Scheme 処理系における C 言語拡張コードへのライトバリア自動挿入

花井 亮[†] 小宮 常康[†]
八杉 昌宏[†] 湯淺 太一[†]

スナップショットごみ集め（GC）では，GC の対象になるヒープにあるオブジェクト間のポインタが置き換えられる場合に，ライトバリアの処理が必要である．一方，C 言語に対するインタフェースを備えた Scheme 処理系は，そのインタフェースを用いてユーザが処理系を拡張することができる．その処理系がスナップショット GC を実装する場合，拡張した部分においてもポインタの置き換えが行われるため，拡張した C 言語コードにライトバリアを挿入する必要がある．しかし，ユーザが明示的にライトバリアを挿入するのは煩わしい．そこで，本論文ではプリプロセッサによって C 言語のソースコードにライトバリアを自動挿入する方法を提案する．提案方式では，プリプロセッサに対して Scheme オブジェクトへのポインタを格納する可能性のある左辺値の型を指示するための構文を C 言語に追加し，プリプロセッサは代入の左辺値の型が，ユーザによって指示された型であればライトバリアを挿入する．また，実装したプリプロセッサを用いて，提案方式を実際に C 言語ソースプログラムに適用した場合にどのくらいのライトバリアが挿入されるかを評価した．

Automatic Insertion of Write Barriers into C-based Extension Code for Scheme Systems

Ryo Hanai,[†] Tsuneyasu Komiya,[†] Masahiro Yasugi[†] and Taiichi Yuasa[†]

The snapshot GC algorithm requires a write barrier operation when a pointer between objects in the garbage-collected heap is overwritten. On the other hand, a Scheme system with a C language interface allows the user to extend the system by using the interface. If the Scheme system has the snapshot GC, write barriers need to be inserted into C-based extension code since pointers are overwritten by that extension. However, it is troublesome for the users to insert write-barriers manually. In this paper, we propose a method to automatically insert write barriers into a C source program using a preprocessor. In our method, we add to C language a new construct so that the users declare the type of an lvalue which may hold pointers to Scheme objects. The preprocessor inserts a write barrier if the type of the lvalue of an assignment is specified by the construct. We have implemented the preprocessor and evaluated how many write barriers are actually inserted into some C source program when we use this method.

1. はじめに

Scheme[4] 処理系がサポートしていないプラットフォーム依存の機能を利用したり，既存のライブラリを利用したりするなどの目的で，C 言語によって処理系を拡張するためのインタフェースを備えている Scheme 処理系が多い．これらの処理系のロボット制御などの分野での利用を考えた場合，C 言語拡張部分も含めて実時間システムとなることが望ましい．そのためには，C 言語拡張部分から Scheme のオブジェクトに対する参照を考慮したごみ集め（GC）が不可欠である．さらに，実時間システムを実現するためにはその GC がプログラムの一定以上の停止時間をともなわない GC，すなわち実時間 GC である必要がある．

実時間 GC は GC の一連の処理を小さな部分処理に細分化し，プログラムの実行と並行して GC の処理を少しずつ進行させる手法である．ただし実時間 GC では GC 処理中にもプログラムの実行が継続するため，GC 処理中のヒープのリスト構造の変化に対応する必要がある．

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

```
Scheme_Object *my_setcar_prim(int argc,
                              Scheme_Object **argv){
  SCHEME_CAR(argv[0]) = argv[1];
  return scheme_void;
}
```

図 1 MzScheme の拡張コード例
Fig. 1 An example of MzScheme extension.

