令和 3 年度修士論文

組込み機器向け JavaScript 処理系に向けた空間効率の良いごみ集めの実現

電気通信大学 情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスコース

学籍番号 : 2031039
氏名 : 小野澤 拓
指導教員 : 岩崎 英哉 教授
指導教員 : 中山 泰一 教授
提出日 : 2022 年 1 月 28 日

要旨

多くのオブジェクト指向言語では、処理系がメモリ管理を行うため、処理系にはごみ集めが実装されている。メモリに制約のある組込みシステムにこのような処理系を実装する場合、ごみ集めにおいてコンパクションを行うことで、限られたメモリ空間を効率よく利用する必要があり、コンパクションアルゴリズム自身も空間効率の良いものである必要がある。空間効率の良いコンパクションアルゴリズムのひとつとして、Jonkers のスレッド化コンパクションアルゴリズムが知られている。Jonkers のアルゴリズムではオブジェクトを移動させるため、それらのオブジェクトを指すポインタを更新する。特に、オブジェクト内部のポインタを更新するためには、オブジェクト内部のポインタが存在する位置を記録した、レイアウト情報が必要になる。レイアウト情報はメタオブジェクトに記録する技法がよく用いられるが、メタオブジェクト間にポインタがある場合、オブジェクトとメタオブジェクトを同じヒープ内に保持してしまうと、Jonkers のアルゴリズムを適用することができない。なぜなら、コンパクション中はメタオブジェクト間のポインタが辿れなくなるからである。本研究では、Jonkers のアルゴリズムを、オブジェクトとメタオブジェクトを同じヒープ内に保持しながら適用できるよう、改良したアルゴリズムを提案する。提案手法は、Jonkers のスレッド化コンパクションと同等の空間効率で実装することが可能である。提案手法を、組込みシステム向け JavaScript 処理系に実装し、他の複数のごみ集めアルゴリズムと性能を比較した。その結果、提案手法は空間効率が最も高く、ヒープサイズが小さい場合には実行時間も最も短くなることを確認した。

目次

1 はじめに

計算資源が限られた実行環境でも動作する JavaScript 処理系のひとつとして、組込みシステム向け JavaScript 処理系である eJS [19] がある。eJS は、JavaScript プログラムを実行する仮想機械 (VM) である eJSVM を提供する。JavaScript は処理系がメモリを管理するプログラミング言語であり、eJSVM にはごみ集め (Garbage Collection、以下 GC) が実装されている。eJSVM の標準的な設定では、マークスイープ方式の GC が使われる。しかし、マークスイープ方式の GC は、オブジェクトを移動させることが無い非ムービング GC であり、プログラムを実行するにつれ、空き領域が生存オブジェクトで分断される断片化と呼ばれる状態が発生する。断片化が激しくなると、メモリの総空き容量は足りていても、連続した空き領域が確保できないために、オブジェクトのためのメモリの確保に失敗してしまう場合がある。

断片化を解消する方法のひとつに、マークコンパクト方式の GC がある。これはコンパクションと呼ばれる操作により、生存オブジェクトをヒープの端に寄せることで、空き領域を 1 つにまとめる GC として知られている。中でも Jonkers によるスレッド化コンパクションアルゴリズム [12] は、オブジェクトやメモリ中に追加の領域を必要とせずにコンパクションを行うことが可能な、メモリ効率が良いコンパクションアルゴリズムとして知られている。

eJSVM では、JavaScript プログラムを効率よく実行するため、オブジェクトのレイアウト情報を Hidden クラスと呼ばれるメタオブジェクトに記録する。ここで、レイアウト情報とは、オブジェクト内に存在するデータの種類や位置、個数などの情報である。Hidden クラスは、一般オブジェクトと同じヒープ内に作られ、GC の対象になる。Hidden クラスは、実際にはポインタで連結された複数のメタオブジェクトから構成されている。そのため、オブジェクトのレイアウトを特定するためには、メタオブジェクトが持つポインタを辿る必要がある。ここで、メタオブジェクトのレイアウトは固定されており、メタオブジェクト自身のレイアウトは他のメタオブジェクトを参照することなく特定できるものとする。

Jonkers のスレッド化コンパクションは、生存オブジェクトにマークをつけた後、ヒープを先頭から走査し、すべての生存オブジェクトが持つポインタに対してスレッド化という操作を行う。この処理を行うためには、各生存オブジェクトのレイアウトを特定する必要がある。しかし、スレッド化されたポインタはコンパクションが終わるまで辿ることができない、という制約がある。この制約のため、Hidden クラスと一般オブジェクトを区別せず同じヒープで管理すると、メタオブジェクトが持つポインタがスレッド化された後はそのポインタを辿ることができない。結果として一般オブジェクトのレイアウトが特定できなくなり、Jonkers のスレッド化コンパクションアルゴリズムを適用することができない。

本研究では、この問題を解決するために、Fusuma と呼ぶ新たなアルゴリズムを提案する。提案するアルゴリズムでは、一般オブジェクトとメタオブジェクトを割り当てに際して区別し、一

般オブジェクトはヒープの片方の端から、メタオブジェクトはヒープの他方の端から、順に割り当てる。コンパクションを行う際には、まず一般オブジェクトをスレッド化し、Hidden クラスを参照する必要がなくなった後に、Hidden クラスを構成するメタオブジェクトをスレッド化する。このように、Hidden クラスを参照する処理を、Hidden クラスに対する処理が実行されるタイミングと分けることで、Jonkers のスレッド化コンパクションとほとんど同様にしてコンパクションを行うことができる。提案する Fusuma アルゴリズムは、Jonkers のアルゴリズムと同様に、追加の領域を必要としない。

本研究では、提案する Fusuma アルゴリズムを含む、メタオブジェクトを考慮したいくつかのコンパクションアルゴリズムを eJSVM に実装し、ヒープサイズを変化させて複数のベンチマークを実行して性能を評価した。これにより、提案するアルゴリズムは空間効率が最もよく、オブジェクトのアロケーションがある程度頻繁に行われ、GC 回数が多いプログラムでは、既存の GC アルゴリズムよりも実行時間を短くできることを示した。

本研究で扱う問題は、eJS に特有のものではない。たとえば、V8 JavaScript エンジン[*1]では、各プロパティがポインタかボックス化されていない値かを判定するために、GC は Hidden クラスが持つポインタを辿る必要がある [7]。このような処理系に Jonkers のスレッド化コンパクションを実装する際には eJS と同じ問題が生じるが、本研究で提案するアルゴリズムで解決できる。また、Jonkers のスレッド化コンパクション以外のスライディングコンパクションアルゴリズム [13, 10] も、本研究で注目する問題と同じ問題を抱えており、本研究で提案するアルゴリズムを応用することで解決することが可能である。これらの議論については第 8 章で詳しく取り扱う。

本論文では、まず第 2 章で eJS について説明する。第 3 章で既存の GC アルゴリズムの概要と問題点を説明し、第 4 章でそれらのアルゴリズムを eJS へ実装する単純な方法を説明する。第 5 章では複数の問題を解決した Fusuma アルゴリズムを提案し、第 6 章で eJSVM への実装について説明する。第 7 章では、第 6 章で実装した Fusuma アルゴリズムの性能を実験により評価する。第 8 章では、関連研究を紹介し、他の処理系でのメタオブジェクトへの対応との比較や、他の GC アルゴリズムへの応用について議論する。

[*1] https://v8.dev/

2　eJS

本章は、組込み機器向け JavaScript 処理系の 1 つである eJS の概要について説明した後、eJSVM 内部でのオブジェクトの表現に関する説明をする。その後、JavaScript 処理系で広く用いられる高速化技法である Hidden クラスについて説明し、eJSVM 内部における Hidden クラスの実装法を述べる。

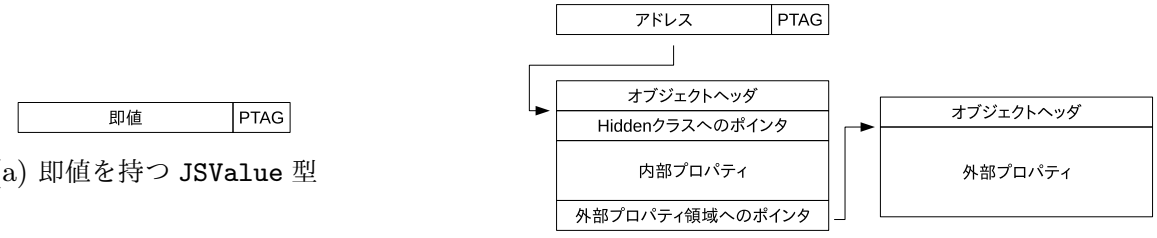
2.1　eJS の概要

eJS (embedded JavaScript) [19] は、IoT 機器を含む組込みシステムにおけるプログラム開発を、広く利用されている高級言語のひとつである JavaScript で行えるようにすることによって、プログラム開発者の負担を減らすとともに、アプリケーション開発の効率化・試作の容易化を目指して開発されている JavaScript 処理系である。

eJS は、実行環境と実行するアプリケーションに合わせてカスタマイズされた JavaScript 仮想機械である eJSVM を自動生成する仕組みを提供する。実行環境に合わせたカスタマイズの例として、eJSVM 内部で利用するデータ型の実装を、32 ビットプロセッサ向けの実装と 64 ビットプロセッサ向けの実装から選択することができる [16]。実行するアプリケーションに合わせたカスタマイズの例として、仮想機械命令のオペランドが取りうる型を制限することで、必要最低限の実装を持つ小型で高速な仮想機械を生成することができる [19]。eJSVM は 100KB 程度のヒープを持つ機器で動作させることを目指し開発が行われている。

2.2　オブジェクトのレイアウト

eJSVM では JavaScript の第一級の値を表す C 言語におけるデータ型として `JSValue` 型を定義し利用している。図 2.1 に `JSValue` 型の構造を示す。`JSValue` 型は型情報 PTAG (pointer



(a) 即値を持つ `JSValue` 型

(b) アドレスを持つ `JSValue` 型

図 2.1: `JSValue` 型のイメージ図

tag) と、即値またはオブジェクトへのアドレスを持つ。`JSValue` 型の値が整数や真偽値などの場合には即値を利用し、JavaScript における `Object` や `Array` などを表す場合はオブジェクトへのアドレスを持つ。eJSVM のオブジェクトは、GC 対象のヒープ内に、32 ビット境界または 64 ビット境界に合わせて割り当てる。そのため、オブジェクトを指すポインタの下位 2 ビットか下位 3 ビットは常に 0 となる。これを利用して、常に 0 となる下位ビットに PTAG の情報を保持する。eJS を利用する開発者は、頻出するデータ型に固有の PTAG 値を割り当てることにより、`JSValue` 型の値の型判別を高速化することができる。PTAG 値を共有するその他の複数のデータ型は、次に説明するオブジェクトヘッダに格納される型情報である HTAG (header tag) を用いて型の判別を行う。

一般オブジェクトは、サイズや型情報を格納したオブジェクトヘッダ、オブジェクトのレイアウトを記録する Hidden クラスへのポインタ、内部プロパティ領域、外部プロパティ領域へのポインタの 4 つの要素からなる。オブジェクトを指すポインタは、オブジェクトヘッダの下側 (次) のアドレスを指す。

オブジェクトヘッダには、主に GC で利用する情報を記録する。オブジェクトヘッダの内容は GC アルゴリズムごとに異なるが、多くの実装で共通して、オブジェクトのサイズ情報と、型情報 (HTAG) が含まれる。また、オブジェクトヘッダ自体のサイズも GC アルゴリズムごとに異なるが、1 ワード (1 ポインタ幅)、または 2 ワード (2 ポインタ幅) となっている。

JavaScript では一般オブジェクトのプロパティを動的に追加、削除することができる。そのため eJS では、オブジェクトの外部に外部プロパティ領域を用意し、オブジェクトのプロパティを通常は外部プロパティ領域に記録する。

eJS には、組込み関数オブジェクトが持つ関数ポインタの様に、JavaScript プログラムから直接参照することはできないが、特定の型のオブジェクトが必ず保持する、特殊なプロパティが存在する。これらの特殊プロパティは非 `JSValue` 型の値として、オブジェクトの内部プロパティ領域へ格納される。内部プロパティのレイアウトやデータ型はオブジェクトのデータ型ごとに異なり、特に GC 対象ヒープ内のオブジェクトを指すポインタである場合、GC の際に過不足なく適切に処理する必要がある。

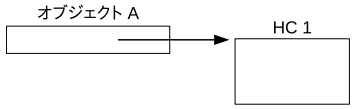
eJS 以外の JavaScript 処理系 [6] では、それまでのプログラムの実行履歴から、そのオブジェクトに将来追加される可能性が高いプロパティを内部プロパティとして扱い、オブジェクトの割り当て時に予め内部プロパティ領域を追加で確保する。一方で、プログラム実行中に動的に追加されるプロパティは外部プロパティとして扱い、外部プロパティ領域に格納する。

2.3 Hidden クラス

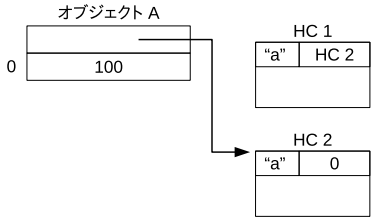
JavaScript は、プログラムの実行時に一般オブジェクトにプロパティを自由に追加したり、削除したりすることができる。このように動的に追加、削除されるプロパティを実現するために、素朴な実装では、プロパティ名を Key とし、プロパティの値を Value とした Key-Value ペアの

```
var A = {};    //ステップ1
A.a = 100      //ステップ2
A.b = "Hello" //ステップ3
var B = {};    //ステップ4
B.a = 200      //ステップ5
```

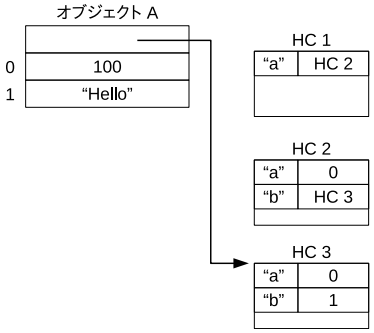
(a) JavaScript コード例



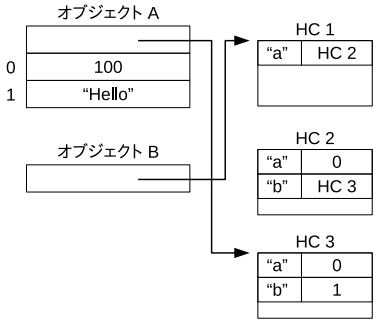
(b) ステップ 1



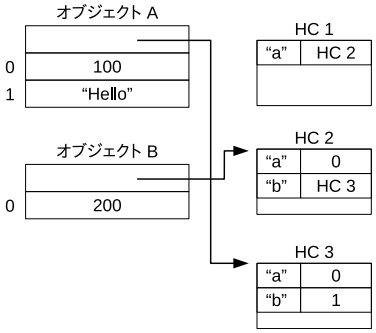
(c) ステップ 2



(d) ステップ 3



(e) ステップ 4



(f) ステップ 5

図 2.2: オブジェクトへのプロパティの追加と、それに伴う Hidden クラスの「成長」の例

配列や、Key をハッシュしたハッシュ表を用いる。しかしオブジェクトごとに Key-Value ペアの配列やハッシュ表を用意する素朴な実装では、空間効率や時間効率が悪い。これを解消する技法のひとつが Hidden クラス [4] である。

Hidden クラスを利用する場合、各オブジェクトはプロパティの値だけを配列 (以後、プロパティ配列と呼ぶ) を用いて管理する。それとは別に、各オブジェクトは Hidden クラスへのポインタを持つ。Hidden クラスは、プロパティ名からプロパティ配列内の添字を求める関数として機能する。Hidden クラスは書き換え不可のメタオブジェクトであり、同じプロパティを持つ一般オブジェクト間で共有される。オブジェクトに新たなプロパティを追加するときは、既存のプロパティと追加されたプロパティの両方の情報を持つ Hidden クラスを、既に存在すればそれを利用し、まだ存在しなければ新たに生成する。後者の場合、それまで利用していた Hidden クラスには、今後、その名前のプロパティが追加された場合に利用すべき、すなわち新たに生成した Hidden クラスへの参照を持たせる。これにより、Hidden クラスは新たなプロパティが追加されるごとに「成長」し、既に成長した Hidden クラスがあればそれを共用する。図 2.2 に、オブジェクトへのプロパティ追加により成長していく Hidden クラスの例を示す。Hidden クラスから「成長」した Hidden クラスへの参照を遷移と呼ぶ。

Hidden クラスを利用すると、オブジェクトのプロパティの格納場所に関する情報が、複数のオブジェクトで共用されるため、空間効率を改善することができる。また、インラインキャッシュ [9, 11] と呼ばれる技法を併用することで、プロパティに対するアクセスを高速化することもできる。

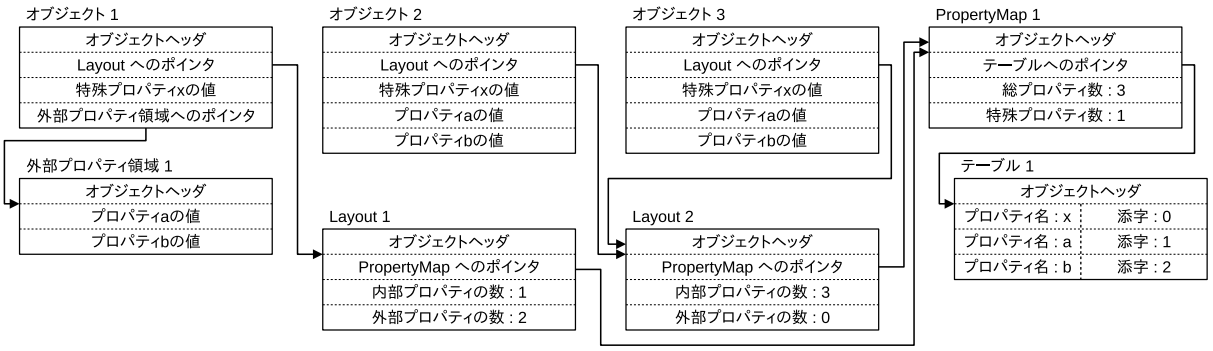


図 2.3: eJSVM における Hidden クラスを構成するメタオブジェクトの関係

2.4 eJSVM における Hidden クラスの実装

eJSVM の Hidden クラスは、実装上は `Layout` と `PropertyMap` の 2 種類のメタオブジェクトからなる。メタオブジェクトは、一般オブジェクトと同様にオブジェクトヘッダを持つ。図 2.3 に、これらのメタオブジェクトの構造を示す。`Layout` は、その `Layout` を持つオブジェクトのメモリレイアウトを記録する。具体的には内部プロパティ領域と外部プロパティ領域のサイズを表す。`PropertyMap` は、プロパティ名を Key とし、対応するプロパティの添字または遷移を Value とするテーブルを持つ。`PropertyMap` は他にプロパティの総数と、特殊プロパティの総数を持つ。特殊プロパティとは、2.2 節で説明した、内部プロパティ領域に格納される、特定の型のオブジェクトが必ず持つプロパティである。

