メモリ管理機能のモジュラーかつ効率的な実装手法

内 山 雄 司† 脇 田 建†

メモリ管理機能のモジュラーかつ効率的な実装手法を提案する．言語処理系の実装においてメモリ管理機能をモジュラーな設計とすることは，既存の処理系に新たなメモリ管理アルゴリズムを実装することを容易にする点で有用である．まず，メモリ管理機能のモジュラーな実装を可能とするために，本研究では，実行時システムとメモリ管理機能との間に抽象的なインタフェースを定義する．このインタフェースはプログラム実行時のメモリ操作を抽象化し，広範なメモリ管理アルゴリズムを抽象的に記述する手段を提供する．また，メモリ管理機能をモジュラーな実装とした処理系の高速化のために，本研究では，実行時システムの特化手法を提案する．一般に，インタフェースの導入による抽象化には，実装上のさまざまな工夫による高速化を妨げるという問題がある．提案する手法は，メモリ管理機能の仕様に基づいて実行時システムの実装を自動的に特殊化することで，モジュラーな実装にともなうオーバヘッドを解消する．本研究では，提案した手法に従って既存の処理系のメモリ管理機能を再実装し，ベンチマークテストを用いて，それぞれの実装での実行性能を比較した．その結果，本研究が提案した特化手法がモジュラーな実装にともなう 10%程度のオーバヘッドを解消し，既存の処理系と同等の性能を達成することを示した．

Modular and Efficient Implementation Scheme for Memory Management Systems

Yuji Uchiyama† and Ken Wakita†

The article describes a novel design and implementation scheme of memory management systems for programming languages. The scheme is modular in that it allows the memory management system to be replaced without modifying the rest of the programming language implementation. It is efficient in that the scheme incorporates optimization of the memory management interface to eliminates possible overhead incurred from using this interface. The paper defines an abstract interface between the runtime system and the memory management substratum. Though the interface is simple, it is flexible enough to describe variety of memory management algorithms. Execution efficiency is achieved by a specialization technique that specializes the programming language implementation with respect to the implementation of the memory management system. Modular design typically incurs execution overhead due to use of generic interface definitions. This inefficiency is resolved by adding efficient bytecode instructions which are specialized for a given memory management system and also applying a bytecode transformation technology that make use of the added instructions. This memory management interface and the specialization technique has been implemented, used for description of various memory management algorithms, and tested effectiveness of the approach with a number of small- to medium-scale benchmark programs. The results show that for most cased the specialization technique removes 10% overhead which is otherwise incurred from using our abstract interface.

1. はじめに

自動メモリ管理機能は，多くの現代的なプログラミング言語処理系が持つ重要な機能の 1 つである．実行時システムによる自動メモリ管理機能はプログラマを明示的なメモリ管理という繁雑な処理から解放し，プログラムを簡潔に効率良く記述することを助ける[12),19)]．

自動メモリ管理機能の性能は，言語処理系の上で動作するプログラムの実行速度に大きな影響を与える．このため，自動メモリ管理機能の高速化は，言語処理系の高速化のために重要である[1)]．

その一方で，既存の言語処理系の実装では，メモリ管理システムと処理系の他の各部分とのインタフェースを高度に最適化することで，言語処理系の性能を向上させることが一般的である．このため，実用的な言語処理系の実装では，メモリ管理システムと処理系の

† 東京工業大学大学院数理・計算科学専攻
Department of Mathematical and Computing Science, Graduate School of Tokyo Institute of Technology

他の部分とが，相互に密接に依存してしまう．結果として，既存の言語処理系では，処理系に実装されているメモリ管理システムを変更することは困難である．

本研究では，処理系にメモリ管理システムを容易に実装でき，かつ，プログラムを高速に実行可能な言語処理系を構成する方法を提案する．メモリ管理システムを処理系の他の部分から独立させる手法として，抽象的なモデルの上でのメモリ管理インタフェースを定義する．このインタフェースは処理系とメモリ管理システムの実装を互いに隠蔽し，それぞれの実装を独立させる．さらに，インタフェースを用いて実装された処理系に対して，実行時システム特化とバイトコード変換による高速化手法を提案する．これは，インタフェースを用いた処理系で，既存の処理系での高速化技法と同等の効果を得ることを目的とする手法であり，メモリ管理アルゴリズムに特化したバイトコード命令を自動生成する技術と，バイトコードプログラムを自動変換する技術によって実現している．

メモリ管理システムを処理系の他の部分から独立させることには，以下の利点がある．第1に，処理系に非依存なメモリ管理システムを提供することで，言語処理系を実装する際のコストを軽減する．第2に，個々のアプリケーションごとに最適なメモリ管理アルゴリズムを選択することが可能になる．第3に，メモリ管理アルゴリズムの性能を特定の処理系に依存しない環境で評価できる．本研究で提案する手法を用いて処理系を構成することによって，これらの要求を満足することができる．

本研究では，提案した手法を関数型言語 Objective Caml（O'Caml）[13] のバイトコード処理系に適用し，本研究が定義したインタフェースを用いて O'Caml のメモリ管理システムを再実装した．実装した処理系の上でベンチマークテストによる性能評価を行い，提案した処理系特化手法が，インタフェースを用いたメモリ管理システムの実装によって生じた 10%程度のオーバヘッドを解消し，既存の処理系と同等な性能を達成できることを示した．

本論文の構成を示す．まず 2 章で，メモリ管理システムと処理系とを互いに独立させるためのインタフェースについて述べる．次に，本研究が処理系の高速化のために提案する手法として，3 章では処理系特化手法について述べ，4 章ではインタフェース拡張について述べる．5 章では，本研究の手法に従う処理系の実装について述べ，その処理系の上で本手法の有効性を評価する．6 章で本研究と関連研究との比較を行う．7 章で本研究の今後の課題について述べ，8 章で本論文をまとめる．

2. メモリ管理インタフェース

本研究では，言語処理系の実行時システムとメモリ管理システムとの間に抽象的なインタフェース（**メモリ管理インタフェース**）を定義する．メモリ管理インタフェースは，処理系とメモリ管理システムの実装の詳細を互いに隠蔽し，これらの機能を独立に設計し，実装することを可能にする．

本研究が定義するメモリ管理インタフェースは，メモリ管理システムの実装に有用なものとなるために，以下の要件を満たす必要がある．

抽象性：メモリ管理インタフェースは，処理系とメモリ管理システムとを互いに独立に設計，実装できるように十分な抽象性を持たなければならない．抽象性は，本システムの汎用性のために重要である．抽象性の高いメモリ管理インタフェースを提供することで，同一のメモリ管理システムをさまざまな言語処理系で共通に利用することや，同一の言語処理系でさまざまなメモリ管理システムを利用することが可能となる．

効率性：メモリ管理インタフェースは，インタフェースを用いて実装された処理系の上で，さまざまなプログラムを高速に実行できなければならない．これは，インタフェースを用いて実装されたメモリ管理システムに対して，既存の処理系でのさまざまな高速化技法を適用することが可能であり，かつ，それらの技法によって期待される性能向上が得られることを意味する．この性質によって，本システムを実用的な処理系として利用することや，メモリ管理システムの性能評価のために利用することが可能となる．

本研究では，言語処理系が扱うデータ構造に対する抽象的なモデルを与え，その上でメモリ管理インタフェースを定義することで，上に述べた抽象性の要件を満足させる．本章では，まず，本研究で用いるモデルについて述べ，その次に，本研究が定義するメモリ管理インタフェースについて具体的に述べる．メモリ管理インタフェースが満たすべき要件である効率性については，次章以降で詳細に述べる．

2.1 メモリモデル

本研究では，一般的な言語処理系が実行時に扱うデータ構造と，それに対してプログラムが行う操作とをモデル化し，そのモデルの上でメモリ管理インタフェースを定義する．データ構造のモデル化によって，処理系が扱うデータ構造の詳細をメモリ管理システムから隠蔽し，メモリ管理システムを特定の処理系の実装から独立にできる．また，プログラムがデータ構造

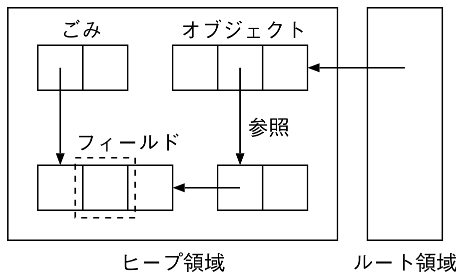

図 1　メモリ空間の抽象モデル
Fig. 1　An abstract view of the memory space.