これまでに提案された実時間 GC のアルゴリズムのうち，Yuasa の考案したスナップショット GC[8] は，マーク&スイープ法を基礎とし，オブジェクトへの参照が変更される場合にのみ変更前の参照を保存する方式によって GC 処理中のヒープのリスト構造の変化に対処する（このように参照が変更される場合に何らかの処理を行う方式をライトバリアと呼ぶ）．しかし，ルート集合（レジスタやスタックなどシステムがいつでも参照することのできるデータ領域）は頻繁に書き換えられるので，スナップショット GC ではマーク操作を行う前にルート集合のスナップショットを撮る．すなわち，ルート集合から直接指されているすべてのオブジェクトに対して一括してマークをする．この処理をルート挿入と呼ぶ．これによりライトバリアの必要箇所はヒープオブジェクト間のポインタの変更に限定される．

このスナップショット GC を，保守的 GC を採用し，C 言語による拡張機能を備えた Scheme 処理系に実装することを考える．保守的 GC を採用する処理系では，C 言語拡張部分から特別なインタフェースを通さず，直接ポインタを使って Scheme 処理系内部のオブジェクトに書き込みを行うことができる．このような処理系の場合，拡張コードにおいてもヒープオブジェクト間のポインタの変更が行われる．このような処理系の例として MzScheme[5),6)] があげられる．**図 1** は MzScheme における C 言語拡張コードの例である．`SCHEME_CAR` は構造体 `Scheme_Object` のフィールドに展開されるマクロである．この例では `argv[0]` が指していたオブジェクトのフィールドが書き換えられる．しかし，ユーザがソースコードに明示的にライトバリアを挿入するのは煩わしい．そこで本論文では，ポインタかどうかを保守的に判断するスナップショットごみ集めを実装する処理系を対象として，C 言語拡張コードにライトバリアを自動挿入する方法を提案する．

保守的 GC を行っている処理系ではポインタでない値にライトバリアを保守的に挿入してもプログラムの実行に支障はないが，プログラムの実行に対するオーバヘッドとなるため，できるだけ不必要なライトバリア挿入は避けたい．そこで，C 言語における型がライトバリアの挿入箇所の判別の大きな手がかりとなることに注目する．提案方式では，GC 処理の対象となるヒープに存在するオブジェクト間のポインタとして使用される値を格納する可能性のある型はどの型かについて，ユーザから支援を受けてライトバリアの挿入箇所を判別する．

本論文では以下に 2 章で提案方式について説明し，3 章で Scheme 処理系 MzScheme における提案方式の実装について説明する．その後 4 章で評価をした後，5 章で議論を行い，6 章で関連研究について比較，検討し，最後に 7 章でまとめを行う．

2. 提案方式

2.1 ライトバリアの必要箇所

提案方式では，C 言語の制御スタックをルート集合とし，保守的（ポインタの可能性があるものはポインタと思って）な GC を基にしたスナップショット GC を実装するシステムを前提とする．スナップショット方式の GC ではルートからの参照はルート挿入時にスナップショットを撮る．すなわち，ルートから参照されているオブジェクトは一括してマークされる．したがって，このようなシステムにおいてライトバリアが必要となるのは Scheme のヒープに存在するオブジェクト（Scheme オブジェクトと呼ぶ．**図 2** のグレーのオブジェクト）間のポインタが変更される箇所だけである．これらのポインタは太線の矢印で表されている．C 言語のコードでは Scheme のヒープとは別にユーザが明示的に割り当て，解放を行うヒープ（図 2 の malloced heap）を利用することができるが，ここに確保されたオブジェクト間のポインタが変更されるような場合はライトバリアは必要ない．保守的 GC を行うシステムの多くでは malloced heap から Scheme heap へのポインタは GC によって参照と見なされない．したがって，malloced heap から Scheme heap へのポインタが変更される場合にもライトバリアは必要ない．

ライトバリアが必要な箇所は C の記述では，ソースコード中に現れる代入式（正確には左辺値の値が書き換えられる記述）のうち，

(1) 破壊される値は Scheme のヒープにあった，
(2) 破壊される値は Scheme オブジェクトへのポインタとして使われていた，

が実行時に同時に満たされる可能性がある箇所である．ただし，Scheme オブジェクトの配列を指すポインタをインクリメントし，アクセスはしないが配列の末尾を越えた位置をポインタが指すような処理を行うことがあ