`Layout` と `PropertyMap` が持つ情報は、一般オブジェクトそれぞれに持たせることも可能である。eJSVM では、一般オブジェクトが `Layout` を介して `PropertyMap` を持つことで、内容が同じ `PropertyMap` や `Layout` を共有し、空間効率を改善している。図 2.3 では、`Object1` と `Object2` は同じプロパティを持つが、内部プロパティの個数が異なる。そのため、2 つのオブジェクトは異なる `Layout` を持つが、同じ `PropertyMap` を共有する。

GC はこれらのメタオブジェクトを参照して、`Layout` から一般オブジェクトの内部プロパティ領域と外部プロパティ領域のサイズを、`PropertyMap` から特殊プロパティの数を、取得する。特に GC は、一般オブジェクトに対する処理を行う際に、`Layout` から `PropertyMap` を辿れる必要がある。

3　既存の GC アルゴリズム

本章では、既存の GC アルゴリズムの中から、特にスライディングコンパクションアルゴリズムである Lisp2 コンパクションと Jonkers のスレッド化コンパクションについて、それぞれの概要とそのアルゴリズムを説明する。さらにこれらを eJS へ適用する際に生じる問題点について説明する。

3.1　擬似コードの説明に共通する変数や関数の定義

本節では、以降の節でのコンパクションアルゴリズムの説明に共通する、変数や関数の定義を説明する。図 3.1 に、共通する変数と関数の定義の擬似コードを示す。

まずはじめに、変数 `HeapBegin` と変数 `HeapEnd` を定義する。これは、コンパクションの対象となるヒープ領域の先頭と末尾を指すポインタである。以降の節では、特に明示しない限り、単一のヒープを分割せずに、ひとつの領域として利用する。

次に、オブジェクトを指すポインタや、オブジェクトヘッダを指すポインタを、他方のポインタへ変換する関数として、関数 `objectToHeader` と関数 `headerToObject` を定義する。本論文で紹介するアルゴリズムの説明では、2.2 節で説明したとおり、オブジェクトヘッダがオブジェクトの直前に配置されることを想定している。オブジェクトを指すポインタはオブジェクトヘッダの次のアドレスを指すので、相互の変換はオブジェクトヘッダのサイズを、アドレスに加えたり減じたりすれば良い。

続いて、オブジェクトのサイズを取得する関数 `getObjectSize` を定義する。この関数は、オブジェクトの実装に合わせて適切に定義する。eJS では、オブジェクトヘッダの内部にサイズ情報を記録するため、オブジェクトへのポインタをオブジェクトヘッダへのポインタに変換し、オブジェクトヘッダからサイズ情報を読み出す。本論文では、オブジェクトのサイズにはオブジェクトヘッダのサイズも含まれることを想定している。

次に、オブジェクトを移動させる関数 `move` を定義する。この関数は、第一引数 `pObject` が指すオブジェクトを、第二引数 `pObjectNext` の位置へ移動させる。

最後に、生存オブジェクトへマークを付ける関数と、関連する関数を定義する。関数 `setMark` と関数 `clearMark` は、それぞれ第一引数 `pObject` が指すオブジェクトへマークを付ける、あるいはマークを取り除く。関数 `isMarked` は、第一引数 `pObject` が指すオブジェクトにマークが付けられているかを真偽値で返す。

関数 `markObjects` は、すべての生存オブジェクトにマークを付ける関数である。利用中のヒープ内のオブジェクトを指すポインタの集合をルートとする。ルートからポインタをひとつ取り出し、そのポインタが指すオブジェクトにまだマークがついていなければ、マークを付ける

```
HeapStart; // ヒープ領域の先頭
HeapEnd;   // ヒープ領域の末尾

headerToObject(pHeader) {
  int_address = pHeader;
  int_address += sizeof(ObjectHeader);
  return int_address;
}

objectToHeader(pObject) {
  int_address = pObject;
  int_address -= sizeof(ObjectHeader);
  return int_address;
}

getObjectSize(pObject) {
  header = objectToHeader(pObject);
  return header->size;
}

move(pObject, pObjectNext) {
  header = objectToHeader(pObject);
  headerNext = objectToHeader(pObjectNext);
  *headerNext = *header;
  *pObjectNext = *pObject;
}
```

```
setMark(pObject) {
  header = objectToHeader(pObject);
  header->mark = True;
}

clearMark(pObject) {
  header = objectToHeader(pObject);
  header->mark = False;
}

isMarked(pObject) {
  header = objectToHeader(pObject);
  return header->mark;
}

markObjects() {
  foreach (ptr in roots)
    markObjectsSub(ptr);
}

markObjectsSub(pObject) {
  if (!isMarked(pObject)) {
    setMark(pObject);
    foreach (ptr in pObject)
      markObjectsSub(ptr);
  }
}
```

図 3.1: マークコンパクトアルゴリズムの擬似コードに共通して現れる変数、関数の定義

(関数 `markObjectsSub` の `if`)。オブジェクトにマークを付けたら、そのオブジェクトが内部に持つ他のオブジェクトへのポインタフィールドを列挙し (関数 `markObjectsSub` の `foreach`)、それぞれについて、そのポインタが指すオブジェクトにマークを付ける処理を再帰的に繰り返す。これにより、はじめにルートから取り出したポインタが指すオブジェクトから到達可能なすべてのオブジェクトに、マークが付けられる。以上の操作を、ルートに含まれるすべてのポインタについて行うことで (関数 `markObjects` の `foreach`)、GC 後にも利用される可能性があるオブジェクトすべてにマークを付ける。

関数 `markObjects` は、マークスイープ GC のマークフェーズそのものであり、以降の節で紹介するすべての GC アルゴリズムも、この関数を実行することで生存オブジェクトへマークを付ける。

3.2 Lisp2 コンパクション

3.2.1 特徴

Lisp2 コンパクション [13] は、古くから知られる典型的なコンパクションアルゴリズムのひとつである。Lisp2 コンパクションでは、Forwarding-Pointer と呼ばれる、そのオブジェクトの移動先を記録するポインタを、すべてのオブジェクトが持つことを要求する。オブジェクトごとに Forwarding-Pointer のための領域が必要となる分、空間オーバーヘッドが生じる。しかしアルゴリズムが非常に単純であるため、実装も容易であり、またプログラムサイズを小さくすること

```
setForwardingAddress(pObject, pNext) {
  pObject->fwdptr = pNext;
}

getForwardingAddress(pObject) {
  return pObject->fwdptr;
}

updatePointer(pPointer, pNext) {
  *pPointer = pNext;
}

computeLocations() {
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    obj = headerToObject(scan);
    size = getObjectSize(obj);
    scan += size;
    if (isMarked(obj)) {
      dest = headerToObject(to);
      setForwardingAddress(obj, dest);
      to += size;
    }
  }
}
```

```
updateReferences() {
  foreach (ptr in roots) {
    dest = getForwardingAddress(ptr);
    updatePointer(&ptr, dest);
  }
  scan = HeapStart;
  while (scan < HeapEnd) {
    obj = headerToObject(scan);
    size = getObjectSize(obj);
    scan += size;
    if (isMarked(obj))
      foreach (ptr in obj) {
        dest
          = getForwardingAddress(ptr);
        updatePointer(&ptr, dest);
      }
  }
}
```

```
moveObjects() {
  scan = HeapStart;
  while (scan < HeapEnd) {
    obj = headerToObject(scan);
    size = getObjectSize(obj);
    scan += size;
    if (isMarked(obj)) {
      clearMark(obj);
      dest
        = getForwardingAddress(obj);
      move(obj, dest);
    }
  }
}

gc() {
  markObjects();
  computeLocations();
  updateReferences();
  moveObjects();
}
```

図 3.2: Lisp2 コンパクションコンパクションの擬似コード

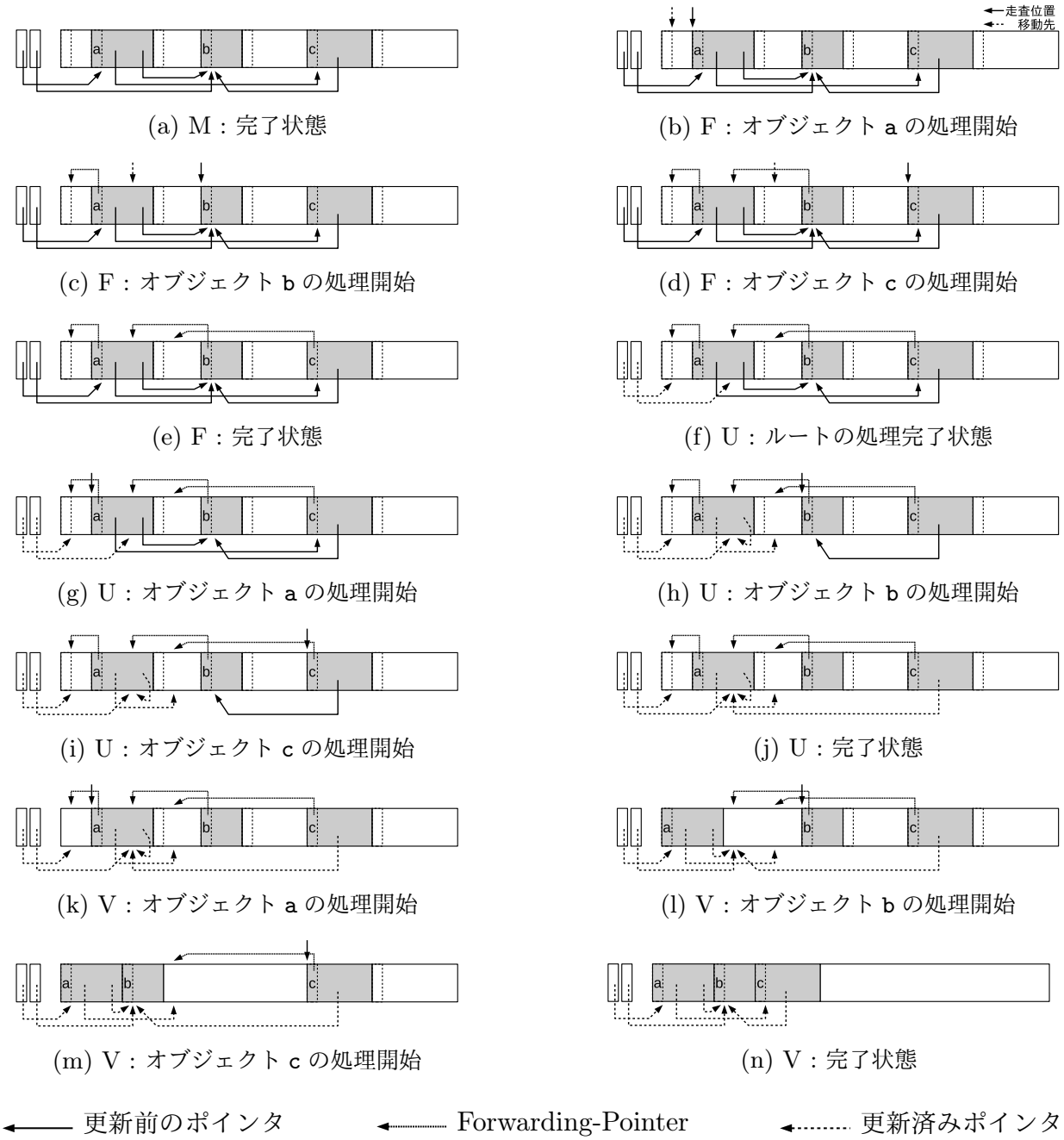
にも貢献する。

3.2.2 アルゴリズム

図 3.2 に、Lisp2 コンパクションの擬似コードを示す。はじめに、3 つの補助関数の説明を行う。1 つ目は関数 `setForwardingAddress` で、この関数は第一引数 `pObject` が指すオブジェクトの Forwarding-Pointer 領域に、第二引数 `pNext` で与えたアドレスを記録する。2 つ目は関数 `getForwardingAddress` で、この関数は第一引数 `pObject` が指すオブジェクトの Forwarding-Pointer 領域に記録されたアドレスを返す。3 つ目は関数 `updatePointer` で、この関数は第一引数 `pPointer` が指すポインタフィールドに、第二引数 `pNext` で与えたアドレスを記録することで、ポインタの指す先を更新する。

Lisp2 コンパクションアルゴリズムは、マークフェーズ、Forwarding-Pointer 更新フェーズ、ポインタ更新フェーズ、オブジェクト移動フェーズの 4 つのフェーズからなる。これらはそれぞれ、擬似コードにおける関数 `markObjects`、関数 `computeLocation`、関数 `updateReferences`、関数 `moveObjects` に対応する。図 3.3 に Lisp2 コンパクションを適用したヒープ内のオブジェクトの状態の遷移を図示する。ここで各図の左にある 2 つのポインタはルートを、その右の横長の領域はヒープを表す。また、オブジェクトのヘッダ部の値 (`a`、`b`、`c`) を用いて、個々のオブジェクトを「オブジェクト `a`」のように呼び区別する。

マークフェーズでは、生存オブジェクトにマークを付ける (3.1 節を参照)。マークフェーズが終了すると図 3.3 (a) のような状態になる。ここで灰色のオブジェクトはマーク済みであることを表す。



(a) M：完了状態　(b) F：オブジェクト a の処理開始

(c) F：オブジェクト b の処理開始　(d) F：オブジェクト c の処理開始

(e) F：完了状態　(f) U：ルートの処理完了状態

(g) U：オブジェクト a の処理開始　(h) U：オブジェクト b の処理開始

(i) U：オブジェクト c の処理開始　(j) U：完了状態

(k) V：オブジェクト a の処理開始　(l) V：オブジェクト b の処理開始

(m) V：オブジェクト c の処理開始　(n) V：完了状態

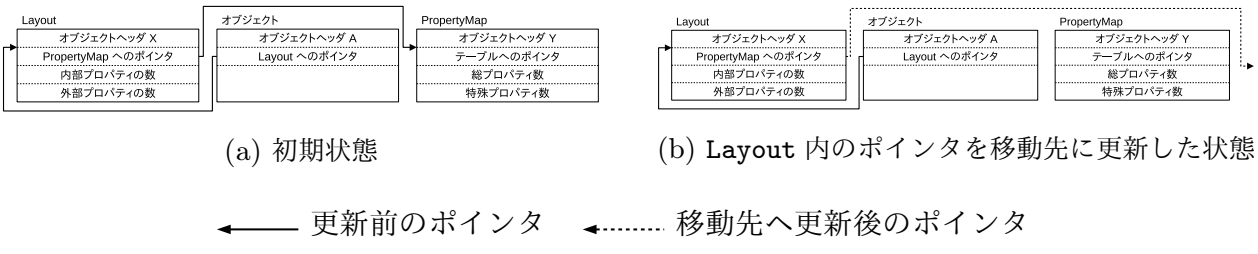
更新前のポインタ　Forwarding-Pointer　更新済みポインタ

M：マークフェーズ、F：Forwarding-Pointer 更新フェーズ、
U：ポインタ更新フェーズ、 V：オブジェクト移動フェーズ

図 3.3: Lisp2 コンパクションアルゴリズム

次に、Forwarding-Pointer 更新フェーズで、生存オブジェクトの移動先の計算と記録を行う (図 3.3 (b) – (e))。ヒープ領域全体を先頭から走査し、それまでに見つけた生存オブジェクトのサイズの総和から、次に見つかるオブジェクトの移動先を計算する。移動先のアドレスは、そのオブジェクトの Forwarding-Pointer に記録する。

続いて、ポインタ更新フェーズで、ポインタフィールドの更新を行う (図 3.3 (f) – (j))。はじ



(a) 初期状態　　(b) Layout 内のポインタを移動先に更新した状態

更新前のポインタ　移動先へ更新後のポインタ

図 3.4: ポインタの更新によってメタオブジェクトが辿れなくなる状態

めにルートに記録されたポインタを更新し (図 3.3 (f))、続いてヒープ全体を走査しながら、各生存オブジェクト内部のポインタフィールドを更新する (図 3.3 (g) – (j))。Forwarding-Pointer 更新フェーズで、すべての生存オブジェクトの Forwarding-Pointer を更新したので、ポインタの更新は、そのポインタが指すオブジェクトの Forwarding-Pointer が指す先に更新すれば良い。

最後に、オブジェクト移動フェーズで、生存オブジェクトを移動させる (図 3.3 (k) – (n))。

3.2.3 問題点

Lisp2 コンパクションアルゴリズムでは、ポインタ更新フェーズで、オブジェクト内のポインタフィールドを知るために、オブジェクトのレイアウト情報が必要である。eJSVM では、一般オブジェクトのレイアウト情報として、Layout が持つ内部プロパティ、外部プロパティの数と、PropertyMap が持つ特殊プロパティの数が必要になる。加えて、eJS が将来的にアンボックス化 [7] を実装する場合、各フィールドが、JSValue 型の値とアンボックス化された値のどちらを保持するか、という情報が PropertyMap に記録され、これも必要となる。

図 3.4 (a) のように、ヒープの先頭から順に Layout、一般オブジェクト、PropertyMap が並んでいる場合を考える。この時、はじめに Layout の内部のポインタが更新されるため、図 3.4 (b) のようになる。ここで、まだ PropertyMap は移動させていないため、Layout から PropertyMap を正しく参照することができないことに注意する。次に一般オブジェクトに対する処理を開始するが、この例の場合では、一般オブジェクトから Layout へポインタを辿ることは可能だが、Layout から PropertyMap へのポインタを正しく辿ることはできないため、特殊プロパティの数を得ることができない。これにより、一般オブジェクトに対して処理を行うことができず、Lisp2 コンパクションアルゴリズムを適用できない。

3.3 Jonkers のスレッド化コンパクション

3.3.1 特徴

Jonkers のスレッド化コンパクション [12] は、空間効率の良いコンパクションアルゴリズムのひとつである。このアルゴリズムは、スレッド化と呼ばれる操作によりポインタを逆向きにする

