) 参照先
- (2) リセット値
- (3) 強 さ
- (4) 減衰カウンタ
- (5) gc 用の作業領域

(1)はこの wp が保持している参照先である。この参照先に対して他からの参照がなければ、このフィールドが(2)の値にリセットされる。(3)はこの wp の強さを表す整数値を持ち、gc の強さとの関係でリセット動作やカウンタの減衰を制御する。(4)は wp の減衰にかかわる整数値を持ち、0 になった時点で(1)のフィールドがリセットの候補となる。(5)については gc アルゴリズムの項で説明する。

wp オブジェクト導入に伴って新設する組込関数としては、wp オブジェクトの生成関数、(1)-(4)の各フィールドの参照/更新関数、gc の強さの設定関数がある。

なお T や MIT Scheme では、オブジェクトのハッシュ値をもって wp として利用している。一般のハッシュと異なり、ハッシュ値と実体との間に一対一対応をもたせ逆ハッシュ関数による復元を可能とすることにより、ハッシュ値を論理的に wp として使用できる。実体が gc により失われた場合(本論文でいうリセットが発生した場合)には逆ハッシュ関数が特別の値(#f)を返すことにより、その事実が分かる。実装を考えると、この方式は wp オブジェクトをヒープとは別の専用領域に格納し、その添字をもってハッシュ値とすることに相当する。

3.2 wp の不可視化

wp オブジェクトを導入した場合、そのままでは wp 経由の参照のたびに組込関数(例えば wpref)を明示的に呼び出す必要がある。これでは、wp である場合とそうでない場合とでプログラムのセマンティクスに影響を与えてしまうので、wp 経由の参照があった場合に wpref の呼出しを自動的に行うのが望ましい。こうすると wp 経由であることが見えなくなるため、これを wp の不可視化と呼ぶことにする。

ここで、不可視化に伴う問題点を、実行効率と処理系への変更量の2点に分けて論じる。

実行効率の点からは、ポインタ参照のたびに wp かどうかの判定と、wp であった場合にはもう一段の参照が必要になる。特に判定操作は通常ポインタも含めてすべての場合に必要となるため、重大なオーバーヘッドとなりうる。

処理系への変更量の観点からは、ポインタ参照を行う多くの場所での前述の判定と参照が必要となり、処理系の複雑化、巨大化をまぬがれない。

(ここで仮に、他の必要性から別種の不可視ポインタが既に導入済であった場合には、不可視ポインタの判別が実装されているので、wp のための不可視ポインタの導入によって新たに生じるポインタ操作のためのコストはほとんどない。「他の必要性」の具体例としては、gc アルゴリズムからの要請などがあげられる。)

この問題点を回避するために本論文で採用した方法は、上述のような全 wp の不可視化ではなく、不可視とするフィールドを限定するものである。具体的には、シンボルの関数定義フィールドからの wp 経由の参照だけを不可視とした。

定義フィールドに限った理由は以下のとおり：

- (1) 処理系への修正がきわめて限定しやすい。(本実装では3箇所)
- (2) 通常ポインタの場合に効率悪化を起こさない。(エラー処理部分への修正ですむ)
- (3) 他フィールドに比較して不可視化の利用価値が高い。

3.3 ポインタ属性とフィールド属性

ところで、wp の「弱さ」について、それがポインタ自身の属性であるのか、あるいはそれを保持しているフィールド(シンボルの値フィールド、コンスセルの car, cdr など)の属性であるかを区別する必要がある。それぞれの立場を、ポインタ属性、フィールド属性と呼んでおく。その区別は、wp の参照・代入時にその「弱さ」も同時にコピーされるかどうかの違いである。表 1 に、それぞれの場合をまとめる。

表 1 ポインタ属性とフィールド属性の相違
Table 1 Different behaviours in wp manipulation.

	他へのコピー	通常ポインタの代入
ポインタ属性	wp のまま	通常ポインタのまま
フィールド属性	通常ポインタになる	wp になる

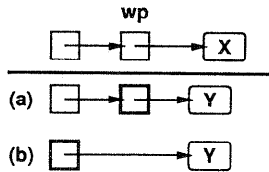


図 4 不可視ポインタへの代入のふたつの方式
Fig. 4 Assignment to wp—two methods.

