他言語関数呼び出しにおける遠慮のかたまり問題の解決に向けたオブジェクトグラフの解析

山崎 徹郎　千葉 滋

新規に開発されるプログラミング言語はコード資源を持たないため，別の言語で実装されたライブラリを呼び出す他言語関数呼び出し (Foreign Function Call) は重要な機能である．他言語関数呼び出しではその引数や返り値は相手側のデータであるため，言語処理系の境界をまたぐリモート参照がしばしば作られる．呼び出し元である新言語と呼び出し先である既存言語がそれぞれ異なる方法でガベージコレクションを行う場合，このリモート参照の存在が問題となる．例えば新言語と既存言語の両方の領域をまたぐ循環参照は回収が困難である．この種のメモリリークはコレクタ同士が回収を譲り合った結果生じるように見えるため，我々は遠慮のかたまりと呼んでいる．本発表ではこの遠慮のかたまり問題への対策を論じる．アプリケーションとして新言語からの他言語関数呼び出しを想定するため，既存言語は処理系を改造しないものとする．

Since a newly implemented programming language has no code resources, foreign function call which enables programmars to call libraries implemented in other language is an important feature. Foreign function calls often create remote references which reference to objects in other language to pass arguments or receive results those are data in the other language. Remote references are problematic if both caller and callee languages supports garbage collection. For example, it is hard to collect cyclic references that go through both languages. We call this kind of memory leaks “the last piece of cake” since they seems to occur as a result of hesitance between collectors. In this paper, we present an approach to the last piece of cake problem. Since we suppose foreign function call as the application, we avoid customizing an existing language system.

1　はじめに

近年では多様なプログラミング言語が利用可能である．そのため，しばしば複数のプログラミング言語が連携動作するアプリケーションが開発される．例えば web アプリケーションではサーバーサイドとクライアントサイドをそれぞれ異なるプログラミング言語で開発することは珍しくない．また機械学習では広く Python が使用されるが，その主要な計算はしばしば C 言語で実装される．プログラミング言語によってはグラフィックスライブラリを標準としてサポートしていないため，Ruby や Haskell などの言語ではグラフィカルなプログラムを作成するために C バインディングを利用する．

ある言語から他の言語で定義された関数を呼び出す仕組みを Foreign Function Interface (FFI) という．従来では C 言語との間の FFI さえあればよかったが，近年では Python や Javascript のようなスクリプト言語が人気であるため，これらの言語との間の FFI が注目されている．また近年では多くの言語がガベージコレクションによる自動メモリ管理を行う．Python も Javascript もガベージコレクションによって自動的にメモリを管理する．

ガベージコレクションを備えた言語の間の FFI では，ガベージコレクタの間の連携が課題の一つとして挙げられる．ガベージコレクタはふつう言語ごとに開発され，自言語のオブジェクトだけを管理する．しか

---

* A method for object graph analysis toward solving the last piece of cake problem in foreign function call.

Tetsuro Yamazaki, Shigeru Chiba, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

し FFI のように複数の言語処理系が連携動作する場合，特に他言語のオブジェクトを直接参照するリモート参照が多用される場合，自言語内の情報だけではオブジェクトの生死を正確に判定することができない．到達不可能であるオブジェクトをしかし回収しなければメモリリークとなり，到達可能であるオブジェクトを誤って回収してしまうとメモリエラーの原因となり，最悪の場合処理系がクラッシュすることもある．

FFI のためのガベージコレクションでは，ガベージコレクタの改造もまた課題の一つである．FFI はすでに存在する言語で定義された関数を呼び出す仕組みであるため，その言語のガベージコレクタもまたすでに存在している．ガベージコレクタは複雑なプログラムであり，改造するにしても一から作り直すにしても多大な労力が必要である．なるべく既存のコレクタは改造せずに再利用できることが望ましい．