に加える操作をモデル化し，それぞれの操作に対してインタフェースを定義することで，処理系の実装を特定のメモリ管理システムから独立にできる．

本研究が用いるモデルを**図 1** に示す．本研究では，このモデルによって，処理系が実行時に扱うデータ構造と，それらのデータが配置されるメモリ領域とを抽象化する．

言語処理系がプログラムの実行時に扱うさまざまなデータ構造を抽象化したものを，**オブジェクト**とよぶ．オブジェクトは，特定の大きさの単位構造の列によって構成される．この単位構造を**フィールド**とよぶ．オブジェクトを構成する各フィールドには，プログラムの実行に必要な情報が格納される．このような情報のうち，他のオブジェクトのフィールドを指し示すために利用されるものを**参照**という．

本モデルでは，オブジェクトのフィールドに格納される参照が，以下の 2 つの条件を満たしているものとする．まず第 1 に，フィールドの値が参照か否かを判別できること．第 2 に，参照先のフィールドが属するオブジェクトと，そのオブジェクトの何番目のフィールドであるかを決定できることである．これらの条件を満たさない言語処理系では，一部のメモリ管理インタフェースを正しく実装できず，処理系に実装できるメモリ管理アルゴリズムが制限される．このことについては，2.2 節で述べる．

また，本モデルでは，参照を含みえないオブジェクトを識別するための型情報が存在するものとする．多くの言語処理系では，浮動小数点型や文字列型といった，参照を含まない型のオブジェクトを持っており，その情報を利用して効率の良いメモリ管理システムを実装することが可能である．ただし，この情報は付加的なものであり，本モデルを適用する言語処理系では，識別できる範囲でこの情報を提供できればよい．

次に，プログラムの実行時に利用するメモリ空間を，以下のようにモデル化する．プログラムの実行時に必要なオブジェクトを配置するために利用される領域の全体を，**メモリ空間**という．メモリ空間は，**ルート領域**と**ヒープ領域**という 2 つの部分から構成される．ルート領域に存在するオブジェクトは，実行時システムによって直接的に利用される．これに対して，ヒープ領域に存在するオブジェクトは，他のオブジェクトからの参照をたどることによってのみ利用することができる．したがって，ルート領域にあるオブジェクトからいかなる参照をたどっても到達できないオブジェクトは，決して利用されない．このようなオブジェクトのことを**ごみ**という．

ここで定義したモデルは，オブジェクトを**節**，オブジェクト間の参照を**辺**とする有向グラフと考えることができる．このグラフを**オブジェクトグラフ**とよぶ．処理系の上で動作するプログラムが実行時にデータ構造に加える操作は，オブジェクトグラフ上での節や辺に対する操作として形式化できる．以下，これらの操作について述べる．

- **節の生成**
  オブジェクトの生成は，オブジェクトグラフに新たな節を追加する操作として表現できる．オブジェクトがヒープ領域に生成される場合には，そのオブジェクトを利用できるように，生成されたオブジェクトを指す参照を作らなければならない．本研究では，ヒープ領域に生成されたオブジェクトに対して，ルート領域のオブジェクトからの参照が生成されるものとする．
- **節の除去**
  不要なオブジェクトの占める領域の回収は，オブジェクトグラフの節を除去する操作として表現できる．節の除去にともない，除去される節から他の節への辺も同時に除去される．
- **辺の生成**
  フィールドへの参照の代入は，オブジェクトグラフに新たな辺を導入する操作として表現できる．
- **辺の除去**
  フィールドへの代入によって参照を上書きする処理は，オブジェクトグラフの辺を除去する操作として表現できる．また，不要なオブジェクトの回収にともない，そのオブジェクトから他のオブジェクトへの参照が消滅する場合にも，オブジェクトグラフ上での辺の除去が発生する．
- **グラフの渡り歩き**
  参照をたどって参照先のオブジェクトを利用することは，オブジェクトグラフの辺をたどる操作として表現できる．

本研究が定義したメモリモデルは，ごみ集めのアル

表 1　メモリ管理インタフェース
Table 1　Memory management interface.

インタフェース	概要
`MM_alloc`(value* *result*, int *size*, int *tag*)	オブジェクトの生成
`MM_set_field`(value* *object*, int *i*, value *oldval*, value *newval*)	フィールドへの代入
`MM_get_field`(value* *result*, value* *object*, int *i*)	フィールドアクセス
`MM_initialize`()	プログラムの開始
`MM_finalize`()	プログラムの終了
`MM_root_scan`(void (**function*)(value*, value*))	ルート領域の走査
`MM_is_reference`(value *field*)	参照の識別
`MM_dereference`(value *field*, value* *object*, int *i*)	参照先オブジェクトの獲得
`MM_alloc_failure`()	メモリ割当て失敗の伝達
`MM_is_atomic`(int *tag*)	非参照型の識別

ゴリズムを議論する際に広く利用されているモデルであり，このモデルに従って，さまざまなメモリ管理アルゴリズムを記述することが可能である．本モデルの上に記述できるアルゴリズムの例として，参照カウンタ方式[5)]，マークスイープ方式[15)]，コピー方式[9)]がある．また，インクリメンタル GC[8)] や世代別ごみ集め[14)] といった手法も，本モデルの上に記述することができる．

その一方で，本モデルの上に記述することができないメモリ管理アルゴリズムとして，並列システムや分散システムでのごみ集め[16)]，静的な情報を利用するごみ集めがあげられる．前者は，本モデルが並列実行の概念を含んでいないことが理由である．後者は，本モデルがプログラムの実行時におけるメモリモデルであり，静的な情報を含まないことが理由である．静的な情報を利用するメモリ管理アルゴリズムの例としては，タグなしごみ集め[10)] や region inference[18)] がある．

2.2　メモリ管理インタフェースの設計

本研究では，メモリ管理システムのためのインタフェースを，**表 1** に示される C 言語のマクロの集合として定義する．マクロに現れる value という型は，オブジェクトのフィールドを表す型である．これらのマクロは，実行時システムからメモリ管理システムに処理を要求するものと，メモリ管理システムから実行時システムに処理を要求するものとに分類できる．以下では，それぞれのマクロの意味について解説する．

2.2.1　実行時システムが使うインタフェース

実行時システムからメモリ管理システムに対して処理を要求するインタフェースは，以下のとおりである．言語処理系の実行時システムは，これらのインタフェースを利用して，プログラムの実行中に発生するメモリ操作をメモリ管理システムに伝達する．