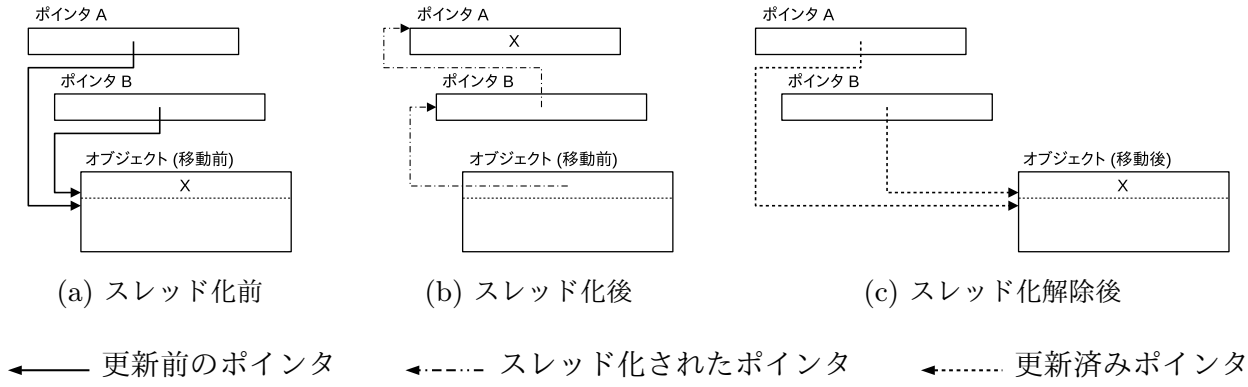


図 3.5: スレッド化のイメージ図

ことで、オブジェクトを指す (複数の) ポインタを、そのオブジェクトを指すポインタ領域の線形リストへ変形する。この線形リストをスレッド化ポインタリストと呼ぶ。スレッド化により、Lisp2 コンパクションとは対照的に、オブジェクトごとの (Forwarding-Pointer のための) 追加領域は不要となる。

図 3.5 (a)–(b) にスレッド化前後のポインタとオブジェクトの関係を示す。スレッド化により、オブジェクトヘッダはスレッド化ポインタリストの先頭となり、リストの終端となるポインタは、アドレスではなくオブジェクトヘッダの元の値を持つ (図 3.5 (b) の `X`)。Jonkers のアルゴリズムでは、リストの終端を判別するために、アドレスとオブジェクトヘッダの値は区別できる必要がある。

図 3.5 (c) にスレッド化を解除した後の状態を示す。Jonkers のアルゴリズムでは、オブジェクトの移動先のアドレスを計算した後、スレッド化ポインタリストを辿りながら、それらのポインタが指す先を移動先アドレスに更新していくことで、スレッド化を解除する。このスレッド化の解除により、スレッド化を行う前にそのオブジェクトを指していたすべてのポインタが、オブジェクトの移動先アドレスを指すよう更新される。

3.3.2 アルゴリズム

図 3.6 に、Jonkers のスレッド化コンパクションアルゴリズムの擬似コードを示す。はじめに、補助関数の説明を行う。関数 `getHeaderValue` と関数 `setHeaderValue` は、それぞれオブジェクトヘッダの値を取得、設定する関数である。これらの関数では、オブジェクトヘッダ内のサイズ情報やマーク情報などを無視して、オブジェクトヘッダ内の先頭の 1 ワードを操作する。関数 `isThreaded` は、関数 `getHeaderValue` で取得したオブジェクトヘッダの値が、スレッド化されたポインタへのアドレスの場合に真を返す。関数 `thread` は、オブジェクトを指すポインタをスレッド化する。関数 `unthread` は、スレッド化ポインタリストを辿りながら、そのオブジェクトを指すポインタを移動先アドレス `dst` に更新する。関数 `threadAllPointers` は、`pObject` が指すオブジェクト内部の全てのポインタをスレッド化する。

```
getHeaderValue(pObject) {
  pHeader = objectToHeader(pObject);
  return *pHeader;
}

setHeaderValue(pObject, ref) {
  pHeader = objectToHeader(pObject);
  *pHeader = ref;
}

isThreaded(hValue) {
  return ((hValue & 1) == 0);
}

thread(ref) {
  pObject = *ref;
  *ref = getHeaderValue(pObject);
  setHeaderValue(pObject, ref);
}

threadAllPointers(pObject) {
  foreach (ptr in pObject)
    thread(&ptr);
}
```

```
unthread(pObject, dst) {
  hValue = getHeaderValue(pObject);
  while (isThreaded(hValue)) {
    next = *hValue;
    *hValue = dst;
    hValue = next;
  }
  setHeaderValue(pObject, hValue);
}

updateForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
      to += size;
    }
  }
}
```

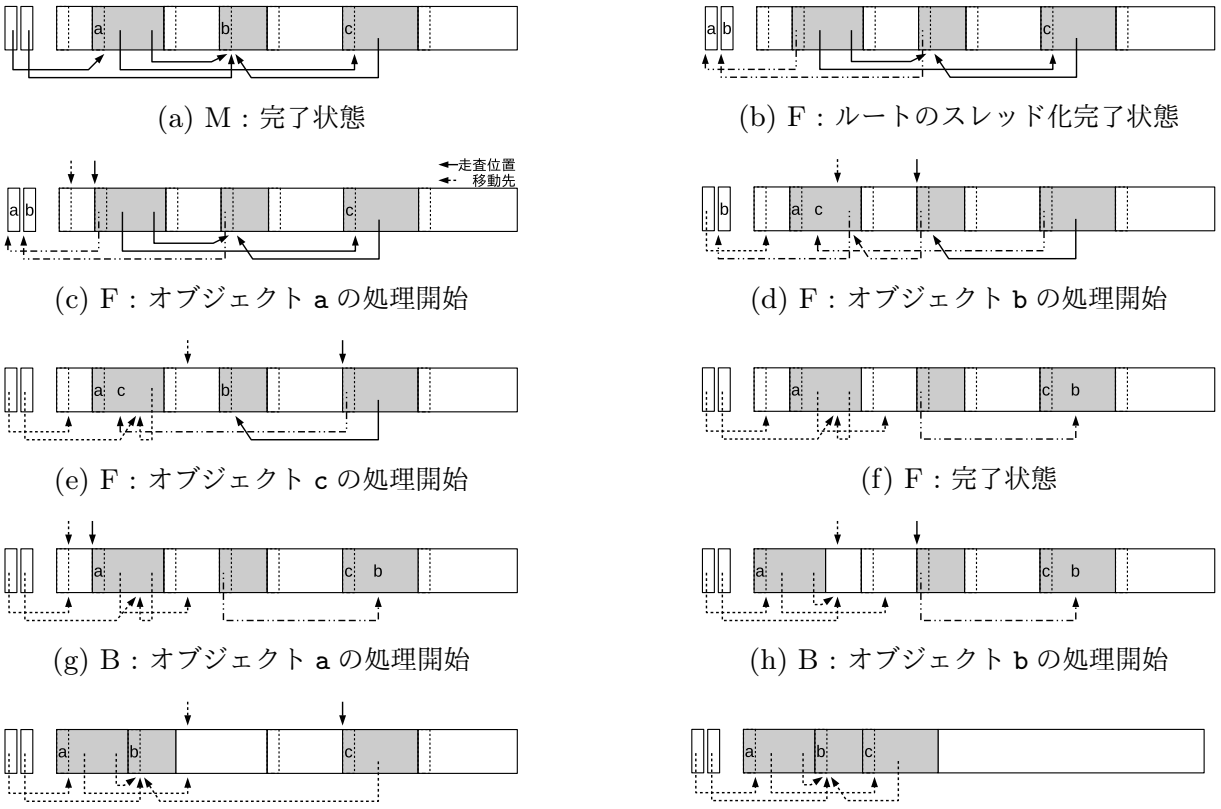
```
updateBackwardReferences() {
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      move(p, dest);
      to += size;
    }
  }
}

gc() {
  markObjects();
  updateForwardReferences();
  updateBackwardReferences();
}
```

図 3.6: Jonkers のスレッド化コンパクションの擬似コード

Jonkers のアルゴリズムは、マークフェーズ、後向きポインタ更新フェーズ、前向きポインタ更新フェーズの 3 つのフェーズに分けられる。これらはそれぞれ、擬似コードにおける関数 `markObjects`、関数 `updateForwardReferences`、関数 `updateBackwardReferences` に対応する。図 3.7 に、このアルゴリズムを適用した時のヒープの変化を示す。各図が表すルートや生存オブジェクトなどは、図 3.3 と同じである。

まずマークフェーズで生存オブジェクトにマークを付ける。これは Lisp2 コンパクションアルゴリズムと全く同じ手順である。マークフェーズが終了すると図 3.7 (a) のような状態になる。ここで灰色のオブジェクトはマーク済みであることを表す。

次に後向きポインタ更新フェーズで、オブジェクト自身よりも前方から自身を指しているポインタ (後向きポインタ) を、そのオブジェクトの移動先を指すように更新する (図 3.7 (b)–(f))。はじめにルートに含まれるすべてのポインタをスレッド化する (図 3.7 (b))。次にヒープを先頭から走査し、生存オブジェクトを探す。見つかった生存オブジェクトに対して次の操作を行う。まず、既にスレッド化された自身を指すポインタを、自身の移動先で更新することで、スレッド化を解除する。自身を指すポインタは、図 3.5 (b) で示したように、オブジェクトヘッダからスレッド化ポインタリストを辿って見つけることができる。また、オブジェクトの移動先は、それまでに見つかった生存オブジェクトの大きさの総和から求めることができる。次に、自身の持つすべてのポインタをスレッド化する。以上の操作を、ヒープ内のすべての生存オブジェクトに、ヒープの前方から順に適用していく。後向きポインタ更新フェーズが終了すると図 3.7 (f) の状態となる。ここでは、すべての後向きポインタがそれぞれのオブジェクトの移動先で更新され、オブジェクト自身よりも後方から自身を指しているポインタ (前向きポインタ) はすべてスレッ



(a) M：完了状態　(b) F：ルートのスレッド化完了状態

(c) F：オブジェクト a の処理開始　(d) F：オブジェクト b の処理開始

(e) F：オブジェクト c の処理開始　(f) F：完了状態

(g) B：オブジェクト a の処理開始　(h) B：オブジェクト b の処理開始

(i) B：オブジェクト c の処理開始　(j) B：完了状態

更新前のポインタ　スレッド化されたポインタ　更新済みポインタ

M：マークフェーズ、F：後向きポインタ更新フェーズ、B：前向きポインタ更新フェーズ

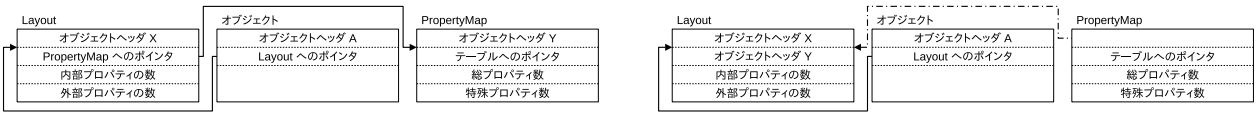
図 3.7: Jonkers のスレッド化コンパクションアルゴリズム

ド化された状態となる。

最後に前向きポインタ更新フェーズで、前向きポインタを更新しながら、オブジェクトの移動を行う (図 3.7 (g)–(j))。後向きポインタ更新フェーズと同様に、ヒープを先頭から走査し、生存オブジェクトを見つけるたびに次の操作を行う。まず、既にスレッド化された自身を指すポインタを、自身の移動先を指すように更新することで、スレッド化を解除する。次に、オブジェクト自身を移動先へ移動させる。前向きポインタ更新フェーズが終了すると図 3.7 (j) の状態となり、生存オブジェクトがヒープの端に集められ、ポインタも正しく更新されていることがわかる。

3.3.3 問題点

Jonkers のアルゴリズムでは、マークフェーズと後向きポインタ更新フェーズで、Lisp2 コンパクションアルゴリズムと同様に、オブジェクトのレイアウト情報が必要である。一般オブジェクトと Hidden クラスを構成するメタオブジェクトがヒープ内に混在する場合、Lisp2 コンパク



(a) 初期状態　　(b) `Layout` 内のポインタをスレッド化した状態

更新前のポインタ　スレッド化されたポインタ

図 3.8: スレッド化によってメタオブジェクトが辿れなくなる状態

ションと似た原因で、Jonkers のアルゴリズムもまた、適用することができない場合がある。

図 3.8 (a) のように、ヒープの先頭から順に `Layout`、一般オブジェクト、`PropertyMap` が並んでいる場合を考える。この時、はじめに `Layout` 内部のポインタがスレッド化されるため、図 3.8 (b) のようになる。ここで、ポインタがスレッド化されると、元の方向にポインタを辿ることはできなくなることに注意する。次に一般オブジェクトに対する処理を開始するが、この例の場合も、Lisp2 コンパクションアルゴリズムと同様に、`Layout` から `PropertyMap` へポインタを辿ることはできないため、特殊プロパティの数を得ることができない。これにより、一般オブジェクトに対して処理を行うことができず、Jonkers のアルゴリズムを適用できない。

4　eJS への GC アルゴリズムの適用方法

本章では、まず、第 3 章で紹介した 2 種類の GC アルゴリズムそれぞれについて、そのアルゴリズムを eJS へ適用するための、最も単純な拡張アルゴリズムを示す。次に Jonkers のスレッド化コンパクションの単純な拡張の他に、異なる戦略で Jonkers のアルゴリズムを拡張した、2 種類の拡張アルゴリズムを示す。本章で示すアルゴリズムは、いずれも eJS に適用可能なアルゴリズムであるが、空間オーバーヘッドや明らかな時間オーバーヘッドを生じる。

4.1　Lisp2 の単純な拡張

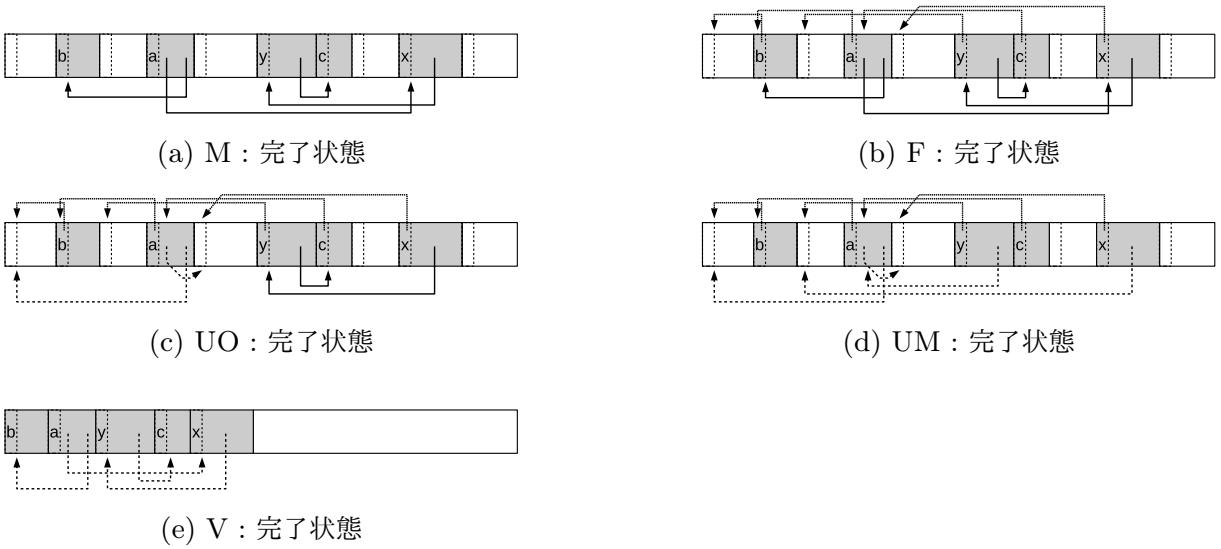
3.2.3 節で説明したとおり、一般オブジェクトとメタオブジェクトが混在するヒープにおける Lisp2 アルゴリズムの問題点は、一般オブジェクトとメタオブジェクトの処理順序が保証されていないため、まだ参照できる状態のままにしておくべきメタオブジェクトへのポインタを、ポインタの更新により参照できない状態にしてしまう可能性があることに起因する。この問題を解決する単純な方法として、ここでは一般オブジェクトとメタオブジェクトのポインタ更新タイミングの分離を考える。以降は、この方法を単純拡張 Lisp2 アルゴリズムと呼ぶ。

図 4.1 に、単純拡張 Lisp2 アルゴリズムの擬似コードを示す。単純拡張 Lisp2 アルゴリズムでは、Lisp2 コンパクションアルゴリズムのポインタ更新フェーズを一般オブジェクトポインタ更新フェーズ (関数 `updateOrdReferences`) とメタオブジェクトポインタ更新フェーズ (関

```
updateOrdReferences() {
  foreach (ptr in roots) {
    dest = getForwardingAddress(ptr);
    updatePointer(&ptr, dest);
  }
  scan = HeapStart;
  while (scan < HeapEnd) {
    obj = headerToObject(scan);
    size = getObjectSize(obj);
    scan += size;
    if (isMetaObject(obj))
      contniue;
    if (isMarked(obj))
      foreach (ptr in obj) {
        dest = getForwardingAddress(ptr);
        updatePointer(&ptr, dest);
      }
  }
}
```

```
updateMetaReferences() {
  scan = HeapStart;
  while (scan < HeapEnd) {
    obj = headerToObject(scan);
    size = getObjectSize(obj);
    scan += size;
    if (isOrdObject(obj))
      contniue;
    if (isMarked(obj))
      foreach (ptr in obj) {
        dest = getForwardingAddress(ptr);
        updatePointer(&ptr, dest);
      }
  }
}

gc() {
  markObjects();
  computeLocations();
  updateOrdReferences();
  updateMetaReferences();
  moveObjects();
}
```

図 4.1: 単純拡張 Lisp2 アルゴリズムコンパクションの擬似コード



M : マークフェーズ、F : Forwarding-Pointer 更新フェーズ、
UO : 一般オブジェクトポインタ更新フェーズ、UM : メタオブジェクトポインタ更新フェーズ、
V : オブジェクト移動フェーズ

図 4.2: 単純拡張 Lisp2 アルゴリズム

数 `updateMetaReference`) のふたつにわける。一般オブジェクトポインタ更新フェーズでは、Lisp2 コンパクションのポインタ更新フェーズと同様に、生存オブジェクトを見つけながら、その内部のポインタを更新するが (`while (scan < HeapEnd)`)、メタオブジェクトについてはポインタの更新を後回しにする (`if (isMetaObject(obj)) continue;`)。逆にメタオブジェクトポインタ更新フェーズでは、発見した生存メタオブジェクトの内部のポインタは更新するが、一般オブジェクトに対しては何もしない (`if (isOrdObject(obj)) continue;`)。ポインタ更新フェーズをふたつのフェーズに分けることにより、メタオブジェクトを参照する必要のある一般オブジェクトポインタ更新フェーズで、常にメタオブジェクトのポインタを正しく辿れることが保証される。図 4.2 に、単純拡張 Lisp2 アルゴリズムを適用したヒープの例を示す。