我々は FFI のためのガベージコレクションアルゴリズムとして，ガベージコレクタの改造を最小限に生死判定の精度を向上させる手法を提案する．我々のアイデアは一方の言語のオブジェクト同士の参照関係を，そのままもう一方の言語のオブジェクト同士の参照関係として再現するというものである．ガベージコレクタ間の連携の失敗はどのコレクタも部分的な情報しか持たないために正確な生死判定ができないことが原因であるため，相手側のオブジェクトの参照関係として全ての情報を再現したなら相手側のコレクタは通常通りのトレースだけで改造なしに正確な生死判定を行うことができる．このアイデアをベースに，コピーする参照関係を必要最小限に圧縮することで性能を向上させる．我々のアルゴリズムはヒープの解析に bloom filter [4] を使用する．Bloom filter は偽陽性があるため得られる情報には誤りが含まれ，100% 正確な生死判定はできない．この誤りの影響については 3.3 節で詳しく触れる．

我々のアルゴリズムは 2 ノードの場合を主に想定するが，3 ノード以上のケースにも適用可能である．我々の手法では無改造で済むガベージコレクタはただ 1 つだけであり，それ以外のコレクタは全て改造する必要がある．3 ノード以上のケースについては 3.2 節で詳しく触れる．

ソースコード 1　Ruby から Javascript を呼び出す例

```
1 require 'jscall'
2 document = JSCall.eval('document')
3 button   = document.getElementById('
      button')
4 message    = document.getElementById(
      'message')
5 button.onclick = proc do
6   message.innerText = 'click'
7 end
```

以下，2 章で FFI 環境下でガベージコレクタ同士の連携が失敗した場合なにが起きるのかを示し，3 章でガベージコレクタ同士で情報共有する方法を提案する．4 章では関連研究を示し，5 章でまとめる．

2　Foreign Function Interface と分散ガベージコレクション

あるプログラムから別の言語で記述された関数を呼び出す仕組みを Foregin Function Interface (以下 FFI) という．新しく開発された言語はライブラリ資源を持たないため，FFI のように他言語のライブラリにアクセスする仕組みは重要である．従来の FFI は Java Native Interface [9] や libffi [2]，Python ctypes [6] のように C 言語のライブラリを呼び出すことができれば十分であった．しかし近年では C 言語だけでなく PyCall [7] や PyV8 [1] のように Python や Javascript のライブラリを呼び出す FFI も注目を集めている．

例えば我々が開発している Ruby から Javascript を呼び出す FFI を使用するとソースコード 1 のようなプログラムを作成することができる．このプログラムはウェブページのボタンがクリックされたとき，メッセージを “click” に設定する．2 行目では `JSCall.eval` 関数によって，文字列 `'document'` を Javascript の式として評価する．`'document'` の評価結果は Javascript の `HTMLDocument` オブジェクトであるため，Ruby から Javascript のオブジェクトを参照するリモート参照が作成される．`document` のようにリモート参照によって参照されるオブジェクトを公開オブジェクトと呼ぶ．3 行目では `document` のメソッドを呼び出す．このメソッド呼び出しは参照

先である HTMLDocument オブジェクトへと転送され，Javascript のメソッド呼び出しとして計算される．5 行目では Javascript のオブジェクトである button に onclick コールバックとして Proc オブジェクトを登録する．Proc は Ruby のオブジェクトであるため，実際に登録されるコールバック関数は Javascript から Ruby の Proc オブジェクトを指すリモート参照である．

Ruby から Javascript を呼び出す場合のように，FFI によって連携動作する 2 つの言語がそれぞれ異なる方法でガベージコレクションを行う場合，コレクタ同士の連携が一つの課題である．ガベージコレクタはふつう言語処理系の一部として実装されるため，自言語が管理するメモリ領域の情報しか持たない．しかし FFI のように複数の言語が連携動作する場合，特に言語の境界をまたぐリモート参照が多用される場合，自領域の部分的な情報だけではオブジェクトの生死を正確に判定することができない．オブジェクトが到達不可能であるにもかかわらず回収できないことによるメモリリークや，オブジェクトが到達可能であるにもかかわらず誤って回収してしまうことによるメモリエラーが起きる可能性がある．どのようにしてメモリリークが起きるのかについては 2 節に詳しい．