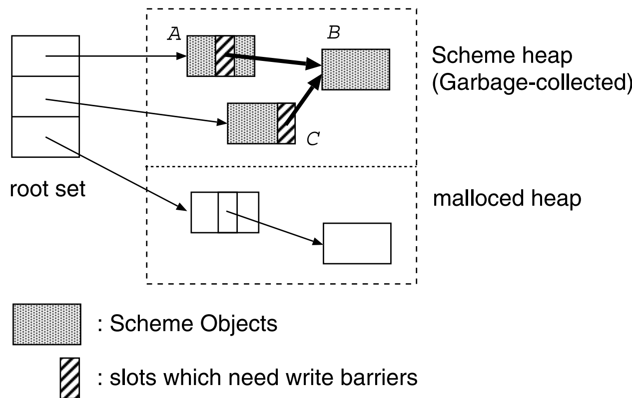

図 2 ヒープ
Fig. 2 Heap.

るので代入式には ⟨ 左辺値（2.2 節参照）⟩=⟨ 式 ⟩ で表される通常の代入式だけでなく，演算子 `+=`，`-=`，`++`，`--` などを用いて値を変更する式も含める．

これら以外の箇所にライトバリアを挿入したとしても，保守的 GC を行う処理系では本来なら回収されるはずのオブジェクトが回収されずに残る可能性があるだけで，プログラムの実行に支障はない．しかし，必要のないライトバリアの処理はプログラムの実行に対するオーバヘッドとなる．したがって，必要な箇所にライトバリアを挿入したうえで，不必要な箇所にはできるだけ挿入しないようにすることが求められる．

2.2 左辺値の型による判定

提案方式ではライトバリアの必要箇所の判断基準として Scheme オブジェクトへのポインタを格納する可能性のある左辺値の型を用い，ライトバリアの挿入を行う．

一般に Scheme オブジェクトを C 言語で扱うときの型は決まっている．したがって，代入式の左辺値に注目することにより，破壊される値が Scheme オブジェクトへのポインタとして使われていたかどうかによるライトバリア挿入箇所の絞り込みができる．

C 言語ではキャストによって任意のポインタ型間での型変換が可能である．ポインタをそれを保持するのに十分大きい整数型に変換することも可能である．したがって，ユーザのプログラミングのスタイルしだいで，Scheme オブジェクトへのポインタが破壊される場合でも左辺値の型が指示された型にならず，ライトバリアの挿入漏れが生じる可能性がある．そのような場合はユーザ自身でライトバリアのコードを記述して対処する必要がある．しかし，通常の拡張コードのプログラミングにおいてはそのような場合は稀であると考えられるうえ，仮にそのようなコードをユーザが

register-directive:
　　`SCM_PTR_TYPE (` *type-name* `) ;`
（例）`SCM_PTR_TYPE ( Scheme_Object * );`

図 3 型登録ディレクティブ
Fig. 3 Directive to register a type.

記述したとしても，ライトバリアが挿入される型が決まっているので，ユーザは対処が必要であることを容易に判別できる．

また，Scheme オブジェクトへのポインタを格納する可能性のある型はユーザが後述のプリプロセッサに指示するものとするが，それ自体はユーザにとってたいした労力にはならないと考えられる．

これらの理由で型によるライトバリア挿入箇所の判定には妥当性があると考えられる．

2.3 プリプロセッサに対する指示構文

提案方式では，プリプロセッサに Scheme オブジェクトへのポインタを格納する可能性がある型を登録する構文（**図 3** 参照）を C 言語に追加する．ユーザはこの構文を使い，Scheme オブジェクトへのポインタを格納する可能性がある型を C 言語ソースファイル中に記述する．構文はトップ・レベルで使用するものとし，型の登録は登録した位置からそのソースファイルの最後まで有効とする．ユーザはこの構文を複数回使用して，複数の型を登録してよい．