- `MM_alloc`(*result*, *size*, *tag*)
  オブジェクトを生成するための領域をメモリ管理システムに要求するマクロ．*size* は生成するオブジェクトのフィールド数を表し，*tag* はオブジェクトの型情報を表す．メモリ管理システムでは，要求された大きさの連続領域を確保し，その領域の先頭を指すポインタを *result* に格納する．領域の確保に失敗した場合は，後述する `MM_alloc_failure` を用いて，実行時システムに制御を戻す．
- `MM_set_field`(*object*, *i*, *oldval*, *newval*)
  オブジェクトのフィールドに対する代入が発生したことを伝達するマクロ．*object* と *i* によって，*object* の *i* 番目のフィールドに代入が起きたことを示し，*oldval* と *newval* によって，代入前の値と代入後の値を示す．このマクロを実装することで，フィールドへの代入の際に必要な処理（*write barrier*）を実装できる．特別な処理が不要ならば，このマクロの定義を空にすればよい．
- `MM_get_field`(*result*, *object*, *i*)
  オブジェクト間の参照をたどることを伝達するマクロ．*object* と *i* によって，参照元のフィールドを示す．メモリ管理システムでは，参照先のフィールドのアドレスを *result* に格納する．このマクロを実装することで，参照をたどる際に必要な処理（*read barrier*）を記述できる．特別な処理が不要ならば，参照元フィールドの値をそのまま *result* に格納すればよい．
- `MM_initialize`()
  プログラムの実行を開始する直前に，実行時システムからメモリ管理システムに制御を移行するマクロ．メモリ管理システムは，このマクロを実装することで，ヒープ領域の確保や初期化の処理を記述できる．
- `MM_finalize`()
  プログラムの実行が終了した直後に，実行時システムからメモリ管理システムに制御を移行するためのマクロ．メモリ管理システムは，このマクロを実装することで，必要な後処理を記述できる．

2.2.2 メモリ管理システムが使うインタフェース

メモリ管理システムから実行時システムに対して処理を要求するインタフェースは，以下のとおりである．メモリ管理システムは，これらのインタフェースを利用することで，処理系の具体的な実装に依存せずに必要な情報を獲得できる．

- `MM_root_scan`(*function*)
  ルート領域を走査し，ヒープ領域への参照について，*function* によって指定された処理を要求するためのマクロ．*function* は，参照元フィールドを表す *from* 引数と参照先フィールドを表す *to* 引数を持つ関数とする．実行時システムは，ルート領域を走査し，その中にあるヒープ領域への参照に対して，*function* を呼び出す．
- `MM_is_reference`(*field*)
  field の値が参照かどうかを実行時システムに問い合わせるためのマクロ．マクロの値は，*field* の値が参照ならば非零の整数値，非参照ならば零とする．本研究のメモリモデルでは，この判定がつねに可能であることを仮定しているが，言語処理系の実装によっては，参照の識別が不可能なことがある．その場合には，このマクロの意味を変更して，値が参照である可能性がある場合に非零値とすることで，保守的ごみ集め[3] を利用することができる．
- `MM_dereference`(*field*, *object*, *i*)
  field に格納されている参照が指し示すオブジェクトを要求するマクロ．実行時システムでは，*object* に，*field* が指し示すオブジェクトの先頭アドレスを格納し，*i* に，何番目のフィールドを指しているかを格納する．*field* の値が非参照の場合には，マクロの動作は未定義とする．本研究のメモリモデルでは，参照先フィールドが属するオブジェクトを決定できることを仮定しているが，それが不可能な処理系では，このマクロを実装することができない．そのような処理系にメモリ管理システムを実装するためには，参照からオブジェクトの先頭アドレスを獲得する処理を独自に実装しなければならない．
- `MM_alloc_failure`()
  実行時システムからのメモリ割当て要求に対して，割当て可能なヒープ領域が存在しないことを伝達するマクロ．実行時システムでは，このマクロに対して，メモリ割当てが失敗した際に行う処理を実装する．
- `MM_is_atomic`(*tag*)
  `MM_alloc` の引数として与えられた *tag* を持つオブジェクトが，参照を含むオブジェクトか否かを問い合わせるマクロ．メモリ管理システムは，このマクロを利用することで，文字列や実数列を走査して参照を探すことを避けることができる．マクロの値は，*tag* を持つオブジェクトが参照を含まないならば非零の整数値，参照を含みうるならば零をとるものとする．このような情報を提供できない処理系では，マクロの値がつねに零となるように実装すればよい．

```
#define MM_alloc(result, size, tag) { \
  value *curr = free_list; \
  int gc = NOT_DONE; \
  while (TRUE) { \
    while (curr != NULL) \
      if (Cell_size(curr) >= (size)) { \
        (result) = curr; \
        break; \
      } else curr = Next(curr); \
    if (curr == NULL) { \
      if (gc == DONE) MM_alloc_failure(); \
      else { \
        garbage_collection(); \
        curr = free_list; \
        gc = DONE; \
      } \
    } else break; \
  } \
}
```

図 2 メモリ割当ての実装例
Fig. 2 An implementation of memory allocation.

2.3 メモリ管理インタフェースの実装例

メモリ管理インタフェースを用いて本処理系にメモリ管理システムを実装する例を示す．**図 2** は，メモリ割当ての処理の実装例である．この実装では，メモリ管理システムは割当て可能な領域をリストとして管理し，`MM_alloc` に対して，そのリストを先頭から順にたどって十分な大きさの連続領域を探し出す．そのような領域が見つからない場合には，ごみ集めを行った後に，再びリストをたどる．ごみ集めを行っても割当て可能な領域が見つからない場合には，`MM_alloc_failure` を実行する．図 2 に現れる `Cell_size` は空き領域の大きさを与えるマクロであり，`Next` は空き領域のリストをたどって次の領域を与えるマクロである．これらは，メモリ管理システムの内部で適切に実装できる．

```
void copy_phase() {
  while (scan_ptr < to_ptr) {
    for (i = 0; i < Size(scan_ptr); i++) {
      field = scan_ptr[i];
      if (MM_is_reference(field)
          && Is_from(field)) {
        MM_dereference
            (field, child_obj, child_i);
        if (Is_forwarded(child_obj))
          Forward(scan_ptr, i);
        else
          Copy(child_obj);
      }
    }
    scan_ptr = Next(scan_ptr);
  }
}
```

図 3 コピー方式によるごみ集めの実装例
Fig. 3 An implementation of copying garbage collector.

次に，**コピー方式**のアルゴリズムによるごみ集めの実装例を示す．**図 3** に示した実装では，幅優先コピー方式に従い，生存しているオブジェクトを *from space* から *to space* にコピーする．コード中に現れるいくつかのマクロは，メモリ管理システムの内部で定義されるものであり，それぞれ以下の処理を行う．

`Size` オブジェクトの大きさを与えるマクロ．オブジェクト生成時に `MM_alloc` に渡される *size* を記録することで，このマクロを実現できる．

`Is_from` 参照先が from space のオブジェクトか否かを返すマクロ．

`Is_forwarded` 参照先のオブジェクトがすでに to space にコピーされているか否かを返すマクロ．

`Forward` 参照元フィールドの値を，参照先オブジェクトのコピー先を指すように変更するマクロ．

`Copy` from space のオブジェクトを to space にコピーするためのマクロ．前述の `Is_forwarded` と `Forward` の処理を可能にするため，コピー元のオブジェクトには，オブジェクトがコピーされたことを示すフラグとコピー先のアドレスを格納する．

`Next` to space に存在する次のオブジェクトを返すマクロ．

3. 実行時システムの特化

前章では，抽象的なメモリ管理インタフェースを与え，それを用いてメモリ管理システムを実装できることを述べた．本研究で定義したインタフェースを利用して，処理系とメモリ管理システムとを互いに独立に実装することが可能である．

```
#define MM_alloc(result, size, tag) { \
  if ((size) <= 256) \
    MM_alloc_small(result, size, tag); \
  else \
    MM_alloc_large(result, size, tag); \
}
```

図 4 `MM_alloc` の実装例
Fig. 4 An implementation of `MM_alloc`.