FFI のためのガベージコレクションではガベージコレクタの改造もまた課題の一つである．FFI はすでに存在している言語で実装された関数を呼び出すため，その言語のガベージコレクタもまたすでに存在している．しかしガベージコレクタは非常に複雑なプログラムであるため，改造するにしても一から作り直すにしても多大な労力を必要とする．例えば ruby のガベージコレクタは 12,000 行からなる巨大なプログラムである．ガベージコレクタの誤作動によるメモリエラーは原因の特定が難しく，また再現性も低く，デバッグが困難なバグの 1 つである．また改造が誤りを含む可能性もあるため，既存ライブラリの動作が保証できなくなる問題もある．FFI のためのガベージコレクションではコレクタの改造を最小限に，可能であれば一方のコレクタを無改造に済ませることが望ましい．

分散ガベージコレクション [10] では FFI のガベージコレクションのように，ガベージコレクタ同士の連携が課題の一つである．分散環境ではマシンごとにメモリ領域も計算資源も分割されているため，マシンごとのローカルコレクタは自領域の情報しか持たず，やはりオブジェクトの生死判定を正確に行うことはできない．分散ガベージコレクションではローカルコレクがが収集した情報をグローバルコレクタが分配することで正確な生死判定を可能とする．我々の知る限りでは，既存のコレクタをローカルコレクタとして無改造にに再利用できる分散ガベージコレクションアルゴリズムは知られていない．

FFI の素朴な実装とメモリリーク

複数のガベージコレクタが連携動作するアプリケーションでは，コレクタ同士の連携に失敗するとメモリリークやメモリエラーが起きることがある．本節ではコレクタ同士の連携の失敗によって引き起こされるメモリリークの一例を示す．まず素朴な FFI として eval 関数とリモート参照の 2 つのインターフェースを持つ FFI の設計を示す．そして素朴な FFI ではリモート参照がガベージコレクタによってどう管理されるかを示し，その後メモリリークの例を示す．

FFI の素朴な実装方法として全ての公開オブジェクトを含むテーブルを用意する方法を考える．このテーブルを公開テーブルと呼ぶ．公開テーブルは公開 ID から公開オブジェクトへのマッピングである．オブジェクトが公開テーブルに登録されるたびに一意の公開 ID を生成する．両方の言語が公開テーブルを用意する．eval 関数は引数として式を表す文字列を受け取り，相手側言語の式として評価しその結果を返す．式を評価する間，自言語の実行はブロックされる．評価結果が文字列や数値のようなプリミティブな値だった場合は自言語の対応する値に変換し，そうでなかった場合はリモート参照を作成し返却する．リモート参照はプロキシオブジェクトの形で実現する．相手側言語のオブジェクトを直接指すリモート参照を本当に作成してしまうと問題があるため，代わりに相手側オブジェクトのようにふるまうプロキシオブジェクトを作成し，リモート参照は直接相手側言語のオブジェクトを参照せず，代わりにプロキシオブジェクト

を参照する．プロキシオブジェクトはフィールドやメソッドが参照された場合，内部的に eval 関数を呼び出すことで参照先オブジェクトのフィールド参照やメソッド呼び出しを計算し，その結果を返す．プロキシオブジェクトは参照先オブジェクトを特定するため，参照先オブジェクトの公開 ID を記憶する．公開テーブルに登録されていないオブジェクトは参照できない．プロキシオブジェクトのメソッドに渡された引数は，プリミティブな値なら対応する値に変換し，そうでなければリモート参照を作成し，参照先オブジェクトのメソッドに渡す．