2.4 ライトバリア挿入箇所の判定

プリプロセッサはソースファイル中に記述されたユーザからの指示に従い代入式に対して以下のチェックを行う．

まず，それぞれの式に対して左辺値の型を調べる．左辺値の型がユーザから指示された型の中になければ，その式において Scheme オブジェクトへのポインタが

```
l_expr op r_expr
    ⟹ { tmp = &l_expr, GC_push(*tmp), *tmp op r_expr
        { tmp = expr, GC_push(*tmp), *tmp op r_expr      (l_exprが*exprの場合)
  ( op: =|+=|-= )
l_expr op
    ⟹ { tmp = &l_expr, GC_push(*tmp), *tmp op
        { tmp = expr, GC_push(*tmp), *tmp op      (l_exprが*exprの場合)
  ( op: ++|-- )
```

図 4　コード変換
Fig. 4　Code translation.

変更されることはないと判断し，ライトバリアの挿入は行わない．左辺値の型がユーザから指示された型の中にあった場合，左辺式に間接演算子（*），ポインタ演算子（->），配列参照演算子（[]）のいずれかを含むかどうかをチェックする．含む場合にはその式をライトバリアを挿入した形に書き換える．いずれも含まない場合は，変更される値はスタックや静的に確保された変数領域にありヒープへの書き込みである可能性はないのでライトバリアの挿入は行わない．

具体的なコード変換については 3.2 節で説明する．

3. MzScheme における実装方法

保守的 GC を行う Scheme 処理系 MzScheme にスナップショット GC を実装したものとして，提案方式を実現するには以下の実装が必要となる．

- ライトバリアを挿入するプリプロセッサの実装．
- MzScheme の GC に依存するライトバリアの実装．

3.1 MzScheme の GC と C 言語インタフェース

MzScheme では Scheme の値は先頭のフィールドとして整数（short）のタグを持つ構造体 `Scheme_Object` へのポインタで表現される．先頭のフィールドに同様のタグを持つものはすべて `Scheme_Object` となりうる．C 言語拡張コードを書くときは Scheme 処理系との Scheme の値の受け渡しに型 `Scheme_Object *` を利用する．MzScheme はメモリ割当てを使用するコードからの有意義な協力がない環境下でメモリ割当てと GC を行うために設計された Boehm-Demers-Weiser GC [1),3)] を利用して，Scheme オブジェクトも処理系が使うデータもまとめてごみ集めを行う．いい換えると，MzScheme は Boehm-Demers-Weiser GC ライブラリを利用したアプリケーションであるといえる．具体的には，GC は C スタック，静的に確保された変数領域，レジスタをルート集合とし，そこからの参照を保守的に（ポインタと思える値はポインタと思って）たどり到達可能なオブジェクトすべてにマークをつけ，マークのつかなかったオブジェクトを回収，再利用する．GC は OS から割り当てられたメモリ領域を同じサイズの複数のブロックに分割し，1 つのブロックには同じサイズのオブジェクトだけを割り付けるため，コンパクションのためにデータを移動させることはない．このような GC を備える MzScheme では C 言語拡張コードにおいて，Scheme オブジェクトであることを特に意識せずに Scheme オブジェクトにポインタを使って直接アクセスすることができる．また，ユーザは新たに Scheme の型を定義することができる．定義した型のオブジェクトに対するメモリ領域の割当てには，通常の `malloc` と同様にサイズを指定して領域を確保する関数 `scheme_malloc(size_t n)` を用いる．定義した型のオブジェクトの先頭には，システムによって割り当てられたタグを持たせる必要がある．