単純拡張 Lisp2 アルゴリズムは 3.2.3 節で述べた問題点を解消するが、一般オブジェクトとメタオブジェクトの処理を別々にするために、ヒープのスキャン回数が 1 回増加し、明らかな時間オーバーヘッドを伴う。4.3 節や 4.4 節で述べる単純拡張 Jonkers アルゴリズムの改良と同様の手法で、この時間オーバーヘッドを削減することは可能だが、その場合は追加の空間オーバーヘッドが生じる。

```
updateOrdForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      if (isOrdObject(p))
        threadAllPointers(p);
      to += size;
    }
  }
}
```

```
updateMetaForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      if (isMetaObject(p))
        threadAllPointers(p);
      to += size;
    }
  }
}
```

```
gc() {
  markObjects();
  updateOrdForwardReferences();
  updateMetaForwardReferences();
  updateBackwardReferences();
}
```

図 4.3: 単純拡張 Jonkers アルゴリズムの擬似コード

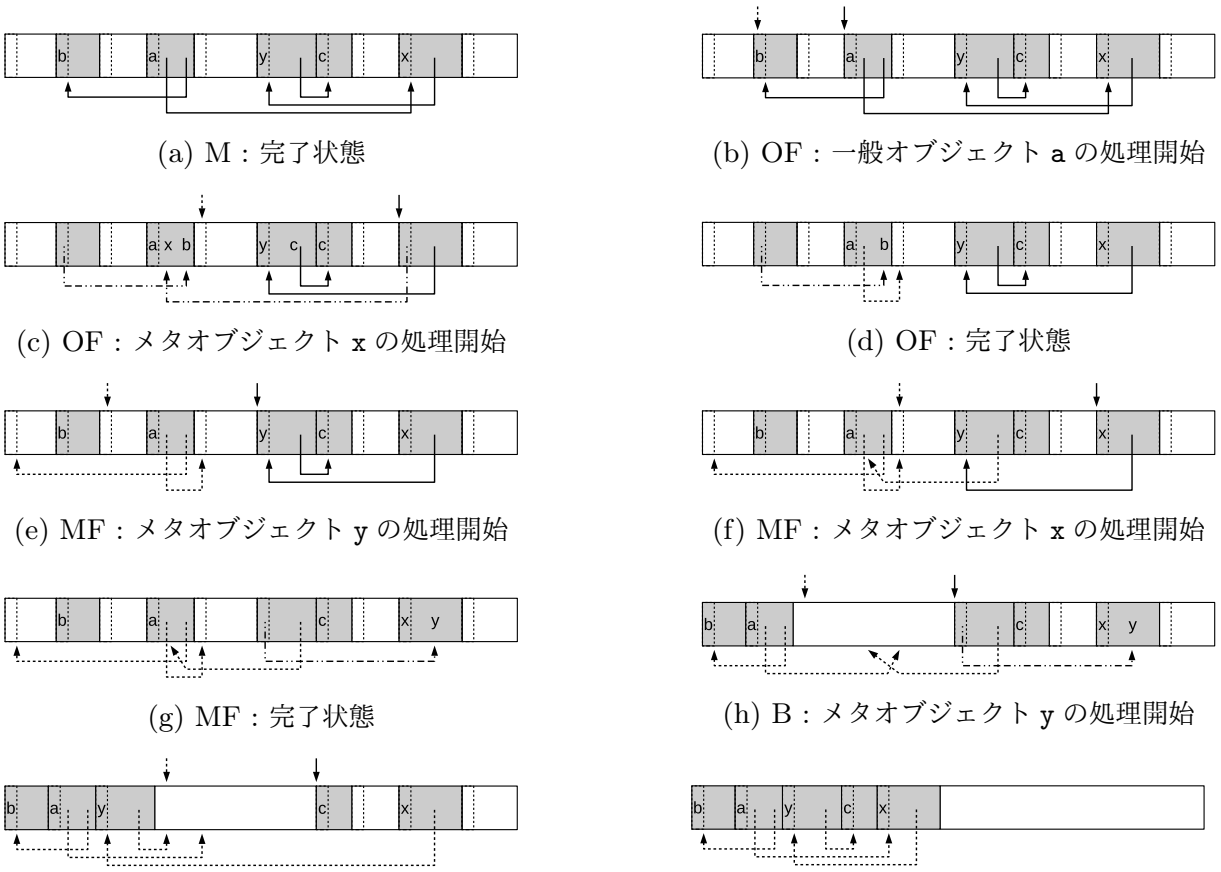
4.2 Jonkers の単純な適用

3.3.3 節で説明した問題も、単純拡張 Lisp2 アルゴリズムと同様に、一般オブジェクトとメタオブジェクトの処理タイミングを分けることで解消することができる。以下では、この方法を単純拡張 Jonkers アルゴリズムと呼ぶ。

図 4.3 に、単純拡張 Jonkers アルゴリズムの擬似コードを示す。単純拡張 Jonkers アルゴリズムでは、Jonkers の後向きポインタ更新フェーズを一般オブジェクト後向きポインタ更新フェーズ (関数 `updateOrdForwardReferences`) とメタオブジェクト後向きポインタ更新フェーズ (関数 `updateMetaForwardReferences`) にわける。一般オブジェクト後向きポインタ更新フェーズとメタオブジェクト後向きポインタ更新フェーズでは、Jonkers の後向きポインタ更新フェーズと同様の方法で、まずオブジェクトのスレッド化を解除し、一般オブジェクト後向きポインタ更新フェーズでは生存一般オブジェクト内部のポインタを (`if (isOrdObject(p))`)、メタオブジェクト後向きポインタ更新フェーズでは生存メタオブジェクト内部のポインタを (`if (isMetaObject(p))`)、それぞれスレッド化する。

図 4.4 に、単純拡張 Jonkers アルゴリズムを適用したヒープの例を示す。一般オブジェクト後向きポインタ更新フェーズが完了すると (図 4.4 (d))、一般オブジェクト内部の全ての後向きポインタが更新される。メタオブジェクト後向きポインタ更新フェーズが完了すると (図 4.4 (g))、一般オブジェクト内部の全ての前向きポインタと、メタオブジェクト内部の全ての後向きポインタが更新される。ポインタ更新フェーズでは、メタオブジェクト内部の前向きポインタを更新しながら、全てのオブジェクトを移動する。

単純拡張 Jonkers アルゴリズムは 3.3.3 節で説明した問題を解消するが、ヒープスキャン回数が 1 回増加するため、明らかな時間オーバーヘッドを伴う。ヒープのスキャン回数が増加した原因は 2 つある。1 つ目は、一般オブジェクトとメタオブジェクトの処理を別々にするために、



M：マークフェーズ、OF：一般オブジェクト後向きポインタ更新フェーズ、
MF：メタオブジェクト後向きポインタ更新フェーズ、B：前向きポインタ更新フェーズ

図 4.4: 単純拡張 Jonkers アルゴリズム

ヒープのスキャン回数を増やしたことである。2 つ目は、一般オブジェクトとメタオブジェクトのヒープ内での順序が保証されておらず、一部の後向きポインタが更新されないまま残ってしまうため、これを更新するためのスキャンが必要なことである。

4.3 メタオブジェクトスキャンを後回しにした Jonkers の拡張

4.2 節とは異なる方法でヒープをスキャンし、ヒープ全体を 1 回だけスキャンして後向きポインタ更新フェーズを行う方法も考える。ここでは、生存メタオブジェクトを指すポインタからなるキューを作成する。以下では、このキューを用いる方法をキュー併用拡張 Jonkers アルゴリズムと呼ぶ。キュー併用拡張 Jonkers アルゴリズムは、後向きポインタ更新フェーズ相当の処理は 1 回のヒープのスキャンで行うが、代わりに前向きポインタ更新フェーズ相当の処理を 2 回の

```
updateOrdForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      if (isOrdObject(p))
        threadAllPointers(p);
      else
        enqueue(queue, p, dest);
      to += size;
    }
  }
}
```

```
updateReferences() {
  scan = to = HeapStart;
  while (scan < HeapEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      to += size;
    }
  }
}
```

```
updateMetaForwardReferences() {
  while (!empty(queue)) {
    p, dest = dequeue(queue);
    unthread(p, dest);
    threadAllPointers(p);
  }
}

gc() {
  markObjects();
  updateOrdForwardReferences();
  updateMetaForwardReferences();
  updateReferences();
  moveObjects();
}
```

図 4.5: キュー併用拡張 Jonkers アルゴリズムの擬似コード

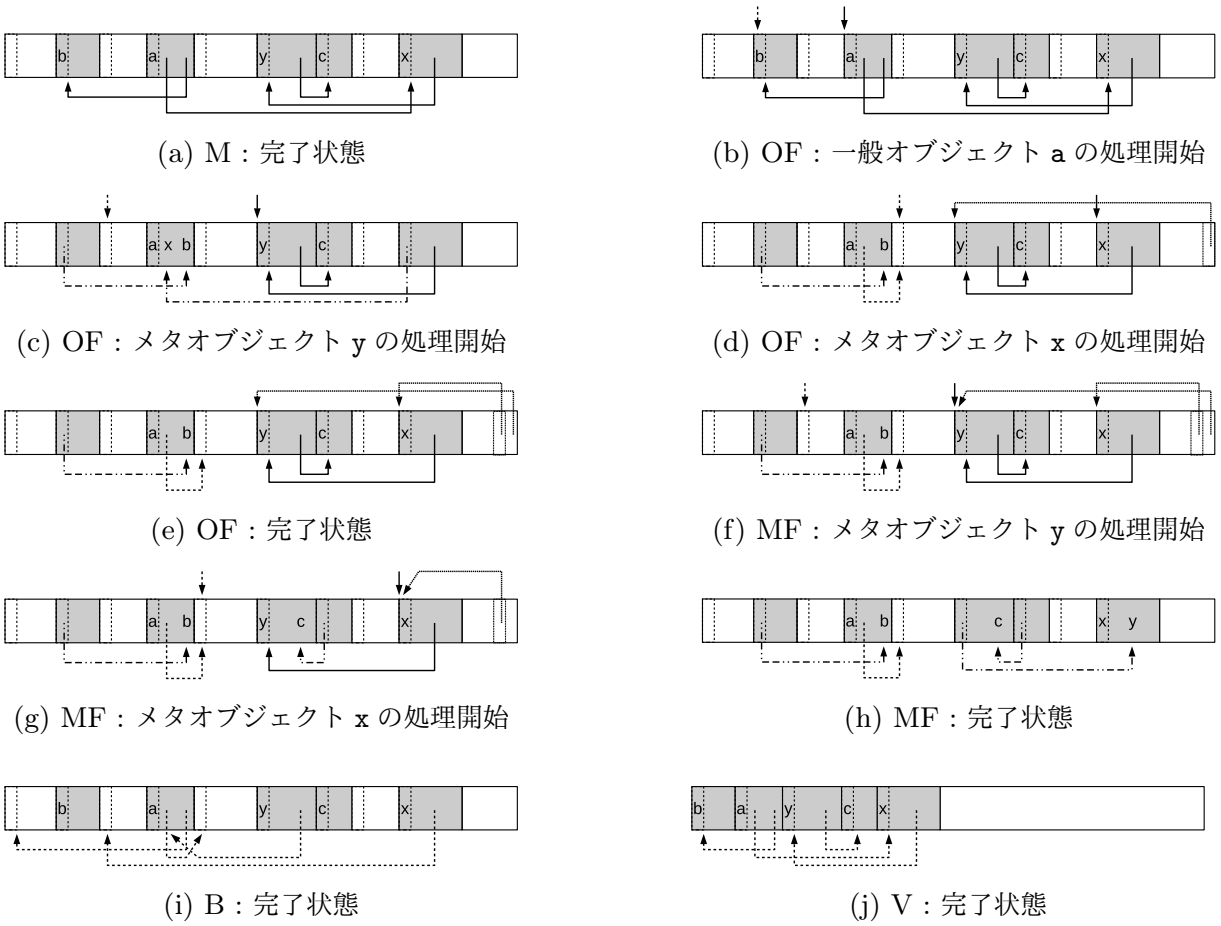
ヒープのスキャンで行う。そのため、全体で 3 回ヒープのスキャンを行うこととなり、単純拡張 Jonkers アルゴリズムとヒープのスキャン回数は同じとなる。

図 4.5 に、キュー併用拡張 Jonkers アルゴリズムの擬似コードを示す。また、図 4.6 に、キュー併用拡張 Jonkers アルゴリズムを適用したヒープの例を示す。キュー併用拡張 Jonkers アルゴリズムの基本的な戦略は、単純拡張 Jonkers アルゴリズムにおける一般オブジェクト後向きポインタ更新フェーズで、生存メタオブジェクト内部のポインタ更新をスキップする際に、キューにこの生存メタオブジェクトへのアドレスを追加しておくことで (図 4.6 (c) – (e))、メタオブジェクト後向きポインタ更新フェーズにおいてキューを走査するだけで、過不足無く生存メタオブジェクトの処理できるようにする (図 4.6 (f) – (h))、というものである。そのために、Jonkers のアルゴリズムの後向きポインタ更新フェーズを、一般オブジェクト後向きポインタ更新フェーズ (関数 `updateOrdForwardReferences`) とメタオブジェクト後向きポインタ更新フェーズ (関数 `updateMetaForwardReferences`) に分け、一般オブジェクト後向きポインタ更新フェーズで一般オブジェクトの処理を行いながらメタオブジェクトへのポインタをキューに追加し、メタオブジェクト後向きポインタ更新フェーズではキューを走査することでメタオブジェクトの処理を行う。

実際には、キューには生存メタオブジェクトへのアドレスとともに、そのメタオブジェクトの移動先アドレスも記録する (関数 `enqueue`)。なぜなら、キューを走査しても、そのメタオブジェクトの移動先は計算できないからである。

この方法では、新たにキュー用の領域が追加で必要となり、空間オーバーヘッドとなる。キュー用の領域はヒープの外に取ることもできる。この場合は利用可能なヒープサイズは維持できるが、ヒープ外に GC のみが利用する領域が存在することとなる。

ヒープ内部にキューを構築する場合、メタオブジェクトひとつ当たり 2 ワード分の空間オーバーヘッドでキュー用の領域を確保することができる。このキューは、ヒープ中に存在するメタ



M：マークフェーズ、OF：一般オブジェクト後向きポインタ更新フェーズ、
MF：メタオブジェクト後向きポインタ更新フェーズ、B：ポインタ更新フェーズ、
V：オブジェクト移動フェーズ

図 4.6: キュー併用拡張 Jonkers アルゴリズム

オブジェクトの個数分の要素数が保持できれば十分である。そこで、メタオブジェクトのアロケーションの際に 2 ワード余分に領域を確保する。ただし余分に確保する領域は、ヒープの末尾とする。こうすることで、GC の際に、ヒープの末尾にキュー用の領域が十分に確保される。GC の最後には、GC を行う過程で実際にキューに詰められた要素分、ヒープの末尾からキュー用の領域を改めて確保しなおす。これにより、次の GC の際に必要となるキュー用の領域が十分に確保される。

キュー併用拡張 Jonkers アルゴリズムは、もとの Jonkers のアルゴリズムにおける後向きポインタ更新フェーズでのポインタのスレッド化を、ヒープ 1 回のスキャンで完了できる点は、単純拡張 Jonkers アルゴリズムよりも優れる。しかしながら、ポインタのスレッド化とスレッド化の解除を行うタイミングを変えたため、メタオブジェクト後向きポインタ更新フェーズを完了した

時点で、メタオブジェクトの中に存在する、一般オブジェクトを指す後向きポインタが、スレッド化されたままとなる。後向きポインタが全て更新されるまで、オブジェクトを移動させることはできないため、結局、メタオブジェクトポインタ更新フェーズが、スレッド化を解除するポインタ更新フェーズ (関数 `updateReferences`) と、オブジェクトの移動を行うオブジェクト移動フェーズ (関数 `moveObjects`) に分けられ、それぞれでヒープ全体をスキャンすることとなる。

キュー併用拡張 Jonkers アルゴリズムは、ヒープ内部にキューを構築した場合、Jonkers のアルゴリズムと比較して、ヒープのスキャン回数が 1 回多いため、時間オーバーヘッドがあり、メタオブジェクト当たり 2 ワード分の空間オーバーヘッドもある。

この改良方法は、4.1 節で示した単純拡張 Lisp2 アルゴリズムにも応用することができる。単純拡張 Lisp2 アルゴリズムはポインタの更新とオブジェクトの移動を、異なるフェーズで行うため、キュー併用拡張 Jonkers アルゴリズムと同じ改良方法により、ヒープのスキャン回数を 1 回減らせる。ただし、時間オーバーヘッドを改善する代わりに、空間オーバーヘッドが生じる。

4.4 領域を分割した Jonkers の拡張

4.2 節で説明したふたつ目の問題点を解消するために、一般オブジェクトとメタオブジェクトのヒープ内の順序を保証する方法を考える。ここでは、最も単純な方法として、一般オブジェクト用の領域とメタオブジェクト用の領域を分割し、それぞれを独立した領域として扱う。以下では、この領域の分割を用いる方法を領域分割拡張 Jonkers アルゴリズムと呼ぶ。

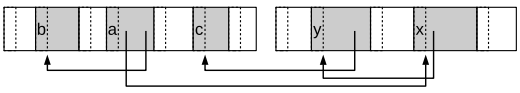
図 4.7 に、領域分割拡張 Jonkers アルゴリズムの擬似コードを示す。領域分割拡張 Jonkers アルゴリズムでは、GC の度に 2 つの領域それぞれに Jonkers のアルゴリズムを適用する。ただし、各フェーズの実行順序を適切に制御する。まずはじめにマークフェーズを行う。ルートから再帰的に生存オブジェクトにマークを付けると、2 つの領域に存在するすべての生存オブジェクトにマークが付く。続いて一般オブジェクト用の領域の後向きポインタ更新フェーズを行う (1 回目の `updateForwardReferencesSub`)。この間はまだメタオブジェクトに対する処理が行われないため、問題なく一般オブジェクトを処理することができる。次にメタオブジェクト用の領域の後向きポインタ更新フェーズを行う (2 回目の `updateForwardReferencesSub`)。ここで、一般オブジェクトは擬似的にメタオブジェクトよりもヒープ前方に配置されていることになり、メタオブジェクト用の領域の後向きポインタ更新フェーズが完了した時点で、すべての生存オブジェクトの後向きポインタは移動先アドレスに更新される (図 4.8 (c))。最後に一般オブジェクト用の領域に前向きポインタ更新フェーズを行ってから (1 回目の `updateBackwardReferencesSub`)、メタオブジェクト用の領域に前向きポインタ更新フェーズを行う (2 回目の `updateBackwardReferencesSub`)。

領域分割拡張 Jonkers アルゴリズムは Jonkers のアルゴリズムと比べて、時間オーバーヘッドは生じない。しかしながら、予めヒープを 2 つの領域に分割する必要が有り、各領域の大きさを動的に変化させることはできない。2 つの領域の大きさが適切になるよう、予めヒープを分割し