その一方で，このような抽象的なインタフェースを導入して処理系とメモリ管理システムとを独立させると，特定のメモリ管理システムに対して処理系を効率的に実装することが困難になり，処理系の上で動作するプログラムの実行速度を低下させる原因となる．たとえば，本研究で実装した処理系の上でいくつかのプログラムを実行した結果，同一のアルゴリズムを用いたメモリ管理システムについて，インタフェースを利用して実装したものは，インタフェースを用いずに効率的に実装したものに対して，およそ 10%の速度低下を示した．

本研究では，このような性能面の問題を解決する方法として，メモリ管理システムの実装に対して処理系の実装を効率的なものへと変換する手法，および，インタフェースの拡張によってメモリ管理システムの効率的な実装を可能にする手法を提案する．本章では，前者の高速化手法について述べる．後者の手法については，次節で述べる．

3.1 メモリ管理インタフェースを用いる実装の問題

メモリ管理インタフェースを用いる実装が，実行時の性能低下を引き起こす例を示す．**図 4** は，`MM_alloc` の実装例である．この実装では，オブジェクトの大きさによってメモリ割当て時に行う処理を変更する．たとえば，*Large Object Space*（*LOS*）[4] を用いてメモリ管理を行う場合などに，このような実装がなされる．

この例のように，マクロの内部で引数の値による分岐が起きる場合，処理系がマクロを利用する箇所で引数の値が定数であれば，分岐先の処理を直接呼び出すように処理系の実装を変更することで，より高速に処理することができる．処理系のソースコードに対する静的な解析によって引数の値が定まる場合は，このような最適化はコンパイラによって自動的になされる．

しかしながら，マクロの引数がバイトコード命令のオペランドによって定まる場合には，その値はバイト

コード命令を読み込むまで判明しない．このような場合には，コンパイラによる最適化は不可能であり，実行時の条件分岐を取り除くことは困難である．

生成するオブジェクトの特徴によってメモリ割当ての処理を変更するメモリ管理手法としては，例にあげた LOS を用いる手法のほかに，オブジェクトの型によって処理を変更する手法[17] などがある．このようなアルゴリズムについても，*tag* の値による条件分岐として本研究の手法で扱うことができる．

3.2 処理系特化の概要

上に述べた問題に対して，本研究では，処理系特化による高速化手法を提案する．これは，メモリ管理インタフェースの内部で引数の値による分岐が起こる場合に，それぞれに対応するバイトコード命令を追加し，バイトコードプログラム中の各命令について，条件を満たすものを，追加された命令で置き換える手法である．この置き換えによって，メモリ管理インタフェース内部での冗長な条件分岐が省かれる．

まず，メモリ管理システムの実装者は，インタフェースの引数が特定の条件を満たす場合に限って利用される，限定的なインタフェースを定義する．たとえば 3.1 節に示した LOS の例では，*size* が 256 以下の場合に利用するインタフェース，*size* が 256 を超える場合に利用するインタフェースを，それぞれ定義する．

このように定義されたそれぞれのインタフェースに対して，それらのインタフェースを利用するバイトコード命令を新たに定義する．さらに，バイトコードプログラムを解析し，メモリ管理インタフェースへの引数の値が定まる場合には，適切な命令を実行するように命令を変換する．新たに定義された命令では，実行時の条件分岐を行わないため，既存のメモリ管理インタフェースを実行する命令よりも，同一の処理をより高速に実行することができる．

本研究が提案する処理系特化の概要を，**図 5** に示す．メモリ管理システムの実装者は，独自に定義したインタフェースに関する情報を，特定の形式に従う記述によって本システムに伝達する（図の破線）．その記述に従って，まず，実行時システム特化器が，メモリ管理システムに合わせて新たな命令を定義し，実行時システムを特化する．バイトコード変換器は，個々のバイトコードプログラムを解析し，効率的な命令へと変換する．本章の以下の部分では，これらの処理の詳細について述べる．

3.3 ユーザ定義インタフェース

メモリ管理システムの実装者は，2 章で定義したメモリ管理インタフェースに対応して，引数が特定の条件を満たす場合にのみ利用されるインタフェースを自由に定義できる．これを，**ユーザ定義インタフェース**とよぶ．ユーザ定義インタフェースは，`MM_alloc`, `MM_set_field`, `MM_get_field` に対して，それぞれ任意個の定義が可能である．たとえば，3.1 節の LOS の実装では，`MM_alloc_small`, `MM_alloc_large` をユーザ定義インタフェースとし，処理系が *size* の値によって適切なものを選択するように設定することができる．これらのユーザ定義インタフェースは，対応するメモリ管理インタフェースと同じ引数を持つ．

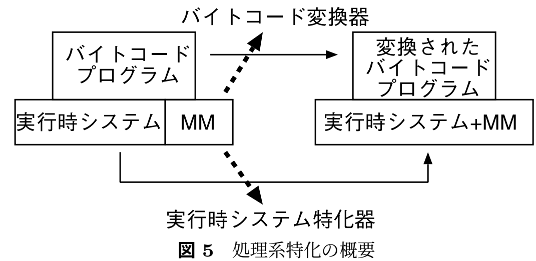

図 5 処理系特化の概要
Fig. 5 An overview of our specialization scheme.

メモリ管理システムの実装者は，ユーザ定義インタフェースの名前とそれぞれの利用条件を，形式的な記述によって処理系に伝える．このための記述を**インタフェース記述**とよぶ．インタフェース記述は，メモリ管理インタフェースと，その引数に関する条件によって，その条件のもとで利用されるユーザ定義インタフェースへの対応を与えるものである．インタフェース記述の形式の例を**表 2** に示す．表の**プロパティ**という項にあるように，インタフェース記述では，ユーザ定義インタフェースが扱うオブジェクトに属性を付加できる．付加された属性は，他のユーザ定義インタフェースの利用条件として指定できる．たとえば，表 2 におけるプロパティの意図は，新たに生成されたオブジェクトの初期化にともなう代入もオブジェクトの大きさによって最適化することである．表 2 では，プロパティを用いて，オブジェクトの大きさを `MM_alloc` から `MM_set_field` に伝達している．

本研究が提案する手法は，**実行時システム特化器**と**バイトコード変換器**によって，与えられたインタフェース記述に基づいて処理系の実装とバイトコードプログラムとを効率的なものに自動変換するものである．実行時システム特化器は，ユーザ定義インタフェースを直接利用するバイトコード命令を処理系に追加し，それに合わせてインタプリタの実装を拡張する．バイトコード変換器は，既存のバイトコードプログラムを解析し，追加されたバイトコード命令を利用するようにプログラムを変換する．以下では，これらの処理の詳

表 2 インタフェース記述の例
Table 2 An interface description.

メモリ管理インタフェース	利用条件	ユーザ定義インタフェース	プロパティ
MM_alloc(*result*, *size*, *tag*)	$size \leq 256$	MM_alloc_small	*result* : small
MM_alloc(*result*, *size*, *tag*)	$size > 256$	MM_alloc_large	*result* : large
MM_set_field(*object*, *i*, *newval*, *oldval*)	*object* : small	MM_set_field_small	
MM_set_field(*object*, *i*, *newval*, *oldval*)	*object* : large	MM_set_field_large	

細について述べる．

3.4 実行時システムの特化

実行時システム特化器は，与えられたインタフェース記述から，個々のユーザ定義インタフェースを直接利用するバイトコード命令を新たに定義し，それらの命令を正しく処理するようにインタプリタの実装を拡張する．この処理は，以下の手順に従って行われる．

まず，実行時システム特化器は，与えられたインタフェース記述から，それぞれのメモリ管理インタフェースに対して，定義されているユーザ定義インタフェースの名前の表を作成する．次に，メモリ管理インタフェースを利用するバイトコード命令に対して，それぞれのインタフェースをユーザ定義インタフェースに置き換えたものを新たな命令として定義し，処理系の命令セットに追加する．さらに，追加した命令を処理するコードをインタプリタに追加する．