これらの区別は、不可視 wp オブジェクトを導入する場合には、2段階のポインタのいずれを操作するかの差となる。ポインタ属性とするには、wp オブジェクトを指している（1段目の）ポインタを操作すればよい。逆に、フィールド属性とするには、wp オブジェクトから出ている（2段目の）ポインタを操作すればよい。例として、現在Xなる値を持つ wp に対しYなる（通常の）値を代入する場合には図4のようになる。（フィールド属性が(a)、ポインタ属性の立場なら(b)である。）

本実装では、自動的に wp に変化してしまう危険性を減らすために、参照時はフィールド属性の立場をとり、代入時はポインタ属性の立場をとった。（ただし、この方式では、自分自身の値を自分自身に代入することによって wp を経由しなくなってしまうという問題点があるが、深刻なものではない。）

3.4 gc の変更

本節では、wp オブジェクトを導入した場合に必要な gc への変更を述べる。ここでは拡張なしの wp を例として記述するが、拡張を行った wp についても本質的には同じである。

3.4.1 マークスイープ方式 gc への組込

wp を初めて訪問した場合、その wp オブジェクトにはマークをつけるがそれが指す先にはマークにいかない。さらに、その wp オブジェクトを「生存 wp リスト」に登録する。（このために wp オブジェクトにリンク領域が1語必要となる。）

マークフェイズ終了後、スイープフェイズに移るまでに、生存 wp リストを走査し、各 wp オブジェクトについて、

- (a) 参照先にマークがあれば（通常ポインタからも参照を受けているので）何もしない。
- (b) 参照先にマークがない場合は、wp のみによる参照であるから、その wp オブジェクトの参照先フィールドを nil にリセットする。

スイープフェイズは wp オブジェクトも含めて普通と同様に行う。

3.4.2 コピー方式 gc への組込

コピー中に wp オブジェクトを初めて訪問した時、以下を順に行う。

- (1) その wp オブジェクト自身は新ヒープにコピーする。
- (2) コピー済のしるしと移転先を残す。
- (3) wp からの参照先のコピーは行わない。
- (4) 生存 wp リストに旧 wp オブジェクトを登録する。

旧ヒープのコピーが完了した段階で生存 wp リストを走査し、各(旧) wp オブジェクトに対して参照の状態に応じて以下のいずれかを行う：

- (a) 対応する新 wp オブジェクトの参照先（旧領域を指す）がコピー済であれば、ポインタを更新する。
- (b) 参照先がコピーされていない場合は、ポインタを nil にリセットする。

ここで、コピー時の生存 wp リストへの登録には、旧 wp オブジェクトの適当なフィールドを利用できる。つまり、wp オブジェクトがポインタ2本以上の大きさを持っていれば、移転先ポインタと登録とに利用できるから、マークスイープ方式と違い、gc のために余分なフィールドはいらなくなる。（本節の説明では wp オブジェクトは参照先ポインタだけを持つように説明してきたので、gc 用フィールドは節約できない。しかし、前節で述べた拡張を行うとフィールドの数が増えるため、この節約が可能になる。）

gc の性能への影響は、いずれの方式の gc についても生存 wp リストの走査が増えるだけであり、wp の個数に比例した増分となる。

4. wp によるアプリケーション例

—関数定義の自動アンロード

自動アンロードとは、自動ロードと逆に関数定義を適当な時点でヒープから捨て、次回に呼び出された時に自動ロードするものである。基本的なアイデアは、wp のリセット値として再ロードのコードを保持しておき、その wp が回収の対象になった段階で関数定義が再ロードのコードに変化するように設定するのである。

この場合に重要なのは、シンボルの定義フィールドから wp を経由する参照を不可視にすることである。対象となる関数を呼び出した場合に、wp オブジェクトからもう一段参照を行う必要があるが、それをユー

- 通常の defun の定義

```
(macro defun (l)
  (putd ',(first l)
    (function (lambda . ,(cdr l))))))
```
- 自動アンロード対応の defun の定義

```
(macro defun (l)
  (putd ',(first l)
    (wp (function (lambda . ,(cdr l))
      (function (macro lambda (ll)
        ,*reload*
        (cons ',(first l) ll)))
        *strength*
        *counter*))))
```
- ファイル "foo.l" の内容

```
(defun foo (a0 a1) (body-of-foo))
```
- 通常の exfile による foo の定義結果

```
(lambda (a0 a1) (body-of-foo))
```
- 自動アンロード対応の exfile による foo の定義結果
 - 参照値

```
(lambda (a0 a1) (body-of-foo))
```
 - リセット値

```
(macro lambda (ll)
  (exfile "foo.l") '(foo . ,ll))
```
 - 減衰カウンタ 3
 - 強さ 1

図 5 自動アンロードのための定義と実行例

Fig. 5 Definitions and example of auto-unloading.