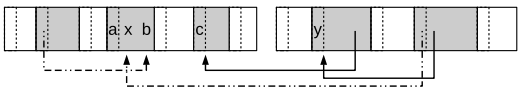
```
updateForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);

  s = HeapOrdAreaStart;
  e = HeapOrdAreaEnd;
  updateForwardReferencesSub(s, e);

  s = HeapMetaAreaStart;
  e = HeapMetaAreaEnd;
  updateForwardReferencesSub(s, e);
}

updateForwardReferencesSub(start, end) {
  scan = to = start;
  while (scan < end) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
      to += size;
    }
  }
}
```

```
updateBackwardReferences() {
  s = HeapOrdAreaStart;
  e = HeapOrdAreaEnd;
  updateBackwardReferencesSub(s, e);

  s = HeapMetaAreaStart;
  e = HeapMetaAreaEnd;
  updateBackwardReferencesSub(s, e);
}

updateBackwardReferencesSub(start, end) {
  scan = to = start;
  while (scan < end) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      move(p, dest);
      to += size;
    }
  }
}

gc() {
  markObjects();
  updateForwardReferences();
  updateBackwardReferences();
}
```

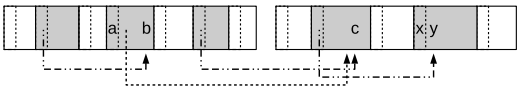
図 4.7: 領域分割拡張 Jonkers アルゴリズムの擬似コード



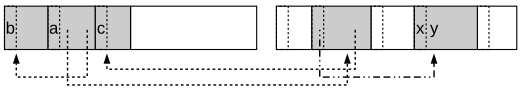
(a) M：完了状態



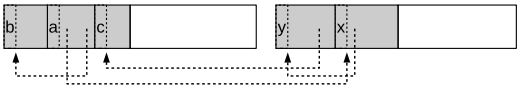
(b) F：一般オブジェクト領域の処理完了状態



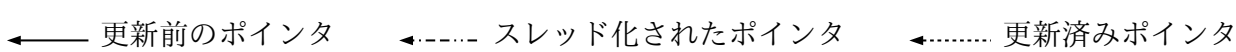
(c) F：メタオブジェクト領域の処理完了状態



(d) B：一般オブジェクト領域の処理完了状態



(e) B：メタオブジェクト領域の処理完了状態



M：マークフェーズ、F：後向きポインタ更新フェーズ、B：前向きポインタ更新フェーズ

図 4.8: 領域分割拡張 Jonkers アルゴリズム

ておくことは非常に難しい。加えて、片方の領域には十分に空きがあるが、他方の領域の空きは不足しているという、空間レベルの断片化が発生し、空間オーバーヘッドとなる。

5 Fusuma アルゴリズム

本研究では、3.3.3 節で述べた問題点を解消した、Fusuma アルゴリズムを提案する。このアルゴリズムは、一般オブジェクトとメタオブジェクトの両方を単一のヒープに配置して、コンパクションを行う。

Fusuma は、一般オブジェクトとメタオブジェクトをヒープの異なる端に割り当て、閉じた襖を開けるように、それぞれをヒープの両端にスライドさせる。このようなオブジェクトの動かし方から連想して、このアルゴリズムに Fusuma という名前を付けた。

Fusuma アルゴリズムは、メタオブジェクトの持つ一般オブジェクトへのポインタを GC が辿る必要がある場合を除き、メタオブジェクトが一般オブジェクトへのポインタを持っていても問題ない。例えば、eJSVM において、Hidden クラスは一般オブジェクトである文字列オブジェクトへのポインタを持つ。しかし、eJSVM の GC はこのポインタを辿ることはない。

5.1 概要

Fusuma アルゴリズムは、Jonkers のコンパクションアルゴリズムと同じ 3 つのフェーズからなる。このアルゴリズムの基本的なアイデアは、後向きポインタ更新フェーズで、メタオブジェクト以外のすべてのオブジェクトを先に処理し、Hidden クラスを参照する必要がなくなってからメタオブジェクトを処理する、というものである。このアルゴリズムでは、すべてのメタオブジェクトのレイアウトは静的に決定できる必要がある。

ヒープ中の一般オブジェクト、メタオブジェクトを区別して処理するために、ヒープの先頭から末尾に向かって順に一般オブジェクトを割り当て、ヒープの末尾から先頭に向かって順にメタオブジェクトを割り当てるようにする (図 5.1)。一般オブジェクトを割り当てる領域を一般オブジェクト領域、メタオブジェクトを割り当てる領域をメタオブジェクト領域と呼ぶ。コンパクションでは、一般オブジェクトはヒープの先頭に向けて、メタオブジェクトはヒープの末尾に向けてスライドさせる。そのため、後向きポインタ更新フェーズと前向きポインタ更新フェーズで、一般オブジェクト領域はヒープの先頭から、メタオブジェクト領域はヒープの末尾から、順に走査する。

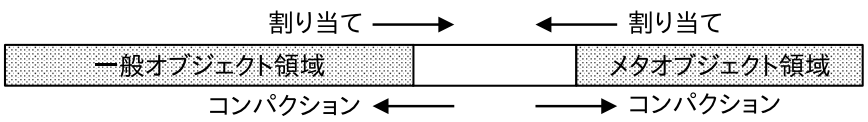


図 5.1: Fusuma におけるヒープ内のオブジェクトの配置方向とスライド方向

```
// 一般オブジェクト領域の先頭
HeapOrdAreaStart;
// 一般オブジェクト領域の末尾
HeapOrdAreaEnd;
// メタオブジェクト領域の先頭
HeapMetaAreaStart;
// メタオブジェクト領域の末尾
HeapMetaAreaEnd;

gc() {
  markObjects();
  updateForwardReferences();
  updateBackwardReferences();
}

getMetaObjectSize(pEndOfMeta) {
  pBtag = pEndOfMeta - 1;
  return *pBTag;
}

metaObjectTop(pEndOfMeta, size) {
  return pEndOfMeta - size;
}
```

```
updateForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr);
  scan = to = HeapOrdAreaStart;
  while (scan < HeapOrdAreaEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
      to += size;
    }
  }
  scan = to = HeapMetaAreaEnd;
  while (HeapMetaAreaStart < scan) {
    size = getMetaObjectSize(scan);
    scan = metaObjectTop(scan, size);
    p = headerToObject(scan);
    if (isMarked(p)) {
      to = metaObjectTop(to, size);
      dest = headerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
    }
  }
}
```

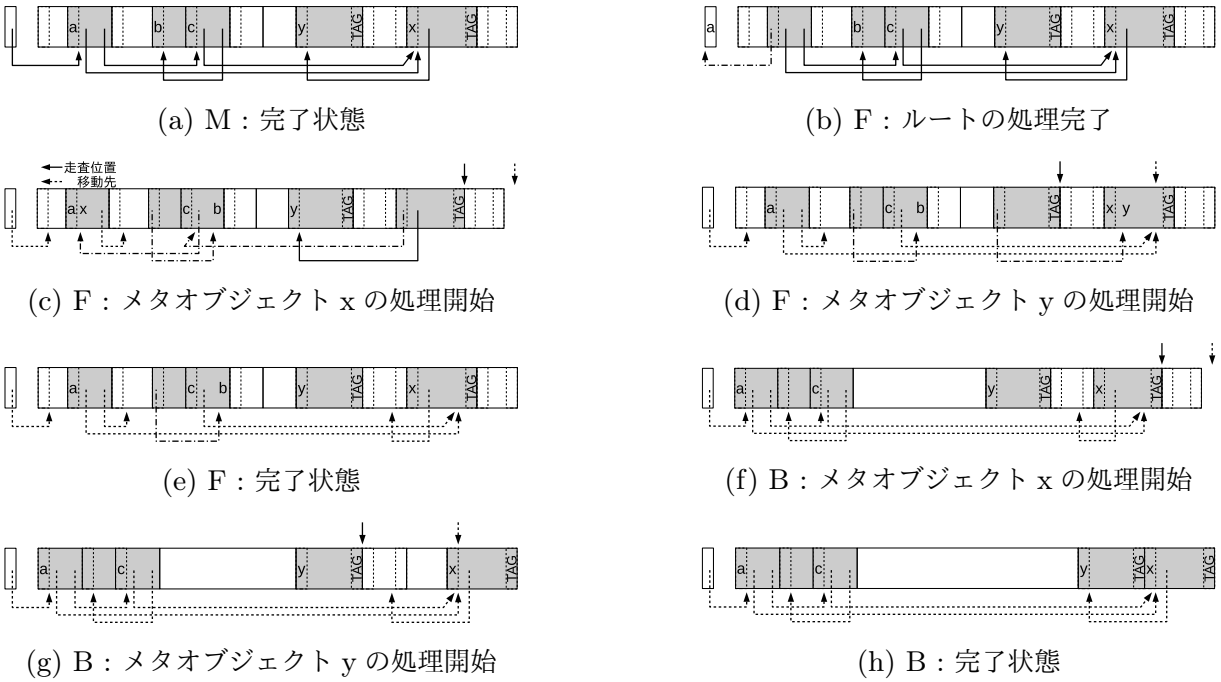
```
updateBackwardReferences() {
  scan = to = HeapOrdAreaStart;
  while (scan < HeapOrdAreaEnd) {
    p = headerToObject(scan);
    size = getObjectSize(p);
    scan += size;
    if (isMarked(p)) {
      dest = headerToObject(to);
      unthread(p, dest);
      move(p, dest);
      to += size;
    }
  }
  scan = to = HeapMetaAreaEnd;
  while (HeapMetaAreaStart < scan) {
    size = getMetaObjectSize(scan);
    scan = metaObjectTop(scan, size);
    p = headerToObject(scan);
    if (isMarked(p)) {
      to = metaObjectTop(to, size);
      dest = headerToObject(to);
      unthread(p, dest);
      move(p, dest);
    }
  }
}
```

図 5.2: Fusuma の擬似コード

5.2 アルゴリズム

図 5.2 に、Fusuma の擬似コードを示す。Fusuma ではメタオブジェクト領域を逆順に走査するために、新たな変数と補助関数をいくつか導入する。変数 `HeapOrdAreaStart` と `HeapOrdAreaEnd` は、一般オブジェクト領域の先頭と末尾を、変数 `HeapMetaAreaStart` と `HeapMetaAreaEnd` は、メタオブジェクト領域の先頭と末尾を、それぞれ表す。関数 `getMetaObjectSize(b)` は、メタオブジェクト末尾のアドレス `b` を取り、メタオブジェクトのサイズを返す。関数 `metaObjectTop(to, size)` は、メタオブジェクトの末尾を指すポインタ `to` とそのメタオブジェクトのサイズ `size` を取り、そのメタオブジェクトの先頭アドレスを返す。

図 5.3 に、Fusuma アルゴリズムを適用したヒープの例を示す。マークフェーズは従来の Jonkers のスレッド化コンパクションと同様のアルゴリズムを利用する。後向きポインタ更新フェーズは、次の 3 つのステップに分割する (関数 `updateForwardReference`)。まずはじめの `foreach` ループにおいて、ルートに含まれるポインタをスレッド化する (図 5.3 (a)–(b))。次に最初の `while` ループにおいて、従来の後向きポインタ更新フェーズと同様に、一般オブジェクト領域をヒープの先頭から順方向に走査する (図 5.3 (b)–(c))。最後に 2 つ目の `while` ループにおいて、メタオブジェクト領域をヒープの末尾から逆方向に走査する (図 5.3 (c)–(e))。ヒープを走査して発見した生存メタオブジェクトに対する処理は、生存一般オブジェクトに対して行う



(a) M : 完了状態　(b) F : ルートの処理完了

(c) F : メタオブジェクト x の処理開始　(d) F : メタオブジェクト y の処理開始

(e) F : 完了状態　(f) B : メタオブジェクト x の処理開始

(g) B : メタオブジェクト y の処理開始　(h) B : 完了状態

M : マークフェーズ、F : 後向きポインタ更新フェーズ、B : 前向きポインタ更新フェーズ

図 5.3: Fusuma アルゴリズム (マークフェーズは省略)

処理と同様である。まず、移動先のアドレスを用いてスレッド化を解除してそのオブジェクトを指すポインタを更新し、次にオブジェクトの持つポインタをスレッド化する。

前向きポインタ更新フェーズ (関数 `updateBackwardReference`) も同様に、まずはじめに最初の `while` ループにおいて、一般オブジェクト領域を順方向に走査し (図 5.3 (e)–(f))、メタオブジェクト領域を逆方向に走査する (図 5.3 (f)–(h))。各生存一般オブジェクト、生存メタオブジェクトに対して行う処理は、Jonkers のアルゴリズムの前向きポインタ更新フェーズと同じで、移動先のアドレスを用いてスレッド化を解除してそのオブジェクトを指すポインタを更新し、オブジェクトを移動先アドレスへ移動させる。

メタオブジェクト内部のポインタをスレッド化する前に、すべての一般オブジェクトを処理することで、3.3.3 節で説明した問題点を解消する。また、図 5.3 では示していないが、メタオブジェクトが一般オブジェクトへのポインタを持つ場合でも、それらは後向きポインタ更新フェーズでスレッド化され、前向きポインタ更新フェーズで更新されるため、問題はない。

一般オブジェクトはヒープの先頭側へ向かって移動させるが、メタオブジェクトはヒープの末尾側へ向かって移動させることに注意する。移動先アドレスは、それまでに見つけた生存オブジェクトのサイズの総和から計算する。そのため、メタオブジェクトをヒープの末尾側へ移動させるためには、ヒープの末尾から先頭に向かって走査する必要がある。

6　eJS における Fusuma の実装

6.1　境界タグ

Fusuma アルゴリズムはメタオブジェクト領域を、ヒープの末尾から逆方向に走査する。そのためにはメタオブジェクトの末端位置を参照したときに、そのメタオブジェクトのサイズを識別できる必要がある。そこで、メタオブジェクトの末端位置にも、そのメタオブジェクトのサイズ情報を記録する。このサイズ情報を記録する領域を境界タグと呼ぶ。

メタオブジェクト領域をヒープの先頭側から順方向に走査することはない。しかしながら、メタオブジェクトはその先頭にオブジェクトヘッダを引き続き必要とする。なぜなら、メタオブジェクトも一般オブジェクトと同様の方法で型情報を取得できる必要があるからである。また、メタオブジェクトを指すポインタが、そのメタオブジェクトのオブジェクトヘッダを見つけることができないと、ポインタのスレッド化を行うことができないという理由もある。

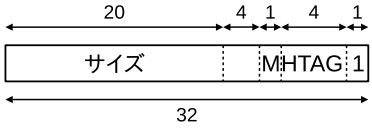
境界タグの素朴な実装として、図 6.1 (a) で示すように、メタオブジェクトの末尾に 1 ワード分の領域を追加し、これを境界タグとする方法が考えられる。この実装では空間オーバーヘッドが生じるため、ヒープサイズが制限される組込みシステムにとって、大きな障害となりうる。この空間オーバーヘッドを解消するために、境界タグの埋め込み [13] と呼ばれる手法の利用を提案する。

図 6.1 (b) に、境界タグの埋め込みを行う場合のメタオブジェクトと境界タグの配置を示す。境界タグの埋め込みを行う場合、メタオブジェクトのオブジェクトヘッダ内のサイズ情報のビット長を半分にする。そうしてできたもう半分の領域を、隣接するメタオブジェクトの境界タグと

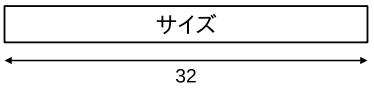
(a) 素朴な実装

(b) 埋め込みを行う実装

図 6.1: 境界タグの二通りの実装方法



(a) 一般オブジェクト、メタオブジェクト共通のオブジェクトヘッダ



(b) メタオブジェクトの境界タグ

図 6.2: 境界タグの素朴な実装を利用する場合のオブジェクトヘッダと境界タグの実装 (M: マークビット).

して利用する。この手法により、ヒープの末尾に配置されるメタオブジェクトの境界タグを記録する 1 ワードを除き、空間オーバーヘッドなしで Fusuma を実装することができる。なお、ヒープの末尾に配置されるメタオブジェクトの境界タグを特別扱いし、メタオブジェクト領域の先頭のメタオブジェクトのオブジェクトヘッダに埋め込むことで、空間オーバーヘッドを完全に除去することも可能である。本論文では実装を単純なままにするために、1 ワードの空間オーバーヘッドは許容することとした。

境界タグの埋め込みの欠点は、メタオブジェクトのオブジェクトヘッダに記録できるサイズ情報が制限される点である。しかしながら、eJSVM の実用上は、一般オブジェクトには影響が無く、メタオブジェクトも通常は十分に小さいため、問題が生じることはほとんど無い。実際、6.2 節で述べるとおり、32 ビット向け実装であっても、オブジェクトヘッダのサイズ情報は 20 ビットある。これを半分にして 10 ビットとしても、最大で 1023 ワードのメタオブジェクトを扱うことができる。eJSVM においては、オブジェクトが 510 を超えるプロパティを持つか、`PropertyMap` に 510 を超える遷移が登録されない限り、このサイズを超えるメタオブジェクトは生成されない。実際、第 7 章で扱うベンチマークプログラムでは、メタオブジェクトの最大サイズは 164 ワードであった。これについては、本論文では取り扱わない。

サイズを記録できないほど巨大なメタオブジェクトの場合には、境界タグやオブジェクトヘッダのサイズ情報に特別な値を記録し、追加のサイズ領域を用意してサイズを記録する、という対策を行うこともできる。

6.2 オブジェクトヘッダ

境界タグの素朴な実装を行う場合のオブジェクトヘッダと境界タグを図 6.2 に示す。また、境界タグの埋め込みを行う場合の、一般オブジェクト、メタオブジェクト、それぞれのオブジェクトヘッダを図 6.3 に示す。

境界タグの埋め込みを行う場合、一般オブジェクトとメタオブジェクトではサイズフィールドのビット幅が異なるため、サイズを取得する際に適切なビットマスクをかける必要がある。Fusuma アルゴリズムでは、オブジェクトからサイズを取得する際、常にそのオブジェクトが一

(a) 一般オブジェクトのオブジェクトヘッダ　　(b) メタオブジェクトのオブジェクトヘッダ

図 6.3: 境界タグの埋め込みを利用する場合のオブジェクトヘッダの実装 (M: マークビット).

