

Weak Pointer の拡張と実装

寺田 実

東京大学工学部機械情報工学科

計算機は情報を捨てるのが苦手である。Lisp において、Weak Pointer (wp) というデータ型を利用すれば、ごみ集めの協力のもとに利用されなくなったデータを捨てることができる。本研究では、この wp の拡張を提案する。すなわち、リセット値を持つ wp、強さを持つ wp、減衰する wp の3つである。これにより、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。次に、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した。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なった。

EXTENSION AND IMPLEMENTATION OF WEAK POINTER

Minoru TERADA

Department of Mechano-Informatics, The University of Tokyo

7-3-1 Hongo, Bunkyo-Ku, Tokyo 113, Japan

It is difficult for computers to discard unused information. In Lisp, a data type known as Weak Pointer can release pointer it is holding by the help of garbage collector. We propose three extensions to the 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 measured its overhead and effectiveness.

1 はじめに

計算機は一般に憶えるのは得意だが、忘れるのは苦手である。その一例を、Lisp を例にとって示そう。Lisp で利用される技法のひとつに関数定義の自動ロードがある。これは、システムを起動した段階ではライブラリ関数をロードすることはせず、その関数が呼出された段階で実際のロードを行なうというものである。関数 f の定義がファイル "f.l" 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：¹

```
(defun f(arg) (exfile "f.l") (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。

ところで、関数 f がもはや不要になったとき、これを自動的に捨てることは難しい。必要になったことは f が呼出されたことによってわかるが、不要になったことは、呼出されなくなって久しいという形でわからないからである。

Lisp が対話的利用の基本ツールとして長時間にわたり利用されるようになると (例えば emacs), ユーザが行なう種々の作業 (メール、ニュース、かな漢字変換) によって大量の関数定義などがヒープ中に残されてしまい、貴重なメモリを浪費することになる。

それでも計算機の資源が有限である以上、どこかで情報を捨てる必要が生じる。現在それを担当しているのは、オペレーティングシステムのメモリ管理部分であろう。デマンドページング方式による仮想記憶では、実メモリ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なわれる。一方それとは独立に実メモリを走査しているプロセス (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 のような機能を Lisp でも実現するものである。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、ガーベジコレクション (以後 gc と呼ぶ) 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。

¹この例は UtiLisp を用いてある。exfile はファイルの内容を評価する関数である。

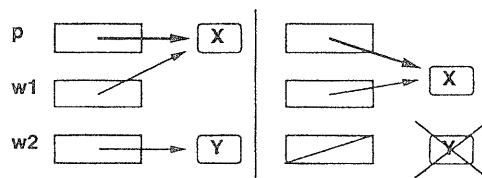


図 1: wp の動作

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。

本報告の構成は、第2節で wp について解説し、第3節ではその拡張を提案する。第4節では 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5節では、拡張 wp を利用したアプリケーションの例を示す。第6節では、拡張 wp の UtiLisp/C[2] 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7節では、実現法や問題点について議論を行なう。

2 wp とは

Generational gc に関する論文 [1] 中に、以下のような wp への言及がある：

通常の場合、オブジェクトへのポインタの存在は、そのオブジェクトが必要であることを示し、ポインタがない場合に限って gc はそれを回収できる。だが、Lisp 処理系によっては別種のポインタを許す場合がある。これが wp であり、参照先のオブジェクトを gc から保護しないという特徴をもつ。

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、状況は右のようになる。 X は生存し、 $w1, p$ とも 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 に対する参照を失ない nil 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。

このような wp の利用法として、[1] にあげてあるのは、

1. 現在使用中の(ある種の)オブジェクト全部のリスト
通常のポインタで指した場合には、使用中でなくなってもごみにならず、回収されない。

2. 冗長性を持つ逆ポインタ

例えば親子関係において、子から親へのポインタなど。

のふたつである。

しかし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のポインタからも参照を受けるので、ごみになる危険性がない。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 でその情報が捨てられることになる。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。

3 wp の拡張

本節では、wp オブジェクトを3つの方法で拡張し、より高い機能を与える。

3.1 リセット値つきの wp

これまでの wp の説明では、参照対象を失なった wp は nil にリセットされるとしてきた。しかし、個々の wp オブジェクトの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引数として指定すればよいのである。

5節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。

3.2 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。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回収する。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 が回収されることになる。

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) 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、ヒープ不足などの致命的エラーが起きた場合には強さ 0 の最強 gc を行な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。

3.3 減衰する wp

wp のリセットについて、すぐ次の gc でリセットするのではなく、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 ごとにカウンタをもたせ、gc が起きるたびにそれをへらしていく。カウンタが 0 になった段階でその wp は本来の wp となり、回収の対象となる。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 ですぐに回収の対象になる。

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。

3.4 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

- (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、ただちに回収の対象となる。

- (wp の強さ = gc の強さ) 減衰カウンタを 1 減じて、0 になった場合のみ回収の対象とする。
- (wp の強さ > gc の強さ) 減衰カウンタもへらさず、回収の対象にもしない。

4 wp の実現方式

4.1 wp オブジェクトの導入

[1]にあるような拡張のない wp の実装は、あるポインタが wp であるかどうかの判定だけでできればよいので、ポインタに 1 ビットの場所を確保できれば可能である。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、前節で述べた拡張をほどこした 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なった場合には、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) のリセット値にリセットされる。(3) はこの wp の強さを表わす整数値を持つ。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) のフィールドのリセットが起こる。(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 / 更新関数がある。

使用の具体例をあげる：