この素朴な FFI において，ガベージコレクタがリモート参照をどう扱うかを示す．公開テーブルはプログラムと同じ寿命を持つグローバルオブジェクトである．公開テーブルから公開オブジェクトへの参照は強参照であるため，公開オブジェクトは基本的には回収されない．公開オブジェクトはいつリモート参照を経由して参照されるかわからないため，公開オブジェクトが回収されない実装は保守的であり，特別変な実装ではない．一方プロキシオブジェクトは，通常のオブジェクトと同様にガベージコレクションによって回収される．プロキシオブジェクトにファイナライザを設定することで，回収されたプロキシオブジェクトの参照先オブジェクトの公開を停止することができる．リモート参照が回収された公開オブジェクトはもはや公開オブジェクトではないため，公開テーブルから削除することでガベージコレクションによる回収が可能となる．この素朴な FFI ライブラリのガベージコレクタは公開オブジェクトの回収を，リモート参照が回収されるまで保留する．

上で示した公開テーブルによる FFI の実装では，公開オブジェクトのガベージコレクションは参照カウント [5] の一種である．リモート参照の数をカウントし，その数が 0 であった場合は公開オブジェクトの公開を停止し，回収可能にする．そのため，参照カウントと同様に循環参照を回収することができない．この種類のメモリリークはコレクタ同士が回収を譲り合った結果に見えるため，遠慮のかたまりと呼ぶことにする．

3 提案手法

我々は参照関係情報を共有することでガベージコレクタの改造を最小限にリモート参照を含む循環参照 (遠慮のかたまり) を回収可能にする手法を提案する．提案するアルゴリズムは遠慮のかたまりを回収することに特化した，バックアップ用のガベージコレクションである．領域ごとの通常のガベージコレクションは別途行う．以下では 2 章で示した方法で実装された素朴な FFI を仮定する．

我々の基本的なアイデアはメモリ領域 A に存在するオブジェクトの参照関係を全て，もう一方のメモリ領域 B にコピーすることである．そうすることでメモリ領域 B を管理するコレクタは全ての参照関係を辿ることが可能となり，正確なオブジェクトの生死判定が可能となる．このとき，メモリ領域 B を管理するコレクタの改造は不要である．メモリ領域 B の到達不能オブジェクトが回収されたならば，遠慮のかたまりはその一部が回収され循環ではなくなるため，残りも通常のガベージコレクションによる回収が可能となる．

本当に全てのオブジェクトをコピーしてしまうとムダなので，必要最小限だけをコピーする．正確な生死判定のために必要な情報は領域 A・B 両方を含んだオブジェクトの参照関係である．そのため，メモリ領域 B を管理するコレクタに不足している情報はメモリ領域 A を経由することでどのプロキシオブジェクトからどの公開オブジェクトを参照可能であるか，その参照関係である．これはメモリ領域 A を管理するもう一つのコレクタから見ると，メモリ領域 A の中で，メモリ領域 B を経由せずに，どの公開オブジェクトからどのプロキシオブジェクトが参照可能であるかの参照関係である．Tracing Garbage Collection のマークフェイズのような計算によって調べることができる．

我々の提案するガベージコレクションはストップ・ザ・ワールド方式である．ガベージコレクションを始める前に両方の言語処理系の計算を停止し，ガベージコレクションの全てのフェイズが完了した後に計算を再開する．これはユーザースレッドがガベージコ

ソースコード 2　参照関係情報の収集

```
1  stack := ϕ
2  for public do
3    mark(public) <- { public }
4    push(stack, public)
5  end
6  while stack != ϕ do
7    c <- pop(stack)
8    for n ← pointers(cur)
9      if mark(c) \not \subseteq mark(n)
           then
10       mark(n) <- mark(c) \cup mark(n)
11       push(stack, n)
12     end
13   end
14 end
```