般オブジェクトとメタオブジェクトのどちらであるかが静的に決定できる。そのため、サイズ情報の取得のためのビットマスクを、オブジェクトに合わせて動的に決定する必要はない。

6.3 終端ビット

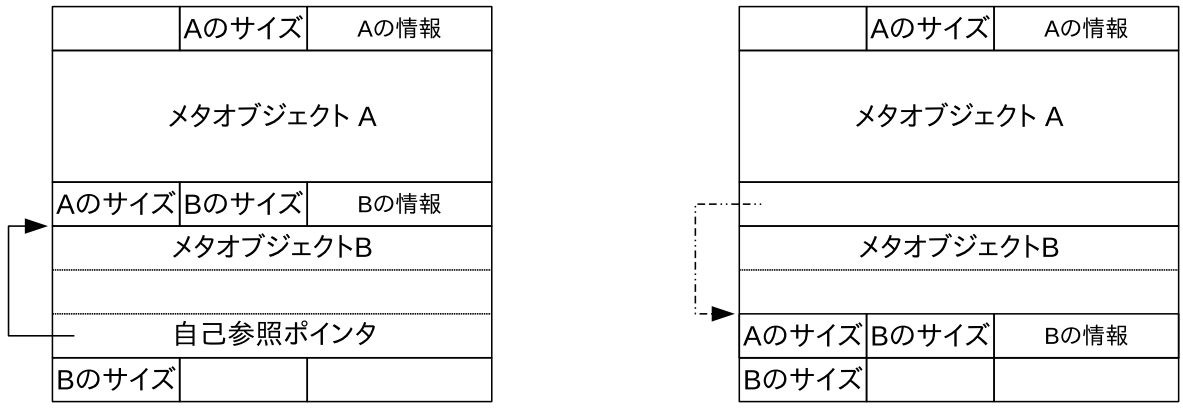
境界タグの埋め込みを行う場合と行わない場合のどちらであっても、すべてのオブジェクトヘッダの LSB には終端ビットとして、常に 1 を設定する。3.3.1 節で述べたとおり、オブジェクトヘッダとポインタフィールドへのアドレスは常に区別できる必要がある。終端ビットはこれを実現する目的で利用する。

オブジェクトを指すポインタはメモリの偶数番地に配置されることを期待する。オブジェクトを指すポインタは、一般オブジェクトやメタオブジェクトが割り当てられる GC 対象のヒープ内に存在する場合と、C 言語の局所変数として存在するポインタなどの GC 対象ヒープ外に存在する場合がある。オブジェクトを 64 ビット境界または 32 ビット境界に合わせて配置することから、前者のポインタは必ず偶数番地に配置される。一方で、後者のポインタは、偶数番地に配置されることは保証されていない。しかしながら、一般的には C コンパイラがポインタを奇数番地に配置することは通常はない。これらのことから、オブジェクトヘッダの終端ビットが 0 であることは、オブジェクトヘッダがスレッド化されたポインタフィールドへのアドレスを格納していることを意味する。

6.4 スレッド化の解除と型情報の復元

2.2 節で述べたとおり、eJSVM では JavaScript の第一級の値を表すデータ型として `JSValue` 型を定義し利用している。`JSValue` 型の値の下位数ビット (32 ビット向け実装では 2 ビット、64 ビット向け実装では 3 ビット) は、型情報を表す PTAG として利用される。PTAG の値は、その `JSValue` 型の値のデータ型ごとに異なる値を取りうる。

もし、ポインタを持つ `JSValue` 型の値をそのままスレッド化すると、オブジェクトヘッダの終端ビットと PTAG が競合してしまう。そのため、`JSValue` 型の値をスレッド化するときは、PTAG を除去し生のポインタへ変形してからスレッド化を行う必要がある。また、スレッド化を解除する際には、そのデータ型ごとに適切な PTAG 値を設定して `JSValue` 型に復元する必要



(a) 自己参照ポインタを含むメタオブジェクト　(b) 自己参照ポインタをスレッド化した状態

図 6.4: スレッド化によってメタオブジェクトが辿れなくなる状態

がある。

eJSVM では、PTAG と HTAG の対応はカスタマイズ時に決定され、また HTAG はデータ型ごとに一意の値が設定される。そのため、HTAG から対応する PTAG の値は一意に定まる。この性質を利用して、スレッド化を解除する際には、はじめに PTAG の値を決定してからスレッド化の解除を行う。PTAG の値を決定するための HTAG の取得には、オブジェクトヘッダの元の値を参照する必要がある。つまり、スレッド化ポインタリストを末尾まで辿る必要がある。従って、eJSVM では、スレッド化の解除を行う際は、スレッド化ポインタリストを 2 回辿る必要がある。

6.5　オブジェクトの自己参照

一般オブジェクトと、境界タグの埋め込みを行う実装におけるメタオブジェクトは、内部に自己参照ポインタが存在しうる場合は、後向きポインタ更新フェーズでサイズ情報を取得する際に、注意が必要となる。オブジェクト内部に自己参照ポインタが存在しなければ、後向きポインタ更新フェーズで、各オブジェクトの処理が完了した時点で、そのオブジェクトのオブジェクトヘッダはスレッド化されていない状態を維持するが、自己参照ポインタが存在する場合はそうでは無くなるためである。もし自己参照ポインタが存在しないことが保証される場合、ヒープを走査する際に、スレッド化ポインタリストを辿らずともサイズ情報を参照でき、実装を単純化することができる。

実際に自己参照ポインタをもつメタオブジェクトが含まれるヒープを図 6.4 に示す。この図では、メタオブジェクト `B` が自己参照ポインタを持っている。Fusuma はメタオブジェクト領域を末尾からスキャンするため、メタオブジェクト `B` の後にメタオブジェクト `A` を処理する。図 6.4 (a) の状態からメタオブジェクト後向きポインタ更新フェーズを開始すると、はじめにメタオブ

ジェクト B の末尾の境界タグを参照し、メタオブジェクト B のオブジェクトヘッダの位置を特定して、オブジェクトヘッダのスレッド化を解除してから、メタオブジェクト B 内部のポインタをスレッド化していく。メタオブジェクト B 内部のポインタのスレッド化が完了すると、図 6.4 (b) の状態となる。次に、メタオブジェクト A の処理へ移るが、そのためにはメタオブジェクト A のサイズ情報が必要になる。もし自己参照ポインタが無ければ、メタオブジェクト B に対する処理を終えた時点でスレッド化は解除されたままとなり、スレッド化ポインタリストを辿らずとも、埋め込まれた境界タグを参照できる。しかし図 6.4 (b) では自己参照ポインタが存在したため、メタオブジェクト B のオブジェクトヘッダはスレッド化され、スレッド化ポインタリストを辿らなければ境界タグを参照できない。この問題への対策として、常にスレッド化ポインタリストを辿って境界タグを参照する方法や、オブジェクトの処理を開始する前にサイズ情報を局所変数へコピーしておく方法が考えられる。

境界タグの埋め込みを行わない実装におけるメタオブジェクトが自己参照ポインタを持つ場合には、特別な対策は必要ない。この実装の場合、境界タグはスレッド化されず、常に直接参照することができるためである。

eJSVM ではオブジェクトが自己参照ポインタを持つことはない。しかしながら、他の処理系で Fusuma アルゴリズムを実装する場合には、上で述べたような対策を施す必要がある可能性がある。

7　性能評価

本章では、Fusuma アルゴリズムを含むいくつかの GC アルゴリズムを eJS に実装し、それらのアルゴリズムの性能を評価、比較する。

7.1　実験環境

本研究では、提案する Fusuma アルゴリズムの性能を評価するために、新たに 5 種類の GC アルゴリズムを eJS に実装した。表 7.1 に、本研究で利用した GC アルゴリズムの一覧を示す。MS を除く 5 種類が、新たに実装したアルゴリズムである。

本研究では eJS が対象とする組込み機器の一例として、STM マイクロコントローラの評価ボードである、NUCLEO-L476RG を利用した。NUCLEO-L476RG の仕様を表 7.2 に示す。また、NUCLEO-L476RG で eJSVM を実行するために、ARM 社が提供する IoT 向け RTOS である、mbed-os[*1] (バージョン mbed-os-6.14.0) を利用した。コンパイラは arm-none-eabi-gcc 7.3.1 を利用した。

ベンチマークプログラムには、JavaScript 向けの標準的なベンチマークである AreWeFastYet (AWFY) ベンチマーク [15] と sunspider ベンチマーク[*2]を利用した。加えて、IoT 機器向けのプログラムの例として dht11 [16] を、大量のメタオブジェクトが生成されごみとなる恣意的なプログラムとして inc-prop [17] を利用した。すべてのベンチマークプログラムについて、プロ

表 7.1: 実験に利用した GC アルゴリズムの一覧

略称	概要
MS	eJS に既に実装されていた、フリーリストを用いるマークスイープ GC アルゴリズム
L2	4.1 節で説明した、単純な拡張を行った Lisp2 コンパクションアルゴリズム
SJ	4.2 節で説明した、単純な拡張を行った Jonkers のスレッド化コンパクションアルゴリズム
QJ	4.3 節で説明した、SJ にキューを用いる改良を行ったアルゴリズム
SF	5.2 節で説明した、単純な Fusuma アルゴリズム
BF	6.1 節で説明した、境界タグの埋め込みを行う Fusuma アルゴリズム

表 7.2: NUCLEO-L476RG の仕様

CPU	STM32L476RG (Arm® Cortex®-M4 @ 80 MHz)
FLASH	1 Mbyte
SRAM	128 Kbyte

[*1] https://os.mbed.com/mbed-os/

[*2] https://webkit.org/perf/sunspider/sunspider.html

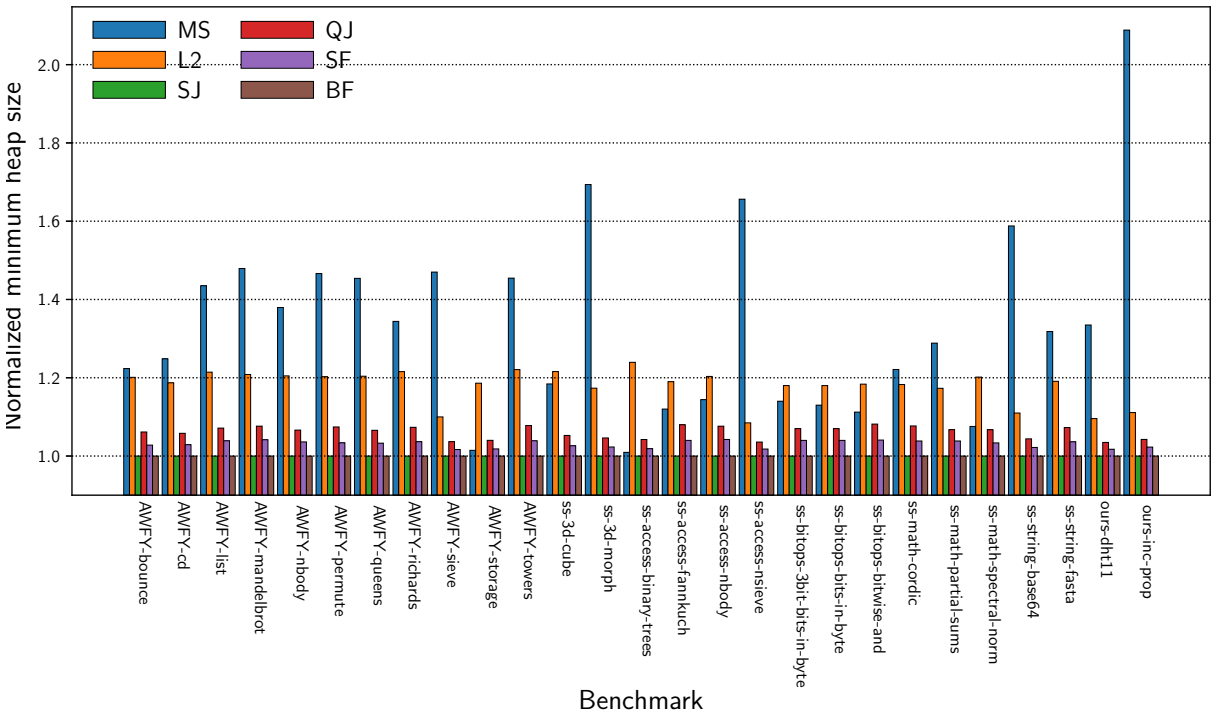


図 7.1: ベンチマーク毎の下限ヒープサイズ

グラムのウォームアップと実行時間の測定を行う、ベンチマークハーネスを適用する修正を加えた。また、NUCLEO-L476RG で実行時間の測定が可能となるよう、ベンチマークプログラムの繰り返し回数を適切に減らす変更も行った。

7.2 空間効率

本節では、表 7.1 で紹介したそれぞれの GC アルゴリズムについて、下限ヒープサイズを比較することで、ヒープの利用効率について評価する。ここで、下限ヒープサイズは、そのベンチマークプログラムを実行するために最低限必要な GC 対象のヒープサイズのこととする。下限ヒープサイズは、NUCLEO-L476RG で割り当て可能な最大のヒープサイズである 64 KiB から測定を開始し、2 分探索的にヒープサイズの縮小と拡大を繰り返し、128 B の粒度で測定した。

図 7.1 に、測定した下限ヒープサイズを、ベンチマーク毎に SJ を 1 として正規化したグラフを示す。SJ は空間オーバーヘッドが無いため、これを 1 としている。このグラフでは、値が 1 に近いほど、すなわち棒が短いほど、空間オーバーヘッドが小さいアルゴリズムである。

BF は、SJ に対し 1 ワード分の空間オーバーヘッドがあるが、128 B 粒度での下限ヒープサイズの測定では、すべてのベンチマークで SJ と同じ下限ヒープサイズを記録した。このことから、BF の 1 ワード分の空間オーバーヘッドは無視できると言える。

SF と QJ はそれぞれ、SJ に対して、メタオブジェクト当たり 1 ワード分、または 2 ワード分

の空間オーバーヘッドがあり、これに伴い下限ヒープサイズも最大で約 1.04 倍、1.08 倍となった。これに対して、オブジェクト当たり 1 ワード分の空間オーバーヘッドが生じる L2 は、下限ヒープサイズは最大で約 1.24 倍となった。SF や QJ の下限ヒープサイズは常に L2 の下限ヒープサイズよりも小さいが、その差はプログラムごとに異なる。例えば、ss-access-binary-trees では L2 の下限ヒープサイズは QJ の約 1.19 倍だが、ss-string-base64 では約 1.06 倍しかない。これらの下限ヒープサイズの違いは、ヒープ中の生存メタオブジェクトの占める割合によるものである。もし、ヒープ中に生存一般オブジェクトが N_L 個、生存メタオブジェクトが N_M 個存在する場合、その時の空間オーバーヘッドは QJ では $2N_M$ ワード、L2 では $N_L + N_M$ ワードである。今回の実験で利用したベンチマークでは存在しなかったが、大量に生存メタオブジェクトが存在し、$N_L < N_M$ となるようなプログラムでは、QJ よりも L2 の方が、下限ヒープサイズが小さくなる。L2 の下限ヒープサイズは SF の下限ヒープサイズより小さくなることはない。SF の空間オーバーヘッドは N_M ワードだからである。

MS の下限ヒープサイズは、SJ との差がベンチマーク毎に大きく異なっている。MS の下限ヒープサイズは、恣意的なプログラムである ours-inc-prop では、SJ の約 2.09 倍である一方で、ss-access-binary-trees では約 1.01 倍であった。特に ss-access-binary-trees では、SF よりも MS の方が、下限ヒープサイズが小さくなっている。MS の下限ヒープサイズは、断片化の程度によって大きく変化する。つまり、ours-inc-prop や ss-3d-morph などの、MS の下限ヒープサイズが SJ に対して大きくなっているプログラムでは、より激しく断片化が生じていたことを意味する。

なお、MS は断片化の影響で、真の下限ヒープサイズよりもヒープサイズが大きい場合に、メモリ割り当てに失敗してプログラムが実行できない場合がある。なぜなら、ヒープサイズが異なることで GC の実行タイミングが変化し、ヒープ中の断片化の程度も変化するためである。本研究での下限ヒープサイズの測定は、2 分探索的に特定のヒープサイズでのみプログラムが実行可能かどうかを判別したため、MS に限り、測定した下限ヒープサイズよりも小さいヒープサイズで動作する可能性がある。

以上をまとめると、次のことが言える。SJ は空間オーバーヘッドがなく、下限ヒープサイズは常に最小で、BF もこれと同等の下限ヒープサイズとなる。QJ や SF は、一定程度 SJ よりも下限ヒープサイズが大きくなるが、その他の空間オーバーヘッドのあるアルゴリズムよりも下限ヒープサイズは小さい。L2 は MS よりも下限ヒープサイズが小さい場合が多いが、断片化の程度が小さいプログラムでは、L2 よりも MS の方が、下限ヒープサイズが小さい場合がある。

7.3 実行効率

本節では、表 7.1 で紹介したそれぞれの GC アルゴリズムについて、ヒープサイズ毎の実行時間や GC 時間を比較することで、アルゴリズムの実行効率について評価する。実行効率の評価対象として、総実行時間、計算時間、総 GC 時間、平均 GC 時間、GC 回数の 5 つを考える。総実

表 7.3: 実行結果の傾向に基づくプログラムの分類

グループ名	概要	該当するプログラム
グループ A	GC 回数がほとんど 0 回	AWFY-permute、ss-bitops-3bit-bits-in-byte、ss-bitops-bit-in-byte、ss-bitops-bitwise-and
グループ B	ヒープサイズの減少に対して GC 回数が段階的に増加する	AWFY-sieve、AWFY-storage、ss-access-nsieve
グループ C	ヒープサイズの減少に対して GC 回数が単調に増加する (おおよそ 100 回未満)	AWFY-bounce、AWFY-list、AWFY-queens、AWFY-richards、AWFY-towers、ss-access-fannkuch
グループ D	ヒープサイズの減少に対して GC 回数が単調に増加する (おおよそ 100 回以上)	AWFY-cd、AWFY-mandelbrot、AWFY-nbody、ss-3d-cube、ss-3d-morph、ss-access-binary-trees、ss-access-nbody、ss-math-cordic、ss-math-partial-sums、ss-math-spectral-norm、ss-string-base64、ss-string-fasta、ours-dht11
グループ E	恣意的なプログラム	ours-inc-prop

行時間は、ベンチマークプログラムの 1 回の実行にかかった全体の時間とする。総 GC 時間は、ベンチマークプログラムの 1 回の実行中の GC を行っていた時間の総和とする。計算時間は、総実行時間から総 GC 時間を引いた時間とする。平均 GC 時間は、1 回の GC 実行時間の平均値とする。GC 回数は、ベンチマークプログラムの 1 回の実行中に生じた GC の回数とする。