```
(setq A (wp (cons 1 2)))
```

コンスセルに対する wp の生成

```
(setq B (wpref A))
```

通常ポインタに変換してコピー

```
(car B)
```

利用時は通常と同じ

```
(setq C A)
```

wp のままコピー

```
(car (wpref C))
```

利用時に参照先をたぐる

4.2 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 (前項の例では wpref) を明示的に呼出す必要がある。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なうのが望ましい。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修正の二点に分けて論じる。

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。

処理系への修正の観点からは、ポインタ参照を行なう多くの場所で前述の判定と参照が必要となり、処理系の複雑化、巨大化をまねがれない。

(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、これらの問題点はずっと軽減される。例えば、gc アルゴリズムによっては不可視ポインタを要するものがある。この場合、不可視ポインタかどうかの判定は既に実装されているため、不可視 wp を導入しても通常のポインタ操作の効率悪化にはつながらない。)

この問題点を回避するために本研究で採用した方法は、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通り：

- (1) 処理系への修正がきわめて限定できる。(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。

ここで(3)について説明する。データのキャッシュとしての利用など、値フィールドからの参照の場合、アクセスのための専用関数を用意することが可能であるから、それらの中で明示的な判定と wp 参照を行なうことはそれほど困難ではない。これに対して、定義フィールドは関数呼出しによって直接利用される。明示的な判定と wp 参照のためには関数の形を書き換える必要が生じ、コストは非常に大きくなる。例をあげると、関数 f を wp を経由して定義するには、

```
(putd 'f (wp (function(lambda()(g)))))
```

とすることになる。ここで、定義フィールドの wp が不可視であれば

```
(f)
```

により正しく呼出される。

一方、不可視でない場合には、

```
(funcall (wpref (getd 'f)))
```

とすることがある。さらに、 f の定義が wp 経由でない可能性がある場合には、その判定が必要になり、