3.2 プリプロセッサのコード変換

実装したプリプロセッサの具体的なコード変換について説明する．2.4 節で述べたチェックによりライトバリアが必要と判断された代入式を，

- 代入式に含まれる副作用のある式は 1 回だけ評価する，
- 代入式自身の値を変えない，

ことに注意し，以下の処理を行うコードに変換する（**図 4** 参照）☆．

(1)　一時変数 `tmp` に左辺値のアドレスを保存する．
(2)　ライトバリアの処理を行う（`GC_push`）．
(3)　代入を行う．

実際に，図 1 のコードに対してコード変換を行った例を示す（**図 5**）．

3.3 ライトバリアの実装

ライトバリアの実際の処理は `GC_push(*tmp)` で行われる．MzScheme にスナップショット GC を実装した場合のライトバリアの処理の流れは以下のようになる．

☆ C 言語のコンマ演算子で区切られた式は，関数引数とは異なり左から右に評価される．

表 1　MzScheme へのライトバリア挿入
Table 1　Insertion of write barriers into MzScheme.

	barrier	assign	source lines	barrier/100 lines	assign/barrier
組込み関数	431	3,394	19,849	2.17	7.87
組込み関数（算術）	42	1,384	5,364	0.79	33.0
評価系	455	2,576	8,818	5.16	5.66
例外処理	88	567	1,692	5.20	6.44
オブジェクト指向拡張	233	1,177	3,169	7.35	5.05
メモリ割当て	26	74	1,343	1.94	2.85
型のタグ，印字表現	94	112	279	33.7	1.19
その他	269	845	6,240	4.31	3.14
計	1,638	10,129	46,754	3.50	6.18

```
Scheme_Object *my_setcar_prim(int argc,
                        Scheme_Object **argv){
  struct Scheme_Object **_wb_tmp1;
  _wb_tmp1=&(*(argv+0))->u.pair_val.car,
    GC_push(*_wb_tmp1), *_wb_tmp1=*(argv+1);
  return scheme_void;
}
```

図 5　コード変換の例
Fig. 5　An example of code translation.

(1)　GC がマーク処理中かどうかをチェックする．
(2)　`*tmp` の値が GC が管理するヒープのアドレスとして有効な範囲かどうかをチェックする．
(3)　`*tmp` の値から GC が管理するヒープブロックのデスクリプタを取得する．
(4)　デスクリプタからポインタの指すオブジェクトの先頭アドレスを求める．
(5)　そのオブジェクトのマークビットがセットされていなければビットを立てマークスタックに積む．

4. 評　　価

実際にどれくらいのライトバリアが挿入されるかを評価するために，NCX [9] のパーサを基に，Common Lispで実装したプリプロセッサを用いて MzScheme のソースコードへライトバリア挿入を行った．MzScheme のソースコードは拡張コードとはいえないが，組込み関数の実装部分などソースコードの大部分は拡張コードを書く場合と記述の仕方が同じであるため，ライトバリアがどれくらい挿入されるかの評価に使える．ただし，MzScheme は Boehm-Demers-Weiser GC ライブラリを利用したアプリケーションの形で書かれており，C 言語の文字列なども GC ライブラリが提供するメモリ割当てルーチンを利用して確保されている．そのため `char` などの基本型に対するポインタ型も GC 処理の対象となる領域のオブジェクトを参照するために使われている．

したがって，ライトバリア挿入箇所の判別に用いるポインタ型としてすべてのポインタ型をプリプロセッサに指示して評価を行った．その結果を**表 1** に示す．表の縦方向はそれぞれのソースコードで実装されている機能別に分類したものである．横方向は順に挿入されたライトバリアの個数，代入（インクリメント，デクリメントを含む）の個数，コメントや空行を除いたソースコードの行数，ソースコード 100 行あたりの挿入されたライトバリアの個数，ライトバリア 1 個あたりの代入の個数をそれぞれ表す．