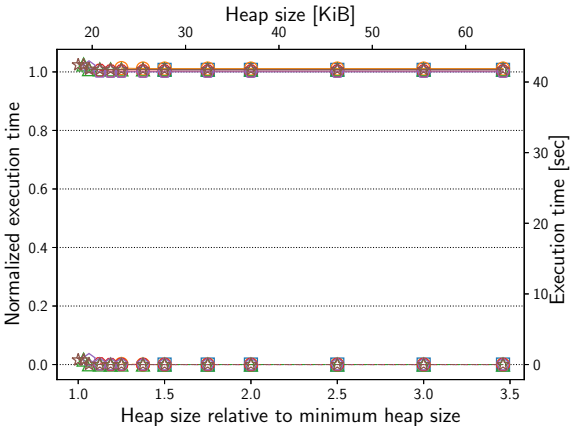
ベンチマークプログラムは、それぞれ 2 回のウォームアップ実行を行った後、続けて 3 回の本番実行を行った。総実行時間と総 GC 時間は、3 回の本番実行それぞれで測定し、最後にその平均を求めた。総 GC 時間は総実行時間から総 GC 時間を引くことで、平均 GC 時間は総 GC 時間を GC 回数で割ることで、それぞれ求めた。

実行効率については、実行結果の傾向をもとにプログラムをグループ A からグループ E の 5 グループに大別し、それぞれの典型的な結果を示したプログラムを代表として紹介する。表 7.3 に、グループとその概要を示す。

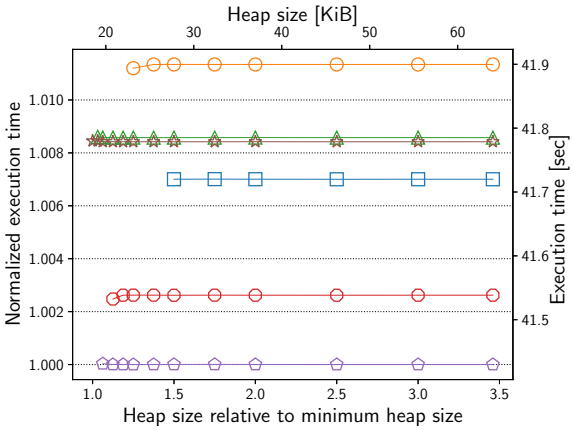
図 7.2 から図 7.6 に、グループ A からグループ E の代表のプログラムの測定結果を示す。測定結果のグラフは、縦軸は時間や GC 回数、横軸はヒープサイズである。縦軸が時間のグラフについては、右側には実時間を、左側には正規化した値を、それぞれ示している。正規化は、各ベンチマーク毎に、ヒープサイズと GC を変化させた際のすべての総実行時間のうち、最も短かったものを 1 としている。横軸は、上側には実際のヒープサイズを、下側には SJ の下限ヒープサイズを 1 とした倍率を、それぞれ示している。

7.3.1 平均 GC 時間

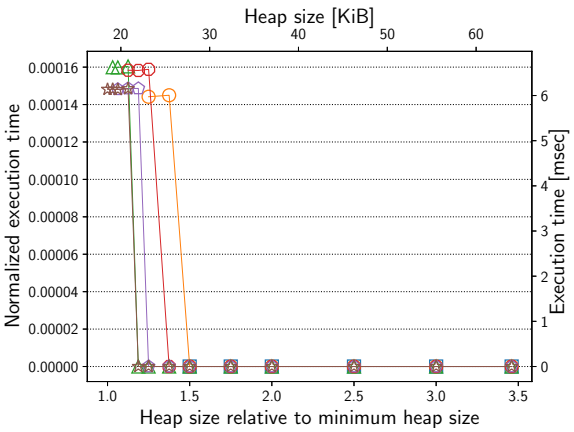
本節では平均 GC 時間の結果とその分析を行う。平均 GC 時間は、十分に GC が実行されないグループ A は、取り扱わない。



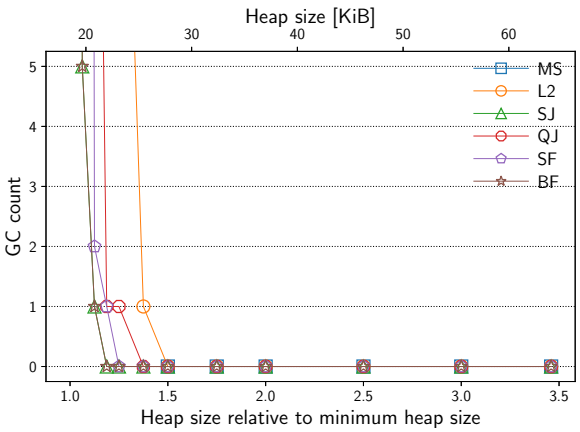
(a) 総実行時間 (実線) と総 GC 時間 (破線)



(b) 計算時間



(c) 平均 GC 時間



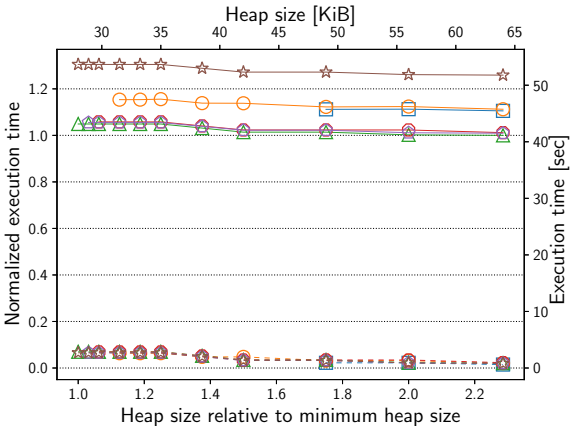
(d) GC 回数

図 7.2: AWFY-permute の実行結果

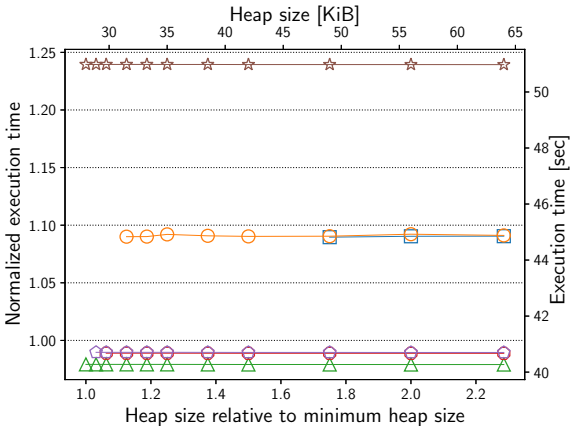
まず、どのグループについても、MS の平均 GC 時間が最も短く、その他は平均 GC 時間が短い方から順に、L2、BF、SF、QJ、SJ という傾向は一致している。特に SF と BF の平均 GC 時間は、ほとんど同じである。

過半数のプログラムでは、平均 GC 時間はヒープサイズに正比例している。しかし、その他のプログラムは、ヒープサイズを大きくすることで、平均 GC 時間が短くなる場合もある。平均 GC 時間は、GC アルゴリズム以外に、ヒープサイズや生存オブジェクト数、オブジェクト内のポインタフィールドの割合などで変化する。ヒープサイズを大きくすることで、平均 GC 時間が短くなるプログラムでは、GC の実行タイミングが変化し、生存オブジェクト数が少ないタイミングで GC が実行されるようになったことで、平均 GC 時間が短くなったのではないかと考えられる。その他に、一部のプログラムで、下限ヒープサイズ付近で平均 GC 時間が増加するプログラムがある。これらのプログラムも、ヒープサイズが減少し、GC の実行頻度が増大し、生存オブジェクト数が多いタイミングでの GC の実行割合が増加したためであると考えられる。

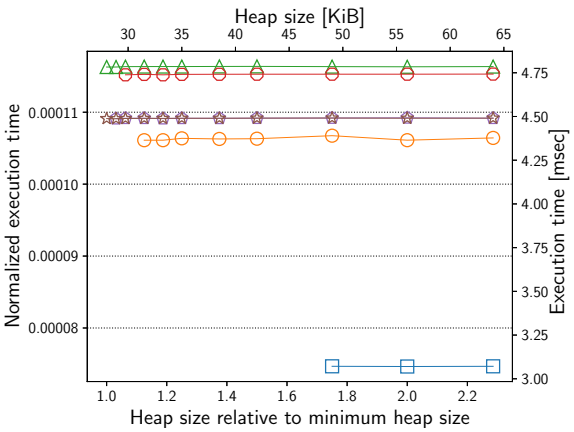
本実験で比較した GC アルゴリズムは、MS を除きすべてマークコンパクト GC であるため、



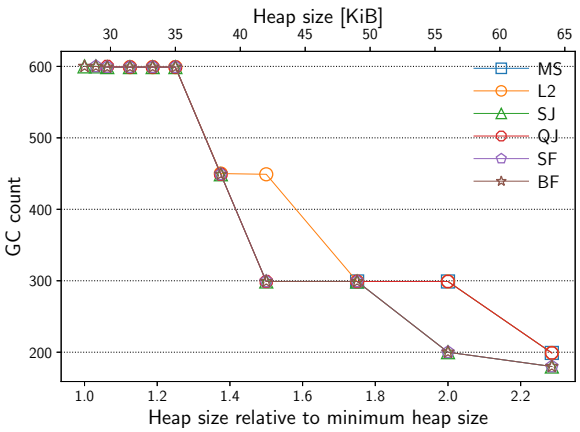
(a) 総実行時間 (実線) と総 GC 時間 (破線)



(b) 計算時間



(c) 平均 GC 時間

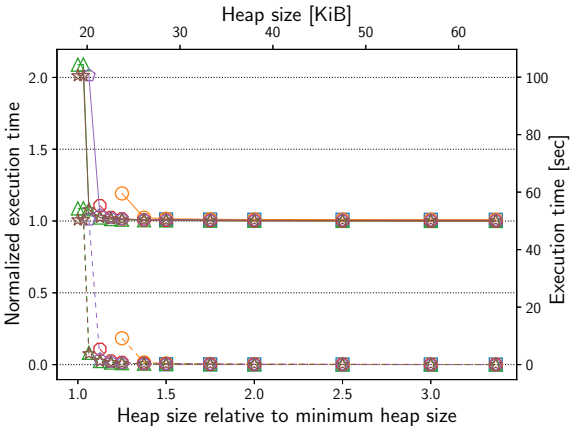


(d) GC 回数

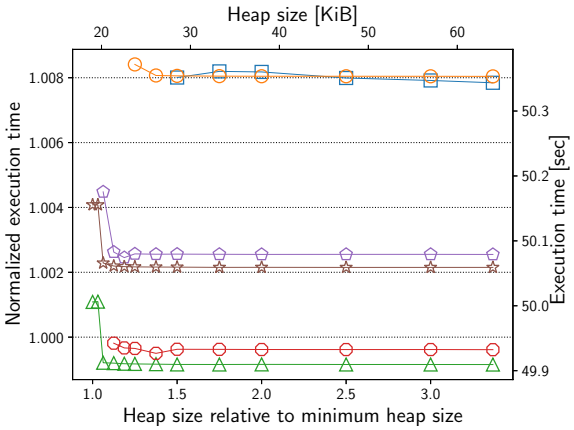
図 7.3: ss-access-nsieve の実行結果

それらはいずれも、平均 GC 時間が MS よりも長かった。特に、Fusuma アルゴリズムを用いる SF や BF の平均 GC 時間は、どちらも多くの場合、MS の約 1.6 倍程度であった。

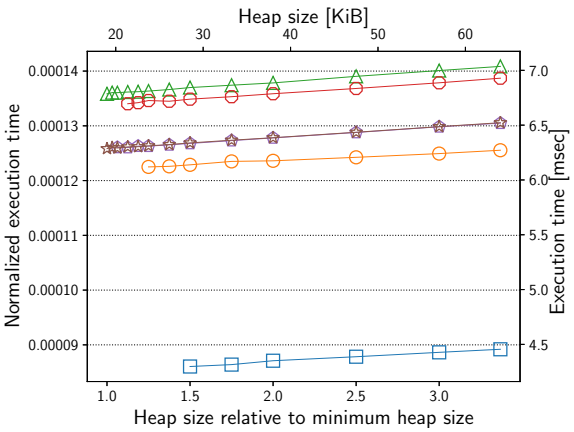
マークコンパクト GC 同士を比較すると、まず、SF と BF を比較することで、境界タグの埋め込みを行っても、平均 GC 時間はほとんど変化しないことがわかる。次に、SF や BF の Fusuma アルゴリズムを用いる場合と、SJ や QJ の Jonkers のアルゴリズムを単純に拡張して用いた場合を比較することで、スキャン回数が少ない Fusuma アルゴリズムを用いる方が、平均 GC 時間が短いとわかる。最後に、Fusuma アルゴリズムを用いる場合と、Lisp2 コンパクションを用いる場合を比較することで、スキャン回数の 1 回少ない Fusuma アルゴリズムより、オブジェクト毎に行う処理が単純な Lisp2 コンパクションの方が、平均 GC 時間が短いことがわかる。



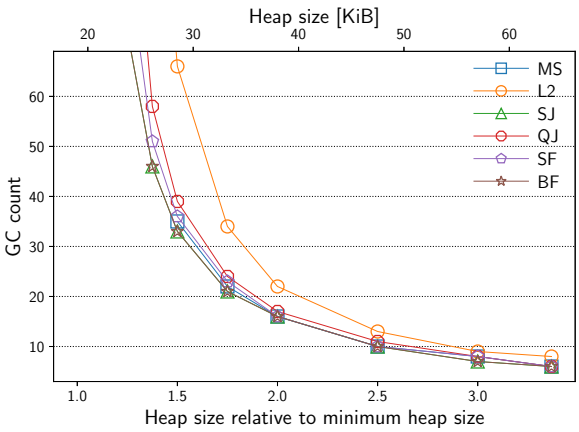
(a) 総実行時間 (実線) と総 GC 時間 (破線)



(b) 計算時間



(c) 平均 GC 時間



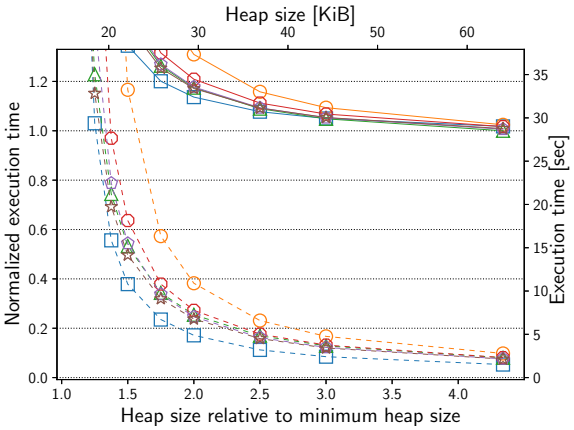
(d) GC 回数

図 7.4: AWFY-queens の実行結果

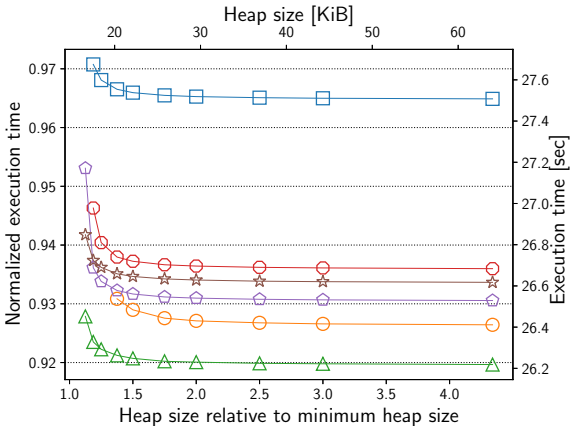
7.3.2 総 GC 時間

本節では総 GC 時間の結果とその分析を行う。総 GC 時間も平均 GC 時間と同様に、十分に GC が実行されないグループ A は、取り扱わない。

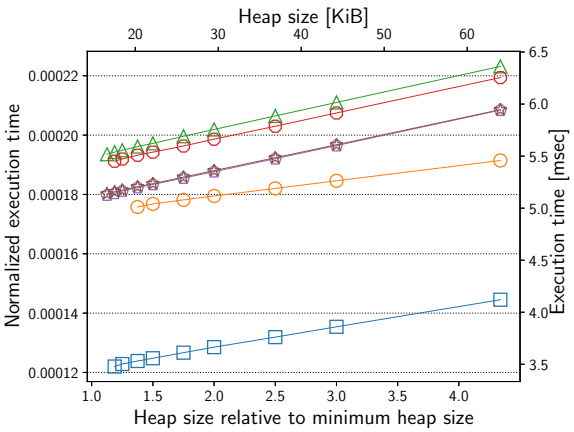
グループ C、グループ D、グループ E のプログラムについて、以下に述べるような傾向がある。まず MS の総 GC 時間が最も短く、その次に総 GC 時間が短いのは、GC 回数の少ない BF や SF で、その次に SF や QJ が並び、最後に L2 の総 GC 時間が最も長い。MS は平均 GC 時間が短く、GC 回数が Jonkers のアルゴリズムを用いてコンパクションを行う場合と大差がないため、総 GC 時間が短くなる場合が多い。ただし、MS の GC 回数は、断片化の程度により増加することがあり、一部のプログラムでは総 GC 時間が長くなる場合も存在する。コンパクションを行う GC アルゴリズムの中で最も平均 GC 時間が短い L2 は、空間オーバーヘッドが大きく GC 回数が他のアルゴリズムよりも多くなるため、ほとんどの場合総 GC 時間が一番長くなる。SJ と BF を比較すると、どちらも空間オーバーヘッドが無く、GC 回数が一致するため、平均



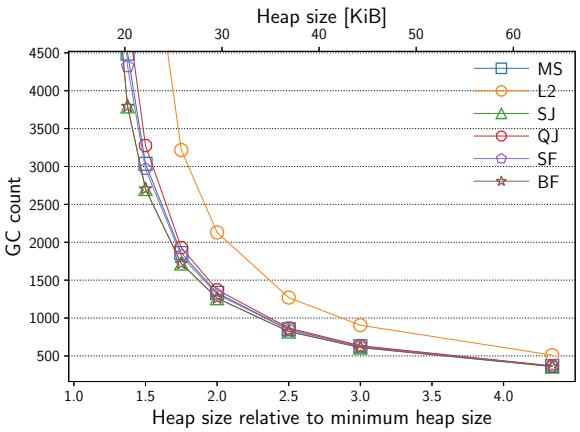
(a) 総実行時間 (実線) と総 GC 時間 (破線)