実行時システム特化器の動作の例として，O'Caml のバイトコード命令の 1 つである MAKEBLOCK 命令に対する処理を示す．MAKEBLOCK 命令は，新たなオブジェクトをヒープ領域に生成する命令である．生成するオブジェクトの大きさは，MAKEBLOCK 命令のオペランドの値によって定まる．生成されたオブジェクトは，仮想機械のスタックに積まれている値で初期化される．

O'Caml の MAKEBLOCK 命令をメモリ管理インタフェースを用いて実装すると，**図 6** のようになる．MAKEBLOCK 命令は，*size* と *tag* をオペランドとして持つ．オブジェクトを生成する処理は，MM_alloc を用いてメモリ管理システムに要求される．生成したオブジェクトを初期化するための代入は，MM_set_field を用いてメモリ管理システムに伝達される．

実行時システム特化器は，MAKEBLOCK 命令の処理で利用される MM_alloc と MM_set_field をユーザ定義インタフェースに置き換えた命令を新たに定義し，インタプリタの実装を適切に拡張する．たとえば，インタフェース記述として表 2 が与えられたとする．

まず最初に，実行時システム特化器は，与えられたインタフェース記述から，MM_alloc に対して MM_alloc_small と MM_alloc_large が定義されていること，MM_set_field に対して MM_set_field_small と MM_set_field_large が定義されていることを知る．次に，実行時システム特化器は，MAKEBLOCK 命令の処理に現れる MM_alloc と MM_set_field に対して，これらをユーザ定義インタフェースに置き換えた命令を，新たなバイトコード命令として定義し，インタプリタに処理を追加する．

```
Instruct(MAKEBLOCK) {
  int size = *pc++;
  int tag  = *pc++;
  MM_alloc(acc, size, tag);
  for (i = 0; i < size; i++) {
    value val = *sp++;
    ((value *) acc)[i] = val;
    MM_set_field(acc, i, NULL, val);
  }
}
```

図 6 MAKEBLOCK 命令の実装
Fig. 6 Generated instruction table for the MAKEBLOCK instruction.

一般に，1 つのバイトコード命令の処理ではメモリ管理インタフェースを複数回利用し，それぞれのメモリ管理インタフェースに対して，複数個のユーザ定義インタフェースが存在する．したがって，1 つのバイトコード命令からは，これらのすべての組合せに対して，新たなバイトコード命令が定義される．MAKEBLOCK 命令の場合には，MM_alloc と MM_set_field の利用について，それぞれ 3 通りのインタフェースが存在する．このうち，MM_alloc と MM_set_field を利用するものは既存の MAKEBLOCK 命令として定義されている．したがって，実行時システム特化器は，**表 3** の中の 8 つの命令（MAKEBLOCK_1，...，MAKEBLOCK_8）を新たに定義し，インタプリタに追加する．

実行時システム特化器は，このようにして新たに定義したバイトコード命令について，既存の命令と利用されるインタフェースの組から新たに定義した命令への対応を与える表（**命令対応表**）を生成する．これは，後述のバイトコード変換で利用される．たとえば，上に示した MAKEBLOCK に対する処理に対応して，

表 3 MAKEBLOCK に対する命令対応表
Table 3 Instruction table of MAKEBLOCK.

利用するインタフェースの組		対応する命令
MM_alloc	MM_set_field	MAKEBLOCK
MM_alloc	MM_set_field_small	MAKEBLOCK_1
MM_alloc	MM_set_field_large	MAKEBLOCK_2
MM_alloc_small	MM_set_field	MAKEBLOCK_3
MM_alloc_small	MM_set_field_small	MAKEBLOCK_4
MM_alloc_small	MM_set_field_large	MAKEBLOCK_5
MM_alloc_large	MM_set_field	MAKEBLOCK_6
MM_alloc_large	MM_set_field_small	MAKEBLOCK_7
MM_alloc_large	MM_set_field_large	MAKEBLOCK_8

表 3 のような表を生成する．

3.5 バイトコード変換

バイトコード変換器は，実行時システム特化器が定義した効率的な命令を利用するように，バイトコードプログラムを変換する．変換器は，バイトコードプログラム，インタフェース記述，命令対応表を入力として受け取る．変換器はこれらの入力を利用して，バイトコードプログラムの各命令のオペランドを調べ，その命令が新たに定義された命令に変換可能であれば，適切に変換する．バイトコード変換器が各バイトコード命令に対して行う処理の手順は，以下のとおりである．

- まず，バイトコード変換器は，バイトコード命令のオペランドを読み込み，その命令の処理で利用されるメモリ管理インタフェースの引数の値を計算する．
- インタフェース記述から，それぞれのメモリ管理インタフェースについて，利用可能なユーザ定義インタフェースを求める．このとき，先行するインタフェースによって付加されるプロパティは，後続のインタフェースに適切に伝達される．
- バイトコード命令の処理について，利用できるユーザ定義インタフェースを求めた後に，命令対応表から，それらのインタフェースの組に対応するバイトコード命令を決定し，命令を変換する．

バイトコード変換の例として，O'Caml のバイトコードプログラムに MAKEBLOCK 5 0 というバイトコード命令が存在した場合の処理について説明する．バイトコード変換器は，命令のオペランドを読み込み，MM_alloc の引数として $size = 5$，$tag = 0$ をそれぞれ求める．表 2 の記述から，MM_alloc の代わりに MM_alloc_small を利用できることが分かる．MM_alloc_small に対するプロパティの計算から，*acc* に small というプロパティが付加される．このプロパティは MM_set_field_small の利用条件を満たすため，後続の MM_set_field は MM_set_field_small に変更できることが分かる．このようにして利用するインタフェースを決定した後に，表 3 を引き，もとの命令（MAKEBLOCK 5 0）を MAKEBLOCK_4 5 0で置き換える．

4. インタフェースの拡張

2 章で定義したメモリ管理インタフェースには，その抽象性の高さに起因する 2 つの問題がある．それは，メモリ管理システムと処理系とがオブジェクトのヘッダを共有できない問題と，ルート領域を走査するために必要な処理を最適な箇所に記述できない問題である．これらは，処理系の上でのプログラムの実行速度を低下させる問題である．

本研究では，2 章で定義したメモリ管理インタフェースに加えて，これらの問題に対処してより効率的に処理を行うためのインタフェースを定義する．本章では，上に述べたそれぞれの問題に対して，インタフェースの拡張による解決方法について述べる．

4.1 ヘッダ領域の共有

一般に，言語処理系の実行時システムやメモリ管理システムは，処理に必要な管理情報をオブジェクトに付加する．この領域を**ヘッダ領域**とよぶ．言語処理系を実装する際には，ヘッダ領域の構造について，できるだけ小さく，かつ高速に読み書きができるように工夫することが重要である．

本研究が定義したメモリ管理インタフェースは，メモリ管理システムと実行時システムとの実装を互いに隠蔽するため，メモリ管理システムと実行時システムは，それぞれが独立にヘッダ領域を付加してしまう．これは，必要以上のヘッダ領域を確保することを意味し，空間効率を悪化させる．その結果として，ごみ集めの頻度が高まる．また，それぞれがヘッダを作成する処理が分けられるため，ヘッダ作成のコストが倍になってしまう．

本処理系では，実行時システムとメモリ管理システムとが同一のヘッダを共有するために，**表 4** に示すインタフェースを提供し，このようなオーバヘッドを解消できるようにする．

定義したインタフェースは，メモリ管理システムと実行時システムとが同一のヘッダ領域を共有するためのものと，メモリ管理システムから実行時システムに対して，ヘッダ領域の情報を要求するものとに分類できる．まず，ヘッダ領域を共有するためのインタフェースについて述べる．

- MM_encode_header(num)

表 4 ヘッダ領域共有のためのインタフェース
Table 4 Interface for sharing types of header information for objects.