ポインタを使った読み書きが少ない算術関数ではライトバリアの挿入頻度が低く，ヒープに多数のオブジェクトを確保し頻繁にリスト構造の変更を行う評価系，例外処理などではライトバリアの挿入頻度が高くなっている．これは予想どおりの結果であるといえる．「型のタグ，印字表現」の箇所でライトバリアの頻度が異常に高くなっているのは，大きな `char *` の配列を固定文字列（各データの印字表現）のアドレスで初期化する関数が含まれるからである．しかし，この初期化は割り当てたばかりのオブジェクトを初期化しているだけなので本当はライトバリアは必要ない．このことから，割り当てた直後の領域の初期化であることが簡単に調べられる場合はライトバリアを省略することによってライトバリアの挿入箇所の最適化が行えることが分かる．もっとも，この例は固定文字列のテーブルを用意するコードなので 1 回しか実行されない．そのため，不必要なバリアが多数入ったところで性能への影響はたいしたことはないであろう．しかし，このような初期化の例は他にも随所で見られる．たとえば**図 6** の Scheme の cons セルを作り car 部，cdr 部を引数で初期化する関数の場合，cons セルを確保するたびに不必要なライトバリアのチェックが入ることになる．このライトバリアは性能にも影響すると考えられるので，できれば省略したい．`GC_malloc`，`GC_malloc_atomic`，`GC_malloc_stubborn` など，領域割当ての関数名が分かるものについては，それらの関数で割当てたメモリ領域を初期化する代入式は簡単

```
Scheme_Object *scheme_make_pair
     (Scheme_Object *car, Scheme_Object *cdr)
{
  Scheme_Object *cons;
  cons=(Scheme_Object *)GC_malloc(
                  sizeof(Scheme_Object));
  cons->type = scheme_pair_type;
  cons->u.pair_val.car = car;
  cons->u.pair_val.cdr = cdr;
  return cons;
}
```

図 6 ライトバリアの不要な例
Fig. 6 Needless write barriers.

なデータフロー解析で検出できる．簡単な解析を行った結果，MzScheme のコードでは，このような箇所がライトバリア挿入箇所全体の 16%近くあることが分かった．

代入全体にライトバリアを挿入する場合と比べると，ライトバリアの挿入箇所は約 1/6 に絞れている．組込み関数だけに注目すると約 1/8 になっている．

MzScheme のソースコードへのライトバリア挿入によって，全体では 46,754 行のソースコードに対して 1,638 カ所のライトバリア挿入が行われた．ライトバリアの挿入箇所を実際にチェックした結果，うえで議論した未初期化領域への書き込みの場合を除けば，GC 対象となるオブジェクトを指しえない箇所はわずかであった．

すなわち，実装したプリプロセッサを用いることにより 1,000 を超えるライトバリア挿入の労力を軽減できたことになる．また，プログラム本来のアルゴリズムと関係ないライトバリアのコードを明示的に記述する必要がないため，元のソースコードの可読性，再利用性も保たれる．

5. 議　　論

5.1 世代別 GC におけるライトバリア挿入

ライトバリアの自動挿入はインクリメンタル GC 以外にも応用できる．たとえば提案方式は，保守的 GC の世代別 GC 化にも利用できる．世代別 GC は「大多数のデータオブジェクトは短命である」という仮説に基づき，データオブジェクトを旧世代と新世代に分け，旧世代の回収作業を省略することで GC の処理時間を短縮させる方式である．旧世代をスキャンせずに済ませるため，通常，旧世代から新世代へのポインタを remembered set と呼ばれる場所に記録しておき，GC は remembered set が指す場所をルートと見なすという手法が用いられる．この手法を用いるためには，新たに旧世代から新世代へのポインタが生じるとき，このポインタを remembered set に追加しなければならない．そのため，旧世代オブジェクトへの書込みに対してライトバリアを設定し，書き込まれた場所を remembered set に登録するという処理を行う．