(b) 計算時間



(c) 平均 GC 時間



(d) GC 回数

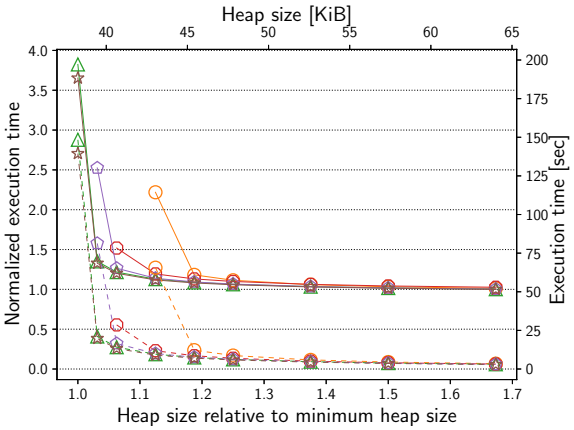
図 7.5: ss-access-nbody の実行結果

GC 時間が短い BF の方が、総 GC 時間も短くなる。SF と BF を比較すると、両者の平均 GC 時間はほとんど同じだが、SF には空間オーバーヘッドがあり、BF よりも GC 回数が多いため、総 GC 時間は BF の方が短くなる。

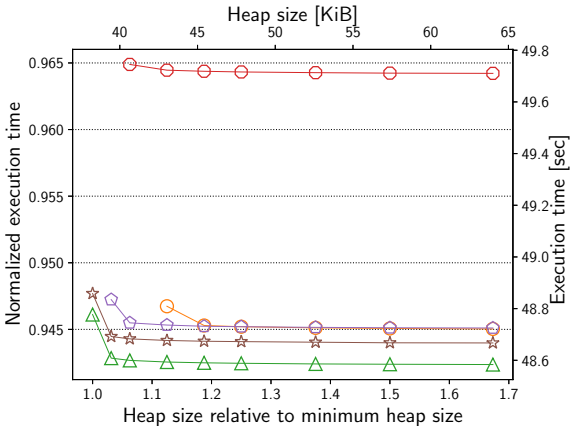
グループ B では、GC 回数がどのコンパクションアルゴリズムでも等しい場合があり、L2 の総 GC 時間が最も短い場合がある。

7.3.3 計算時間

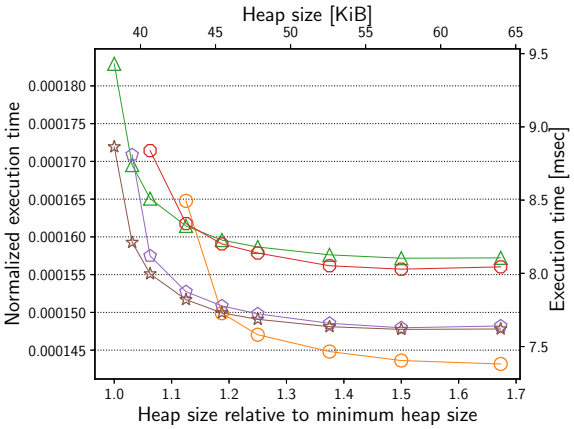
本節では計算時間の結果とその分析を行う。計算時間は、オブジェクトをメモリに割り当てる処理にかかる時間 (アロケーション時間) を含み、それ以外の時間は全く共通の実装を実行するのに要する時間となるため、理想的には GC アルゴリズム間のアロケーション時間の違いが検出できる。しかし実際には、コンパイル時のコンパイラによる最適化や、キャッシュや分岐予測などの高速化機構の影響を受けるため、総実行時間から総 GC 時間を引くだけでは、正確にアロケーション時間の評価はできない。これらの問題を解消し、より厳密な測定を行うことは今後の



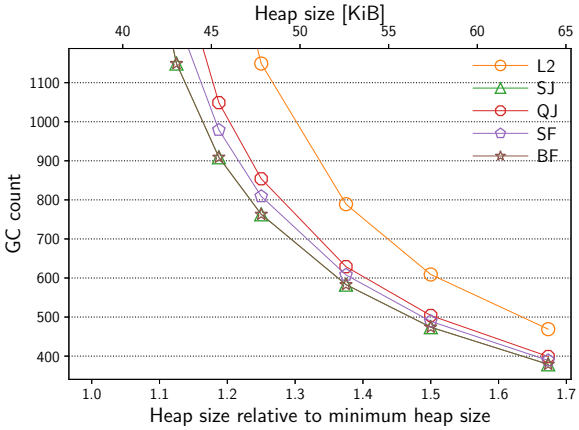
(a) 総実行時間 (実線) と総 GC 時間 (破線)



(b) 計算時間



(c) 平均 GC 時間



(d) GC 回数

図 7.6: ours-inc-prop の実行結果

課題のひとつである。本節では、明らかに GC アルゴリズムの違いによって生じたと思われる計算時間の違いに注目して考察する。

グループによらず、GC アルゴリズムを変化させると、どのプログラムでも計算時間は変化するが、同じ GC アルゴリズムを使う限り、ヒープサイズを変化させても計算時間はほとんど変化しない。GC アルゴリズム間の計算時間の差は、ほとんどのプログラムで、最大でも約 5% 程度に収まっている。また、多くのプログラムで、GC アルゴリズム間の差の大半は約 1% 程度である。特にグループ D の GC 回数が多いプログラムでは、総 GC 時間の占める割合が 10% 以上であり、これと比較すると、計算時間の GC アルゴリズム間の差は十分に小さい。

一部のプログラムでは、ヒープサイズが小さくなると、計算時間が増加していくものがある。これらのプログラムでは、GC 回数も増加しており、空き領域の大きさよりもサイズが大きいオブジェクトのアロケーションが実行されることがある。これにより、アロケーションに失敗してから GC が実行されるといった、ヒープサイズが大きい場合には生じない処理が実行されることがある。また、極端に GC の実行回数が増加すると、GC を行う関数呼び出しのオーバーヘッド

も無視できなくなってくると考えられる。これらの理由から、ヒープサイズが小さい場合に計算時間が増加することがあると考えられる。

グループ A とグループ C は、オブジェクトの割り当てをほとんど行わないため、特にコンパイラによる最適化などの外部からの影響が大きいと見られ、計算時間のグラフから傾向を読み取ることができなかった。また、グループ B も巨大なオブジェクトの割り当てが繰り返され、オブジェクトの割り当てを行う頻度自体が低く、グループ A などと同様に外部からの影響が大きくなったためか、傾向を読み解くことはできなかった。

グループ D では頻繁にアロケーションが生じるため、GC アルゴリズム間の差に特定の傾向が見られる。まず、オブジェクトの割り当ての際に、そのオブジェクトが一般オブジェクトかメタオブジェクトかを区別しない、SJ と L2 の計算時間が短くなっている場合が多い。次に、グループ D ではオブジェクトの割り当てと GC の実行が頻繁に行われるため、割り当て時にフリーリストを辿ることがある MS は、計算時間が長くなっている場合が多い。最後に、BF の計算時間は、ほとんどの場合、SJ や L2 よりが長いが、MS よりは常に短い。このことから、グループ D のようなプログラムでは、コンパクションを行う利点があると言える。

7.3.4 総実行時間

本節では総実行時間の結果とその分析を行う。

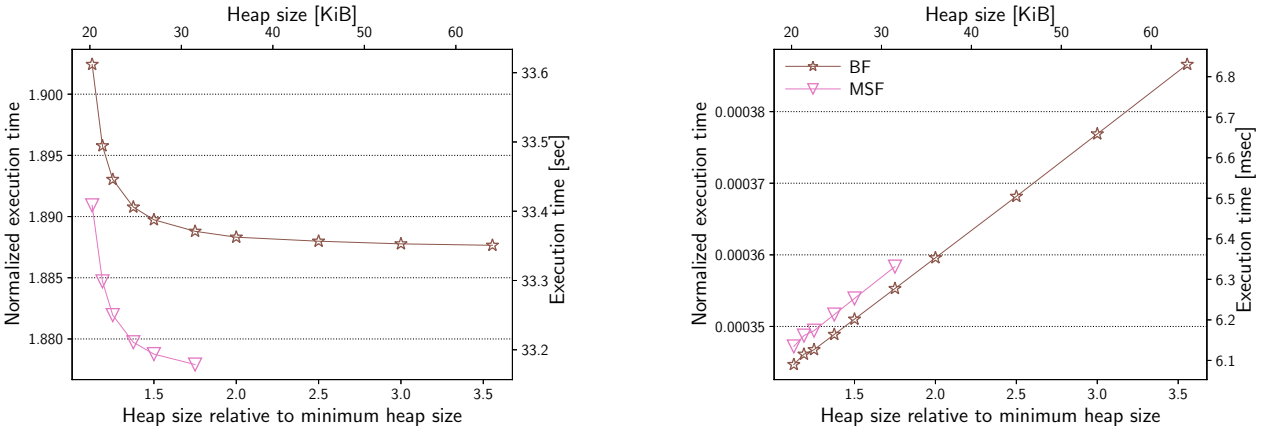
総実行時間は、総 GC 時間と計算時間の和であるため、それぞれの傾向が影響しあい、複雑な傾向を示す。グループ C、グループ D、グループ E に関して、MS でプログラムが実行可能な範囲では、ヒープサイズが大きい間は計算時間が短い SJ が、ヒープサイズが小さくなると総 GC 時間が短い MS が、それぞれ総実行時間を最も短くする。MS ではプログラムを実行できない場合は、総 GC 時間が短い BF の総実行時間が最も短い。

グループ A では、計算時間の差のみが影響し、このグループでは計算時間の差に説明の付く傾向が見られないため、総実行時間も説明のつく傾向は無い。グループ B では、総 GC 時間の占める割合が大きいため、総実行時間は総 GC 時間と同様の傾向を示す。

7.4 逆方向コンパクションの時間オーバーヘッド

本節では、境界タグの埋め込みを行う Fusuma (BF) と、4.4 節で紹介したメタオブジェクト領域を分割して Jonkers のアルゴリズムを適用した場合 (MSF) の実行効率を比較することで、Fusuma のメタオブジェクト領域における逆方向コンパクションと、それに起因する時間オーバーヘッドを確認する。

MSF では、メタオブジェクト領域を常に 32 KiB 確保した。さらに MSF は、一般オブジェクト領域のために N KiB ヒープを確保して、2 つの領域のヒープサイズの合計が $N + 32$ KiB であっても、最大でも N KiB しかヒープ内にオブジェクトを割り当てないように制限した。こ



(a) 計算時間

(b) 平均 GC 時間

図 7.7: AWFY-mandelbrot の実行結果

れにより、MSF で一般オブジェクト領域のために N KiB ヒープを確保した場合と、BF で N KiB ヒープを確保した場合で、GC 実行タイミングが一致する。

図 7.7 に、BF と MSF の比較の代表として、AWFY-mandelbrot の実行結果を示す。グラフは、7.3 節で示したものと同様である。MSF は、ヒープサイズを一般オブジェクト領域のために与えたサイズでプロットしている。MSF はメタオブジェクト領域に 32 KiB ヒープを確保するが、NUCLEO-L476RG では 64 KiB しかヒープを確保できないため、MSF の 32 KiB 以上のプロットは無い。

ほぼすべてのプログラムで、平均 GC 時間は BF の方がわずかに短かった。ただしその差は、総実行時間に対して 0.01% 未満であり、実装上の誤差と言える。BF と MSF は GC 回数が常に一致しているため、総 GC 時間もほとんど同じで、わずかに BF の方が短かった。一方で、計算時間は MSF の方が 0.5% 程度短くなる場合が多かった。BF は、メタオブジェクトの割り当ての度に境界タグの処理を行うが、MSF では必要ない。この違いにより、オブジェクトの割り当てが頻繁に生じるプログラムでは、MSF の方が少しだけ計算時間が短くなる。

以上をまとめると、Fusuma における逆方向のコンパクションは、平均 GC 時間に与える影響はほとんど無い一方で、境界タグの埋め込みを行う場合、オブジェクトの割り当てにかかる時間が増加し、全体の実行時間が 0.5% 程度増加する。

8 関連研究

Java などの静的型付け言語でも、オブジェクトのレイアウトをメタオブジェクトに記録する実装が一般的である。そのような実装では、メタオブジェクトを一般オブジェクトと同時にコンパクションする際にメタオブジェクトを参照できないという、本研究が注目したのと同じ問題が起こる。Jikes RVM [1] のメモリ管理システムである MMTk [3] は、様々な GC アルゴリズムを実装するフレームワークである。MMTk では、オブジェクトを移動させる GC も実装できるように、メタオブジェクトを、マークスイープ GC で管理する専用の領域に割り当てる。これと比較すると、Fusuma アルゴリズムは、一般オブジェクトとメタオブジェクトを同じヒープで管理でき、また、メタオブジェクト領域の断片化を心配する必要がないという利点がある。

コピー GC [5] のように、オブジェクトの移動元と移動先の領域が重ならないコンパクションアルゴリズムであれば、コンパクション中に容易にメタオブジェクトを参照することができる。たとえばコピー GC では、メタオブジェクトを参照するときに、そのメタオブジェクトがコピーされていればフォワーディングポインタをたどるだけでよい。多空間コピー GC [14] や G1GC [8] のように、ヒープを固定長のブロックに分割し、一部のブロックから他のブロックにオブジェクトを移動させることで連続した空き領域を作るコンパクションでも、コピー GC と同様の方法で GC 中にメタオブジェクトを参照できる。V8 JavaScript エンジンはこの方式を使っている。オブジェクトの移動元と移動先の領域が重ならないようにするためには、GC 時には移動対象となるオブジェクトの領域が二重に必要になる。特に二空間コピー GC では、プログラムが使うオブジェクトの総量の 2 倍のヒープが必要になる。一方、本研究で提案する Fusuma アルゴリズムは、オブジェクトをヒープの端に寄せるスライディングコンパクションであり、このような空間オーバーヘッドはない。

GC 中にオブジェクトを参照する必要があるという制約は、GC とミューテータが並行して動作する並行 GC とも共通する。コンパクションを行う並行 GC [8, 20, 2] でも、ほとんどのアルゴリズムはオブジェクトの移動元と移動先の領域が重ならないようにしている。

スライディングコンパクションには、本研究で扱った Lisp2 コンパクションやスレッド化コンパクションの他に、Break-Table コンパクション [10] がある。このコンパクションアルゴリズムも、Lisp2 コンパクションやスレッド化コンパクションと同様に、簡単にはメタオブジェクトを一般オブジェクトと一緒にコンパクションすることはできない。

Break-Table コンパクション [10] は、オブジェクトの移動先を Break-Table と呼ばれるポインタのテーブルで管理してコンパクションを行う。Break-Table には、生存オブジェクトの先頭アドレスと、そのオブジェクトの移動量を記録する。Break-Table は、複数の断片に分割して、生存オブジェクトの隙間に作られ、オブジェクトの移動に伴って 1 箇所に集められる。そのため、空間オーバーヘッドはない。しかし、集めた断片をソートするため、オブジェクト数

N に対して時間計算量が $O(N \log N)$ になる。また、全てのオブジェクトを移動し終わるまで、Break-Table はソートされておらず、ポインタが指すオブジェクトの移動先は分からないので、メタオブジェクトをコンパクションの対象にすると、オブジェクトのレイアウトが参照できなくなる。

Mark-Sweep-Compact [18] は、GC の際に基本的にマークスイープ GC を行い、時折コンパクション GC を行うことで、マークスイープ GC による高速な GC を行いながら、断片化による問題を解消する。この手法では、適切な基準を設けて、GC の際にマークスイープ GC とコンパクションのどちらを行うか切り替えることで、2 つの GC のうち、より高速に動作する方と互角の性能を得ることができる。Mark-Sweep-Compact の戦略は、Fusuma アルゴリズムの戦略とは直交するもので、両者を比較することや、両者を併用することが可能である。これらは今後の課題として挙げられる。

9 おわりに

本研究では Hidden クラスを利用して一般オブジェクトのレイアウト情報を記録している場合に、Hidden クラスを構成するメタオブジェクトと一般オブジェクトを同じメモリ上に配置しながらスレッド化コンパクションを行うための、Fusuma アルゴリズムを提案した。Fusuma アルゴリズムと境界タグの埋め込みを併用することで、空間オーバーヘッドの無い実装とすることが可能であることも示した。

また、提案したアルゴリズムを eJSVM に実際に実装し、他の GC アルゴリズムとの比較を行った。その結果、使用した 27 のすべてのベンチマークプログラムについて、空間効率が最も良いことを実験的に示した。また、実行効率については、オブジェクトのアロケーションがある程度頻繁に行われ、GC 回数が多いプログラムについて、ヒープサイズが小さい場合には、既存の GC アルゴリズムよりも総実行時間を最も短くできることを示した。加えて、Fusuma アルゴリズムに特有の、逆方向コンパクションによる時間オーバーヘッドを調査し、総実行時間の増加が約 0.5% 程度に収まることを示した。

謝辞

本研究を行うにあたり、終始様々なご指導を賜りました指導教員である岩崎英哉教授に深く感謝いたします。また、研究にお力添えいただきました、東京大学の鵜川始陽准教授に深く感謝いたします。本研究に関して様々なご意見をいただきました、岩崎研究室の皆様にも御礼申し上げます。

参考文献

[1] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–418, 2005.

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2003)*, pages 81–92. ACM, 2003.

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 137–146. IEEE Computer Society, 2004.

[4] Craig Chambers, David M. Ungar, and Elgin Lee. An efficient implementation of SELF – a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA 1989)*, pages 49–70. ACM, 1989.

[5] Chris J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[6] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento mori: dynamic allocation-site-based optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management (ISMM 2015)*, pages 105–117. ACM, 2015.

[7] Ulan Degenbaev, Michael Lippautz, and Hannes Payer. Concurrent marking of shape-changing objects. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM 2019)*, pages 89–102. ACM, 2019.

[8] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM 2004)*, pages 37–48. ACM, 2004.

[9] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1984)*, pages 297–302. ACM, 1984.

[10] Bruce K. Haddon and William M. Waite. A compaction procedure for variable-length

storage elements. *Comput. J.*, 10(2):162–165, 1967.

[11] Urs Hölzle, Craig Chambers, and David M. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 1991.

[12] H. B. M. Jonkers. A fast garbage compaction algorithm. *Inf. Process. Lett.*, 9(1):26–30, 1979.

[13] Donald Ervin Knuth. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

[14] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques*, pages 253–263. ACM, 1987.

[15] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 120–131. ACM, 2016.

[16] Hiro Onozawa, Hideya Iwasaki, and Tomoharu Ugawa. Customizing JavaScript virtual machines for specific applications and execution environments. *Computer Software*, 38(3):23–40, 2021.

[17] Hiro Onozawa, Tomoharu Ugawa, and Hideya Iwasaki. Fusuma: double-ended threaded compaction. In Zhenlin Wang and Tobias Wrigstad, editors, *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, pages 94–106. ACM, 2021.

[18] Tony Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[19] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. ejstk: Building javascript virtual machines with customized datatypes for embedded systems. *J. Comput. Lang.*, 51:261–279, 2019.

[20] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. Transactional sapphire: Lessons in high-performance, on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.*, 40(4):15:1–15:56, 2018.