インタフェース	概要
MM_encode_header(int *num*)	整数値からヘッダ形式への変換
MM_decode_header(int *header*)	ヘッダ形式から整数値への変換
MM_create_header(value* *object*, int *size*, int *tag*, int(*header*)	オブジェクトのヘッダの作成
MM_write_header(value* *object*, int *header*)	オブジェクトのヘッダへの書き込み
MM_read_header(value* *object*)	オブジェクトのヘッダの読み込み
MM_get_size(value* *object*)	オブジェクトの大きさを獲得
MM_get_tag(value* *object*)	オブジェクトの型情報を獲得

これは，C 言語での整数値をヘッダ領域に書き込むための形式に変換するマクロである．メモリ管理システムの実装者は，MM_create_header, MM_write_header を用いてヘッダ領域に値を書き込む前に，このマクロを用いて，書き込む値の形式を変換しなければならない．

- MM_decode_header(*header*)
  これは，MM_encode_header によって変換された値を復元し，C 言語での整数値を得るためのマクロである．MM_read_header によって取り出した値をこのマクロで復元することによって，元の整数値が得られる．
- MM_create_header(*object*, *size*, *tag*, *header*)
  これは，MM_alloc の処理の内部で，生成するオブジェクトのヘッダ領域を初期化するために利用するマクロである．メモリ管理システムの実装者は，MM_alloc の内部で，このマクロを用いてヘッダ領域を初期化しなければならない．*object* には，生成するオブジェクトの先頭を指すポインタを与える．*size* と *tag* には，オブジェクトの大きさと型情報とを与える．これらは，MM_alloc によって処理系から渡された値をそのまま与えればよい．*header* には，メモリ管理システムがヘッダ領域に記録する値を与える．この値は，MM_encode_header によって変換されていなければならない．
- MM_write_header(*object*, *header*)
  これは，既存のオブジェクトのヘッダ領域の値を更新するためのマクロである．*object* には対象となるオブジェクトへのポインタを，*header* にはヘッダ領域に書き込む値を，それぞれ与える．MM_create_header の場合と同様に，*header* に与える値は MM_encode_header によって変換されたものでなければならない．
- MM_read_header(*object*)
  これは，オブジェクトのヘッダ領域から，メモリ管理システムが利用する値を読み込むためのマクロである．*object* には，対象となるオブジェクトへのポインタを与える．このマクロの値は，整数値を MM_encode_header によって変換したものとなる．メモリ管理システムでは，必要に応じて MM_decode_header を利用し，値を復元しなければならない．

次に，メモリ管理システムから実行時システムに対して，ヘッダ領域の情報を要求するためのマクロについて述べる．

- MM_get_size(*object*)
  これは，メモリ管理システムから実行時システムに対して，*object* で示したオブジェクトの大きさを問い合わせるためのマクロである．実行時システムでは，このマクロに対して，マクロの値がオブジェクトのフィールド数となるような実装を与える．
- MM_get_tag(*object*)
  これは，メモリ管理システムから実行時システムに対して，*object* で示したオブジェクトの型情報を問い合わせるマクロである．実行時システムでは，このマクロに対して，オブジェクトの生成時に MM_alloc に渡した *tag* の値を持つような実装を与える．

4.2 ルート領域の走査のための処理

メモリ管理機能を持つ言語処理系の実装では，ごみ集めの処理の前後に，処理系の状態を保存，復元しなければならない場合がある．たとえば O'Caml のバイトコード処理系では，ルート領域の走査を正しく行うために，MM_root_scan の利用に到達する関数呼び出しの前後で，仮想機械のレジスタやスタックなどを表すいくつかの変数を保存，復元しなければならない．O'Caml の既存の実装では，このような処理は適切な箇所で効率的に実現されている．

しかし，本研究が定義したメモリ管理インタフェースはメモリ管理システムの内部の実装を処理系から隠蔽してしまうため，適切な箇所に仮想機械の状態を保存，復元するコードを挿入することは難しい．この結果，仮想機械の状態保存を実行時システム側で実装す

表 5 仮想機械の状態保存のためのインタフェース
Table 5 Interface to save and restore the virtual machine state.

インタフェース	概要
MM_save_state()	仮想機械の状態を保存
MM_restore_state()	仮想機械の状態を復元

ると，メモリ管理インタフェースの前後に処理を挿入することになる．これは，本来，まれにしか起こらないごみ集めに付随して行われる処理を，頻繁に実行されるメモリ割当てのたびに実行してしまうので，著しいオーバヘッドをともなう．

そこで本研究では，仮想機械の状態保存と復元を行うためのインタフェース（**表 5**）を提供し，メモリ管理システムの実装者が，必要な箇所でインタフェースを明示的に利用することとする．MM_save_state は，メモリ管理システムから実行時システムに対して，仮想機械の状態を保存するように要求するためのマクロである．MM_restore_state は，仮想機械の状態を復元するように要求するマクロである．メモリ管理システムの実装者は，MM_root_scan の利用に到達する関数呼び出しの前後に，これらのマクロを記述しなければならない．

5. 実装と性能評価

本研究では，提案した手法に従って，メモリ管理システムを独立な機能として自由に実装できる処理系を実装した☆．この処理系の性能を評価するために，関数型言語 Objective Caml（O'Caml）[13] のバイトコード処理系のメモリ管理機能を本処理系で書き直し，もとの実装との性能を比較した．

5.1 メモリ管理アルゴリズムの実装と評価

本システムが提供するメモリ管理インタフェースを用いて，いくつかの基本的なメモリ管理システムを実装し，メモリ管理インタフェースの記述力を評価した．実装したアルゴリズムとそれぞれの実装に要した行数は，**表 6** に示すとおりである．

さらに，これらのメモリ管理システムが実装された処理系の上で，O'Caml で記述されたベンチマークプログラムを実行し，各アルゴリズムの実行性能を比較した．計測に用いたベンチマークプログラムと，それぞれのプログラムがメモリ割当てを行う回数（MM_alloc(*size*, *tag*) の呼び出し回数），割り当てられるヒープ領域の総量（*size* の累計）は，**表 7** に示すと

☆ 現在のシステムでは，実装の不備により，MM_get_field は正しく扱えない．

表 6 アルゴリズムの記述に要した行数（C 言語）
Table 6 Lines of C code required to describe memory management algorithms.

アルゴリズム	行数
マークスイープ方式[15] (MS)	257
コピー方式[9] (Copy)	199
世代別コピー方式[14] (GenC)	428
遅延参照カウンタ方式[7] (DRC)	336

おりである．

実験の結果を**表 8** に示す．DRC に対する一部のベンチマークプログラムは，メモリリークを原因とするヒープ領域の枯渇のために，実行を完了できなかった．これは，DRC アルゴリズムでは環状のごみを回収できないことに起因する．参照カウンタ方式に基づくメモリ管理では，この問題を回避するために，他の方式を補助的に利用することが一般的である．本処理系においても，DRC と他の方式とを組み合わせることは容易である．

マークスイープ方式とコピー方式との比較では，表 8 から，多くのプログラムにおいて，コピー方式はマークスイープ方式より実行性能に優れ，世代別コピー方式はそれよりもさらに優れていることが分かる．極端な例外として，TSP ではコピー方式における性能が著しく劣っている．この理由は，TSP では，寿命が実行全体にわたる一群のオブジェクトを保持し，そのためにコピー方式では多くのオブジェクトを繰り返しコピーしてしまうためである．世代別コピー方式では，そのようなオブジェクトは旧世代領域にコピーされ，ごみ集めの対象からはずれるため，マークスイープ方式やコピー方式と比較して非常に優れた性能を示す．

5.2 高速化手法の評価

次に，3 章，および，4 章で提案した高速化手法に対する評価を行う．提案した手法の効果を評価するために，まず，既存の O'Caml のメモリ管理方式を本研究が定義したメモリ管理インタフェースを利用して再実装した．

O'Caml に実装されているメモリ管理システムは，世代別コピー方式に基づいている．O'Caml のメモリ管理システムでは，ヒープ領域を 2 つの世代に分けて管理する．メモリ管理システムは，新しいオブジェクトを新世代領域に割り当て，新世代領域のごみ集めの処理として，生存しているオブジェクトを旧世代領域にコピーする．旧世代領域のごみ集めにはマークスイープ方式を用いる．

高速化手法の評価のために，メモリ管理インタフェースを用いて実装された処理系に対して，3 章で提案し

表 7 各ベンチマークプログラムのメモリ割当て回数と割り当てられたメモリ領域の総量
Table 7 The number of allocation times and the amount of allocated memory regions for each benchmark programs.