ザが検出するのでは使いやすさの面でも実行効率の面でも問題になる。ここでは、3.2 節で述べたように、シンボルの定義フィールドに対してだけ wp を不可視とすることにした。

自動アンロード実現のためには、ファイルをロードする関数 exfile と、関数定義のためのマクロ defun を修正する。exfile は引数としてファイル名をとるので、これを再ロード時のファイル名としても利用するためにスペシャル変数（動的バインドの対象となる変数）に保持する。defun は、通常はシンボルの定義フィールドに定義を格納するだけであるが、これを適切な wp を生成して格納するように修正する。その wp の減衰カウンタや強さについてはスペシャル変数であらかじめ設定するようにした。

修正した defun の定義と、実行例を図 5 に示す。putd はシンボルの関数定義フィールドの更新関数で、Common Lisp では symbol-function を setf とともに用いることに相当する。スペシャル変数 *reload* は、再ロードのための式 (exfile "foo.l") を値とし、関数 exfile によってバインドされる。減衰カウンタと wp の強さはそれぞれスペシャル変数 *counter*

strength で設定するようになっており、既定値はそれぞれ 3, 1 としてある。gc の強さは既定値のままの 0 となっている。（つまり、この例では、wp の強さに関する拡張を利用していない。）

5. UtiLisp/C への実装と評価

wp の実現可能性を実証し、その性能を評価する目的で、UtiLisp/C に wp を実装した。

5.1 処理系への変更点

実装方式としては、第 3 章で述べた拡張をすべて含む wp オブジェクトを新設し、その生成/アクセスに関連する組込関数を 10 個定義した（表 2）。UtiLisp/C の gc はコピー方式であるので、3.4.2 項で述べたアルゴリズムを実装した。

さらに、シンボルの定義フィールドからの wp 参照を不可視にするよう変更を行った。これは式評価のための関数 eval, funcall の修正のほか、定義フィールドのとりだし関数 getd の修正も必要であった。

上に要した変更行数を表 3 に示す。UtiLisp/C は C 言語により記述され、全体の行数は約 9300 行である。使用した計算機は、SparcStation 1+ である。

5.2 オーバヘッドの計測

5.2.1 gc 所要時間

ヒープ中に wp が存在する場合の gc の所要時間を測定した（表 4）。第 3.4.2 項での検討どおり、wp 個数に比例する時間がかかっていることがわかる。

表 2 新設の組込関数

Table 2 Functions introduced for wp operation.

名 称	機 能
wp	wp オブジェクトの生成
wpref	wp からの参照
wpset	wp の参照先の変更
wp(get/set)reset	リセット値の参照、変更
wp(get/set)strength	強さの参照、変更
wp(get/set)counter	カウンタの参照、変更
wpsetgcstrength	gc の強さの変更

表 3 wp 導入のための変更行数

Table 3 No. of source lines added for wp.

wp (関数新設)	148
wp (gc 修正)	74
wp (その他)	51
定義 wp 不可視	12
合 計	285

表 4 wp 個数と gc 所要時間の関係
Table 4 Relation between no. of wp's and gc-time.
(gc 100 回あたりの所要 CPU 時間 (1/60 sec))

wp 個数	総所要時間	wp 1 個あたり
0	363	—
100	376	0.13
1000	467	0.104
10000	1392	0.103

表 5 定義フィールド不可視化のオーバーヘッド
Table 5 Runtime overhead for invisible wp.

実験種別	(1)	(2)	(3)
所要時間 (1/60 sec)	1072	1082	1112
比率	(100)	(100.9)	(103.7)

5.2.2 定義フィールド wp 不可視化

自動アンロード機構をささえているのが、第 3.2 節で述べた関数定義における wp の不可視化である。

UtiLisp/C における実現では、関数 eval 内でシンボルから定義をとり出しそのデータ型判別をする部分の最後 (エラーを起こす直前) に wp の場合を追加したため、通常の (wp を経由しない) 関数の起動にはオーバーヘッドはない。

ここでは実際に wp を経由して関数を起動する場合のオーバーヘッドを測定した。

実験は関数 (tarai 10 5 0) を利用し、(1) wp を使用しない場合、(2) tarai の定義だけを wp 経由にした場合、(3) tarai 内部で呼び出される組込関数 (>, 1-) も wp 経由にした場合の 3 通りを比較した。

実行結果を表 5 に示す。最悪と考えられる (3) の場合でも効率悪化が 4 % 程度であり、十分実用にたえることを示している。現実的には組込関数をアンロードする必要はなく、(2) の場合が一般的であろう。その場合には効率悪化は 1 % 程度になっている。

5.3 自動アンロード機構によるヒープ占有量の変化

第 4 章で述べた自動アンロード機構をライブラリ関数に適用し、その効果を測定した。測定は、(1) 自動アンロード機構不使用、(2) 使用、のふたつの場合を行い、それぞれでのヒープ占有量を計測した。実験対象としたライブラリは、プリティプリンタ (prind) と prolog インタプリタ (Prolog/KR) の 2 種である。結果を表 6 に示す。

prolog については、起動によってデータの初期化が行われるため、起動する前と起動後に分けてデー

表 6 自動アンロードによるメモリ占有量の効果
Table 6 Heap usage spared by auto-unloading.