このライトバリアを挿入するためにも，提案方式が利用できる．ライトバリアが必要となる箇所は，旧世代オブジェクト中に新世代オブジェクトへのポインタが書き込まれる可能性がある代入である．実際には，旧世代から新世代へのポインタかどうか静的には分からないので，オブジェクト間のポインタを代入する箇所にはライトバリアが必要になる．したがって，ライトバリア挿入箇所の判定は提案方式がほぼそのまま利用できる．ライトバリアの処理（`GC_push(*tmp)`）の箇所を，「書き込まれた場所が旧世代オブジェクトであり，書き込まれた値が新世代オブジェクトへのポインタの可能性がある」かどうかをチェックし，remembered set に書き込まれた場所を追加する処理を行うコードに置き換えればよい．

5.2 型の等価性

ライトバリア挿入箇所の判別に左辺値の型を用いると述べたが，C 言語では `typedef` 指定子によって型式に名前をつけることができる．そのため，型の等価性に関して，名前等価か構造等価かという問題が生じる．すなわち，次のようなコードがある場合 `struct _Cell *` と `Cell *` は等価かどうかという問題である．

```
typedef struct _Cell{
   struct _Cell *next;
   int val;
} Cell;
```

等価でないとすると，`Cell *` をプリプロセッサに指示したつもりが上記の構造体のフィールド `next` は `struct _Cell *` 型で宣言されているためにライトバリアの対象にならなくなり，ライトバリアが挿入されない．これはユーザにとって分かりにくい．したがって，この場合は構造等価がよい．しかし，4 章のように `char` などの基本型に対するポインタ型も登録する必要がある場合，`char *` などに別名をつけてライトバリアが必要な型とそうでない型を区別できると都合がよい．したがって，この場合は名前等価がよい．

Scheme 処理系とのインタフェースを考える限り `char *` などを指示する必要性は少ないことを考えると構造等価が適しているといえる．両者の長所を取り入れると，`char *`，`int *` などを登録したい場合に備えて，`char`，`int` などの基本型に対するポインタ型については名前等価，それ以外の構造体や共用体に対するポインタ型は構造等価とすることも考えられる．

5.3 構造体，共用体への代入

代入式の左辺値の型が構造体あるいは共用体の場合，代入されるオブジェクトがユーザに指示されたポインタ型のフィールドを持っているならば，それらの代入に対してもライトバリアを挿入することが望ましい．対処法として，構造体や共用体が持つフィールドのうち指示された型のフィールドをすべて GC_push する方法が考えられる．

5.4 MzScheme 本体に対するスナップショット GC の実装

MzScheme にスナップショット GC を実装する場合，処理系にもライトバリアを入れる必要があるが，MzScheme 本体に対するライトバリアの実装にも提案方式のプリプロセッサが使えるため，スナップショット GC を実装する手間を大きく軽減することができる．しかし，MzScheme では memcpy，memset といった直接メモリ領域に書き込むライブラリ関数が使用されているため，これらを用いてポインタを含む領域を書きつぶしている箇所については，ライトバリアを考慮した別の関数に置き換えるなどの対策が必要となる．

5.5 C 言語でのスナップショット GC の利用

本論文では，Scheme 処理系の C 言語拡張コードという観点から話を進めてきたが，もともと Boehm-Demers-Weiser GC は C 言語からライブラリとして利用できる GC として設計されたものであるから，実装したプリプロセッサを用いれば，C 言語でスナップショット GC が利用できるのではないかと考えられる．ただし，C 言語で GC 処理の対象となるオブジェクトを格納する左辺値の型や，memcpy，memset などのライブラリ関数の利用には注意が必要となる．そう考えるとライトバリア（特に不必要なライトバリア）がプログラムの実行に対してどれくらいのオーバヘッドを課すのか，また，C 言語でスナップショット GC を利用した場合の性能，停止時間はどれくらいかを評価することも興味深いと思われる．