ベンチマーク		割当て回数	総メモリ領域（ワード）
FFT	高速フーリエ変換	681240267	2064602303
KB	Knuth-Bendix の完備化	233877974	631965210
Life	ライフゲーム	33050477	106681360
Logic	論理証明	291927857	1105496964
Nucleic	核酸の三次元構造解析	1212741007	3880826288
R-region	画像領域分割	1155208749	5249180590
Simple	流体のシミュレーション	344575448	1067181406
TSP	巡回セールスマン問題	194441195	601643144

表 8 各アルゴリズムでのベンチマーク実行時間（秒）
Table 8 Total execution time for each benchmark programs using different memory management algorithms.

	MS	Copy	GenC	DRC
FFT	177.83	150.93	101.41	—
KB	157.81	137.36	142.03	—
Life	57.72	54.25	55.51	64.29
Logic	117.85	86.51	94.50	—
Nucleic	252.28	134.58	108.31	—
R-region	511.36	521.11	333.75	—
Simple	110.85	81.79	76.18	—
TSP	121.44	324.35	34.05	241.07

た処理系特化手法と 4 章で提案したインタフェース拡張を適用して高速化した処理系を構築し，それぞれの処理系の上で，表 7 に示したベンチマークプログラムの実行時間を計測した．計測に用いた計算機環境は以下のとおりである．

- Intel Pentium III 866 MHz，一次命令キャッシュ16 KB，一次データキャッシュ16 KB，二次キャッシュ256 KB，メモリ 512 MB，Vine Linux 2.1.5.
- Sun UltraSPARC III 750 MHz，一次命令キャッシュ64 KB，一次データキャッシュ64 KB，二次キャッシュ8 MB，メモリ 1 GB，Solaris 8.

実験結果を**表 9**，**表 10** に示す．表 9 は Pentium III における実行時間を，表 10 は UltraSPARC III における実行時間を，それぞれ示している．表の各項は，プログラムの実行に用いた処理系の種類を表す．左から順に，高速化手法を適用していない処理系，処理系特化手法を適用した処理系，ヘッダ共有のためのインタフェースを利用した処理系，仮想機械の状態保存のためのインタフェースを利用した処理系，これら 3 つの高速化手法をすべて適用した処理系，既存の O'Caml のバイトコード処理系である．各項目の括弧内の値は，既存の O'Caml のバイトコード処理系での実行時間に対する比を表す．

表 9，表 10 の結果は，提案した高速化手法を適用することでインタフェースによる素朴なモジュール化にともなう 10%程度のオーバヘッドを解消でき，既存の O'Caml と同等の性能を達成できることを示している．また，この結果から，提案した高速化手法である，処理系特化手法，ヘッダ共有のためのインタフェース拡張，仮想機械の状態保存のためのインタフェース拡張は，それぞれ独立にオーバヘッドの解消に寄与することも分かる．

本システムの高速化手法は，単に処理系を高速化できるだけではなく，従来の高速化技法とほぼ同じ性能を達成できる点に特徴がある．すなわち，本システムを利用して実装されたアルゴリズムの性能は，そのアルゴリズム本来の性能をよくシミュレートする．このような性能の予測性は，本システムをさまざまなメモリ管理アルゴリズムの性能評価に利用できる可能性を示唆している点で重要である．

Pentium III での倍精度浮動小数点数を多用する FFT の実行では，最適化を行う前にすでに O'Caml よりも高い性能を得ているという例外的な現象が見られる．これは，ヘッダ共有を行わないことによって，偶然にほとんどの浮動小数点数が 2 ワード境界に配置され，それらへのメモリアクセスが高速化されることによる．このことを確認するため，高速化を行わない処理系で新世代領域の位置を 1 ワードずらして FFT を実行したところ，実行時間は 109.34 秒であった．この結果と表 9 での結果との平均は 105.38 秒であり，メモリアクセス速度の影響を除外すれば，FFT についてもヘッダ共有の効果があることが分かる．

6. 関連研究

言語処理系に対してさまざまなメモリ管理システムを実装するためのインタフェースを提供する研究として，*Java ExactVM*（*EVM*）での GC インタフェース[6] に関する研究がある．EVM では，処理系の開発者に対してメモリ管理システムを実装するためのイン

表 9 Pentium III でのベンチマークプログラムの実行時間（秒）
Table 9 Elapsed time on the Pentium III platform (seconds).

	高速化なし		処理系特化		ヘッダ共有		状態保存		すべて適用		O'Caml
FFT	101.41	(97%)	101.85	(98%)	104.21	(100%)	101.72	(98%)	104.26	(100%)	104.31
KB	142.03	(106%)	139.22	(104%)	137.21	(103%)	141.04	(105%)	132.65	(99%)	133.85
Life	55.51	(102%)	55.09	(101%)	54.28	(99%)	55.22	(101%)	53.69	(98%)	54.65
Logic	94.50	(113%)	88.08	(105%)	90.64	(108%)	93.14	(111%)	82.43	(98%)	83.93
Nucleic	108.31	(107%)	109.39	(108%)	104.35	(103%)	106.52	(105%)	100.13	(99%)	101.50
R-region	333.75	(110%)	307.86	(102%)	324.85	(107%)	326.91	(108%)	292.31	(96%)	303.28
Simple	76.18	(111%)	71.55	(104%)	74.28	(108%)	74.79	(109%)	67.79	(99%)	68.53
TSP	34.05	(106%)	33.37	(104%)	32.96	(103%)	33.65	(105%)	31.84	(99%)	32.07

表 10 UltraSPARC III でのベンチマークプログラムの実行時間（秒）
Table 10 Elapsed time on the UltraSPARC III platform (seconds).

	高速化なし		処理系特化		ヘッダ共有		状態保存		すべて適用		O'Caml
FFT	176.83	(104%)	177.72	(105%)	172.61	(102%)	176.72	(104%)	171.43	(101%)	169.57
KB	208.30	(112%)	204.25	(110%)	199.57	(107%)	201.71	(109%)	196.76	(106%)	185.73
Life	90.67	(100%)	90.78	(100%)	90.80	(100%)	88.82	(98%)	90.86	(100%)	90.87
Logic	135.94	(113%)	130.88	(109%)	130.51	(109%)	132.35	(110%)	123.06	(102%)	120.23
Nucleic	229.21	(103%)	229.83	(103%)	220.35	(99%)	227.76	(102%)	217.01	(97%)	223.46
R-region	487.15	(112%)	470.47	(108%)	463.97	(107%)	477.80	(110%)	441.46	(102%)	434.20
Simple	111.18	(108%)	109.62	(106%)	107.36	(104%)	109.03	(106%)	104.41	(101%)	103.18
TSP	56.39	(107%)	55.73	(106%)	53.99	(103%)	55.95	(107%)	53.05	(101%)	52.48