カッコ内は、(1) を 100 とした比率。

自動アンロード	(1) 不使用	(2) 使用	
		リセット前	リセット後
prind	15572 (100)	17796 (114)	3748 (24)
prolog (未起動)	109132 (100)	121132 (111)	34196 (31)
prolog (起動後)	141828 (100)	154588 (109)	88332 (62)
起動前後の差	32696	33456	54136

タを採用した。prind, prolog (未起動) いずれについても、wp を利用することで常駐部分を 30 % 程度にまで減少できることが示されている。

ところで、prolog の起動前後の占有量を比較すると (表の最下段)、不使用とリセット前についてはほぼ同量 (約 33 KB) の増加がみられる。これは初期化により生成される内部データ量に対応する。ところがリセット後の残量の増加はそれよりもずっと大きい。この原因は、初期化の過程で関数定義の一部がシンボルの属性リストに設定されることに起因する。(Prolog/KR の組込述語の処理関数が同名のシンボルの属性を利用して保持されているのである。) その結果として wp を経由しない参照が増えて、回収率が悪化するのである。このような場合は、属性からの参照を wp 経由とし、明示的に参照するようにプログラムすれば問題はなくなる。

6. 議 論

6.1 減衰の速度について

減衰 wp の減衰速度は、gc の回数に比例する。一方、gc の頻度は、セル消費量に比例し、ヒープの余裕に反比例する。したがって、減衰 wp の減衰速度はメモリ要求がきびしい場合に高速になり、非常に望ましいといえる。

これによって、メモリに余裕がある場合にはキャッシュとして長時間利用が可能になり、不要な再ロードを回避できる。

6.2 変更を受けたデータの書き出しについて

外部からキャッシュしたデータが変更を受けた場合、それを捨てる前に書き出す必要がある。しかし、これを現在の wp で単純に実現するのは難しい。gc 中にポインタのリセットのかわりに書き出しのための式を

実行すればよいのだが、それは gc が完了するまでは実行できない。

改善案としては、まず wp オブジェクト中のリセット値フィールドの解釈を変更し、参照先を捨てるために実行すべき式を格納することにする。gc 中にはリセットすべき wp をリストに収集するにとどめ、gc 終了後にこのリストを走査してリセット値フィールドを実行すればよい。(その場合、その wp オブジェクトを引数として与える仕様が適当であろう。) キャッシュの書き出しの例では、その段階で参照先の変更を調べ、外部に書き出せばよい。これはリセットだけではなく任意の操作を許す、より一般的な wp ということになる。

7. おわりに

拡張 wp を提案し、それを Lisp に導入することでヒープの有効利用に関して多くの機能が実現できることを示した。さらに拡張 wp を実装することにより、コストの面でも問題がないことを実証した。

今後はさらに高い機能を持つ wp の実装を行うとともに、Emacs lisp などへの実装を通して有効性を確認していく必要がある。

参 考 文 献

- 1) Stallman, R.M.: *GNU Emacs Manual*, Free Software Foundation, Cambridge, MA (1985).
- 2) Lieberman, H. and Hewitt, C.: *A Real-Time*

Garbage Collector Based on the Lifetimes of Objects, *CACM*, Vol. 26, No. 6, pp. 419-429 (1983).

- 3) Rees, J. A., Adams, N. I. and Meehan, J. R.: *THE T MANUAL*, Computer Science Department, Yale University (1988).
- 4) Hanson, C.: *MIT Scheme Reference Manual*, Edition 1.0 for Scheme Release, 7.1, MIT Scheme Team (1991).
- 5) 田中哲朗: SPARC の特徴を生かした UtiLisp/C の実現法, 情報処理学会論文誌, Vol. 32, No. 5, pp. 684-690 (1991).

(平成 4 年 3 月 3 日受付)

(平成 5 年 5 月 12 日採録)



寺田 実 (正会員)

1959 年生。1981 年東京大学工学部計数工学科卒業。1983 年同大学院工学系研究科情報工学修士課程修了。同大学計数工学科助手、電気通信大学電子情報学科助手を経て 1991 年東京大学工学部機械情報工学科講師、1992 年同大学助教授。工学博士。主な研究分野は Lisp を中心とするプログラム言語処理系で、ごみ集め、並行動作記述などを扱っている。オペレーティングシステム、オブジェクト指向にも関心を持つ。最近は分散処理のためのプロセス間通信の研究も行っている。ソフトウェア科学会、ACM 各会員。