6. 関連研究

同様の C 言語に対するインタフェースを備えた処理系に guile[2] がある．guile では，cons cell に対する代入は専用のインタフェース（たとえば void SCM_SETCAR(SCM cell, SCM x)）を用いて行う．このように Scheme オブジェクト間のポインタを変更する代入のインタフェースを定めれば自動挿入の必要はない．これに対しライトバリアの自動挿入方式は CAR(cell)=x と代入式で記述できることにより，car ポインタを取り出して変数に保持したり別の関数に渡したりなど，より柔軟な記述ができるという利点を持つ．また，新たにユーザが Scheme の型を定義し，その型のオブジェクトに Scheme オブジェクトへのポインタを格納した場合，あるいは C 言語で使う構造体や配列などに Scheme オブジェクトへのポインタを格納した場合についてもライトバリアが自動挿入されるという利点も持つ．また，代入のインタフェースを定めている処理系にスナップショット GC を実装する場合には，処理系が保守的 GC を行っているという条件の下，処理系へのライトバリアの挿入とともにインタフェースの実装部分へのライトバリア挿入に提案方式のプリプロセッサを用いることができる．

他言語インタフェースというと JNI[7] を思いつかれるかもしれない．JNI の主な目的は所定のプラットフォーム上のすべての Java Virtual Machine の実装間で，ネイティブメソッドライブラリを完全なバイナリレベル互換とすることにあり，ネイティブコードが JVM のメモリ配置や GC の方式に依存することを避けるため，インタフェースは複雑なものとなっている．また，ネイティブコードから Java オブジェクトへ直接ポインタを使ってアクセスすることはできない．一方，MzScheme は保守的 GC を採用することにより，ユーザが Scheme のメモリ管理機構をあまり意識せずに使える簡素な C 言語インタフェースを提供している．そこで，MzScheme にスナップショット GC を実装するにあたり，より C 言語らしい自由な記述ができるよう，ライトバリアの挿入のために代入のインタフェースを定めることはせずにプリプロセッサによる自動挿入方式を採用した．

7. まとめ

本論文では，Scheme 処理系が提供する C 言語インタフェースを利用した拡張コードに対して，代入式の左辺値の型を利用してライトバリアを自動挿入する方法を提案した．MzScheme のように保守的 GC を採用することにより C 言語とのインタフェースが簡単になっている処理系にスナップショット GC を実装する場合，この方式を利用することにより，特別なインタフェースを用意することなく C 言語らしい記述のもと，ライトバリアの自動挿入を行うことができる．

参考文献

1) Boehm, H.-J. and Weiser, M.: Garbage collection in an uncooperative environment, *Software Practice and Experience*, Vol.18, No.8, pp.807–820 (1988).

2) Free Software Foundation: *Data Representation in Guile* (1999). http://www.gnu.org/software/guile/docs/data-rep.html
3) Jones, R. and Lins, R.: *Garbage Collection*, chapter 9.2, John Wiley & Sons (1996).
4) Kelsey, R., Clinger, W. and Rees, J. (Eds.): *Revised^5 Report on the Algorithmic Language Scheme* (1998).
5) Rice University, University of Utah: *Inside PLT MzScheme* (2000).
6) Rice University, University of Utah: *PLT MzScheme: Language Manual* (2000).
7) Sun Microsystems, Inc.: *Java Native Interface Specification* (1997). http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html
8) Yuasa, T.: Real-Time Garbage Collection on General-Purpose Machines, *The Journal of Systems and Software*, Vol.11, No.3, pp.181–198 (1990).
9) 湯淺太一ほか：超並列 C 言語 NCX 仕様書（Version 3）(1993).

(平成 14 年 9 月 30 日受付)
(平成 14 年 11 月 8 日採録)

花井　亮

1978 年生．2001 年京都大学理学部卒業．同年より同大学大学院情報学研究科修士課程に在学中．言語処理系に興味を持つ．

小宮　常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．1998〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

湯淺　太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著），「C 言語によるプログラミング入門」，「コンパイラ」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．