タフェースを提供しており，本研究で提案した処理系と同様に，処理系の実装と独立にメモリ管理システムを実装できる．

本研究が提案したインタフェースと EVM が提供するインタフェースとの違いは，インタフェースの抽象性である．EVM が提供するインタフェースは，処理系の開発者を対象としたものであり，仮想機械の実装に対応した具体的なインタフェースとなっている．これに対して，本研究が提案するインタフェースは，処理系の具体的な実装によらない抽象的なインタフェースであり，処理系の内部の実装に詳しくない者でも，容易にメモリ管理システムを実装できる．

また，インタフェースを用いる実装による性能低下については，EVM では考慮されていない．その代わりに，EVM では処理系の実装に対応した具体的なインタフェースを提供することで，性能低下を回避している．本研究のアプローチは，抽象的なインタフェースを提供することでメモリ管理システムが特定の処理系に依存することを避けながら，既存の処理系と同等な性能を達成できるという点で，より一般性の高い手法であるといえる．

本研究の有用性として，アプリケーションごとに最適なメモリ管理システムを実装できる点をあげた．これを目的としたシステムとしては，C++用の拡張可能なメモリ管理ライブラリとしての Customisable Memory Manager（CMM）[2] がある．CMM では，アプリケーションを実装するクラスに自動メモリ管理を実装するクラス（`CmmHeap` と `CmmObject`）を継承し，再定義することで，クラスごとに異なるメモリ管理ポリシーを定義できる．

メモリ管理システムを処理系非依存に実装し，さまざまな処理系で利用できることを目標にした研究として，文献 11) がある．本研究と文献 11) との違いは，本研究のインタフェースでは処理系とメモリ管理機能とを互いに独立にし，同一の処理系にさまざまなメモリ管理システムを自由に実装することも可能にしている点である．

7. 将来課題

本章では，本研究の今後の課題について述べる．

まず，3 章で述べた処理系特化による高速化の手法については，以下の 2 点が今後の課題である．1 つは，新たなバイトコード命令を定義する際に，無駄な命令を生成しないことである．現在の実装では，バイトコード命令の実行で複数のメモリ管理インタフェースが使われる場合，ユーザ定義インタフェースのあらゆる組合せに対して新たなバイトコード命令を定義してしまう．これは，インタプリタのサイズを不必要に増加させてしまい，問題である．実行時システム特化器が新たな命令を定義する際に，ユーザ定義インタフェースの利用条件の記述を積極的に利用することで，不必要な命令を生成しないようにできると考えられる．

もう１つは，バイトコード変換時の解析である．現時点で実装されているバイトコード変換器は，各命令のオペランドが定数の場合に対応する命令に置き換えるという単純な変換しか行っていない．バイトコードプログラムに対する解析を通して命令間でオブジェクトのプロパティを伝播することができれば，より多くのバイトコード命令を変換することが可能だと考えられる．

次に，本研究が提案した手法を他の処理系に実装することが，重要な課題である．本研究では，O'Camlのバイトコード処理系を変更して，メモリ管理システムを自由に実装できる処理系を構築し，この処理系にいくつかのメモリ管理アルゴリズムを実装することで，特定のメモリ管理システムに依存しない処理系を構築できることを確認した．その一方で，特定の処理系に依存しないメモリ管理システムを実装できることを示すためには，提案したインタフェースを利用する他の処理系を構築し，メモリ管理システムの同一の実装をさまざまな処理系に適用できることを示すことが必要である．

8. おわりに

本研究では，メモリ管理システムと処理系の他の部分との実装を独立にしながら，既存の処理系と同等の実行性能を実現するための手法を提案した．提案した手法に従って処理系を実装し，その処理系の上にいくつかのアルゴリズムに基づくメモリ管理システムを実装できた．また，処理系の高速化のために提案した処理系特化手法とインタフェース拡張を実装した処理系に適用し，これらの手法の効果を調べた．ベンチマークテストによる性能評価の結果，これらの高速化手法が，メモリ管理システムと処理系とを独立にすることによって生じる10%程度の性能低下を解消することを示した．この結果，本手法に基づく処理系の実装において，既存の処理系と同等な性能を達成することができた．

参考文献

1) Appel, A.W.: *Compiling with Continuations*, chapter 16, pp.205–214, Cambridge University Press (1992).
2) Attardi, G. and Flagella, T.: A Customisable Memory Management Framework, Technical Report TR-94-010, International Computer Science Institute, Berkeley (1994).
3) Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice and Experience*, Vol.18, No.9, pp.807–820 (1988).
4) Caudill, P.J. and Wirfs-Brock, A.: A Third-Generation Smalltalk-80 Implementation, *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, Meyrowitz, N.(Ed.), ACM SIGPLAN Notices, Vol.21, No.11, pp.119–130, ACM Press (1986).
5) Collins, G.E.: A Method for Overlapping and Erasure of Lists, *Comm. ACM*, Vol.3, No.12, pp.655–657 (1960).
6) White, D. and Garthwaite, A.: The GC interface in the EVM, Technical Report SML TR–98–67, Sun Microsystems Laboratories (1998).
7) Deutsch, L.P. and Bobrow, D.G.: An Efficient Incremental Automatic Garbage Collector, *Comm. ACM*, Vol.19, No.9, pp.522–526 (1976).
8) Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M.: On-The-Fly Garbage Collection: An exercise in Cooperation, *Comm. ACM*, Vol.21, No.11, pp.965–975 (1978).
9) Fenichel, R.R. and Yochelson, J.C.: A Lisp Garbage Collector for Virtual Memory Computer Systems, *Comm. ACM*, Vol.12, No.11, pp.611–612 (1969).
10) Goldberg, B.: Tag-Free Garbage Collection for Strongly Typed Programming Languages, *ACM SIGPLAN Notices*, Vol.26, No.6, pp.165–176 (1991).
11) Hudson, R.L., Moss, J.E.B., Diwan, A. and Weight, C.F.: A Language-Independent Garbage Collector Toolkit, Technical Report COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science (1991).
12) Jones, R.E.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
13) Leroy, X.: Objective Caml. `http://caml.inria.fr/ocaml/`
14) Lieberman, H. and Hewitt, C.E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (1983).
15) McCarthy, J.: Recursive Functions of Symbolic Expressions and their Computation by Machine, *Comm. ACM*, Vol.3, pp.184–195 (1960).
16) Plainfossé, D. and Shapiro, M.: A Survey of Distributed Garbage Collection Techniques, *Proc. International Workshop on Memory Management*, Baker, H.(Ed.), Lecture Notes

in Computer Science, Vol.986, ILOG, Gentilly, France, and INRIA, Le Chesnay, France, Springer-Verlag (1995).

17) Reppy, J.H.: A High-Performance Garbage Collector for Standard ML, Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ (1993).

18) Tofte, M. and Talpin, J.-P.: A Theory of Stack Allocation in Polymorphically Typed Languages, Technical Report Computer Science 93/15, University of Copenhagen (1994).

19) Wilson, P.R.: Uniprocessor Garbage Collection Techniques, *Proc. International Workshop on Memory Management*, Bekkers, Y. and Cohen, J.(Eds.), Lecture Notes in Computer Science, Vol.637, University of Texas, USA, Springer-Verlag (1992).

(平成 13 年 5 月 31 日受付)
(平成 13 年 8 月 31 日採録)

内山　雄司

1975 年生．1999 年東京工業大学理学部情報科学科卒業．2001 年同大学大学院情報理工学研究科数理・計算科学専攻博士前期課程修了．同大学院博士後期課程在学中．プログラミング言語，メモリ管理手法等に興味を持つ．ACM 学生会員．

脇田　　建（正会員）

1965 年生．1989 年東京大学理学部情報科学科卒業．1991 年同大学大学院理学系研究科情報科学専攻修了．東京工業大学大学院情報理工学研究科数理・計算科学専攻講師．博士（理学）．プログラミング言語，分散処理等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE 各会員．