```
(let ((def (getd 'f)))  
  (cond ((wpp def) (funcall (wpref def)))  
        (t (funcall def))))
```

と複雑化する。(この変更を f を呼出すすべての場所で行なうことに注意。あるいは、中間関数を準備して、これを介在させることもできるが、いずれにせよ実行コストの増大はまぬがれない。)

ところで、不可視 wp への代入については、二通りの方式が考えられる。ひとつは wp の値フィールドに代入するもの(図 2(a))で、もうひとつは (wp を指している) 定義フィールドに代入するもの(図 2(b))である。

(a) の方法では、一度 wp 経由となった定義フィールドは、それ以降何を代入しても wp 経由のままとなり、不注意によってデータを失う可能性がある。(b) の方法はそれに比べて安全であり、本実装ではこちらを採用した。(ただし、(b) 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。)

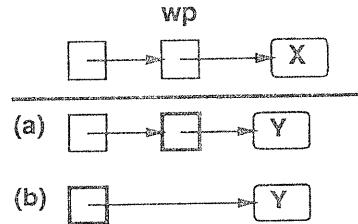


図 2: 不可視ポインタへの代入のふたつの方式

4.3 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。

4.3.1 マークスweep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「生きている wp オブジェクトリスト」に登録する。(このために wp オブジェクトに作業領域が 1 語必要となる。)

マーキングフェーズ終了後、sweep フェーズに移るまえに、「生きている wp オブジェクトリスト」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。

sweep フェーズは wp オブジェクトも含めて普通と同様に行なう。

4.3.2 コピー方式 gc への組込

コピー中に wp オブジェクトを訪問した時、以下を順に行なう:

- (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なわない
- (4) 「生きている wp オブジェクトリスト」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「生きている wp オブジェクトリスト」を走査し、各(旧)wp オブジェクトに対して

- (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、ポインタを更新し、
- (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「生きている wp オブジェクトリスト」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 本以上の大きさをもっていれば、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な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「生きている wp オブジェクトリスト」の走査が増えるだけであり、wp の個数の比例した増分となる。

5 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出した場合に、wp オブジェクトからもう一段参照を行なう必要があるが、それをユーザが検出するの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、4.2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 を不可視にすることにした。

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ベ

◦ 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 0

図 3: 自動アンロードのための定義と実行例

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 3 に示す。スペシャル変数 `*reload*` には、再ロードのための式 `(exfile "foo.l")` が関数 `exfile` によってバインドされる。減衰カウンタは 3、強さは 0 に設定してある。

表 1: wp 導入のための変更行数

wp (subr 新設)	148
wp (gc 修正)	74
wp (その他)	51
定義 wp 不可視	12
合計	285

6 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。

6.1 処理系への変更点

実装方式としては、4節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 (subr) を 10 個定義した。

新設の subr は以下の通り:

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

UtiLisp/C の gc はコピー方式であるので、4.3.2節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 1 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。

6.2 オーバヘッド

6.2.1 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 2)。4.3.2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: wp 個数と gc 所要時間の関係

(gc100 回あたりの所要 CPU 時間 (1/60sec))		
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 3: 定義フィールド不可視化のオーバーヘッド

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

6.2.2 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、4.2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だ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 3 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いはずであり、(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。

6.3 自動アンロード機構によるヒープ占有量の変化

5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ない、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、

表 4: 自動アンロードによるメモリ占有量の効果

カッコ内は, (1) に対する比率.

自動アンロード	(1) 不使用	(2) 使用	
		回収前	回収後
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

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 4 に示す.

prolog については, 起動によってデータの初期化が行なわれるため, 起動する前と起動後とに分けてデータを採取した. 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, (1) 不使用と (2) 回収前についてはほぼ同量 (約 33KB) 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 wp 回収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ふえて, 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.

7 議論

7.1 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るというとても望ましい性質を持つ.

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.

7.2 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評価すればよいのだが, 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, リセットすべき wp を gc 中にはリセットせずにリストに収集して, gc 終了後にそれらを順に書き出せばよい. (この場合, 書き出すための式も wp の 1 フィールドとして保持する.) その意味で, 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.

8 おわりに

Lisp に拡張 wp を導入することで, 非常に少ないコストでさまざまな機能が実現できることを示した.

今後はさらに高い機能を持つ wp の実装を行なうとともに, 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.

東大工学部計数工学科和田研究室のみなさんに, UtilLisp/C について協力をいただきました.

参考文献

- [1] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," CACM, Vol. 26, No. 6, pp. 419-429 (1983).
- [2] 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).