レクションの不変条件を破ることを防ぐためである．ユーザースレッドと並列実行可能なガベージコレクションも研究されているが，これらはライトバリアやリードバリアが必要である．言語処理系はしばしばライトバリア・リードバリアを実装していないため，コレクタの改造が必要となる．ワールドストップもまた，コレクタの改造を必要とする機能である．しかし Ruby，Python のようにグローバル・インタプリタ・ロックを持つ言語や，Javascript のようにシングルスレッドで計算する言語であれば改造なしにワールドストップすることができる．

3.1　アルゴリズム

我々の手法は参照関係情報の取得と，参照関係情報の反映の 2 つのステップがある．参照関係情報を取得するために tracing garbage collection のマークフェイズのような計算を行う．通常マークフェイズでは，オブジェクトに生死を表す 1 ビットのマークを記憶させ，ルートセットから深さ (または幅) 優先探索を行うことで生死情報を伝搬させる．我々の手法では，それぞれのオブジェクトがどの公開オブジェクトから参照可能であるかを調べる．そのため，それぞれのオブジェクトが記憶するマークはそのオブジェクトを参照可能である公開オブジェクトの集合である．マークが真偽の 2 値ではなくなったため，探索も単純な深さ (または幅) 優先探索ではうまくいかない．同じマークが少しずつ繰り返し更新される可能性があるためである．我々はマークが更新される限り探索を続ける単純なアルゴリズムを採用した．アルゴリズムの外観をソースコード 2 に示す．

各オブジェクトが記憶するマーク (集合) のデータ表現として bloom filter [4] を採用する．集合はやや大きなデータ構造であり，オブジェクトの数だけ用意するとなるとかなりの量のメモリが必要となる．Bloom filter は任意個の要素からなる集合を固定長のビット数で表現できるデータ構造であるため，オブジェクトごとに 32 ビットか 64 ビット程度の領域を追加で用意すればそれで済む．Bloom filter の欠点は偽陽性があることである．Bloom filter の包含判定は，本当はその要素が含まれていなかったとしても包含すると判定する場合がある．これは我々の手法では，本当は到達できないオブジェクト同士が誤って到達可能であると判定されるためにメモリリークが起きる可能性があることを意味する．偽陽性については 3.3 で議論する．

収集した参照関係情報をもう一方の言語に送信し，オブジェクト同士の参照関係として再現する．参照関係はメモリ領域 A を経由することでどのプロキシオブジェクトからどの公開オブジェクトが参照可能であるかの情報である．プロキシオブジェクトの隠しフィールドとして配列のような可変長のデータを用意し，領域 A を経由することで参照可能な公開オブジェクトへの参照を格納する．反映が完了した後であれば正確なオブジェクトの生死判定が可能であるため，公開テーブルから各公開オブジェクトへの参照を弱参照に変更し，通常通りのローカルガベージコレクションを行う．ローカルガベージコレクションによってプロキシオブジェクトが回収された場合ファイナライザが起動するため，公開テーブルの対応するエントリは削除される．ただし，領域 A でルートから到達可能である公開オブジェクトは公開テーブルからの参照を強参照のままにする．

通常のガベージコレクションが完了した後は公開テーブルから公開オブジェクトへの参照を強参照に戻し，ワールドストップを解除する．遠慮のかたまりが存在していたならば，そのメモリ領域 B に属する部分は回収され循環ではなくなっているので，残りは通常のガベージコレクションで回収することができる．

ユーザースレッドによってメモリ領域 A のオブジェクトが変更された場合，収集した参照情報と実際のメモリ領域 A との間の整合性が失われてしまう．整合性が失われると参照可能なオブジェクトが誤って回収されてしまう恐れがあるため，公開オブジェクトへの参照を強参照に戻すまで両言語のワールドストップは解除しない．

3.2 ３ノード以上の環境への適用

3.1 節で示したアルゴリズムは 2 言語間の FFI に特化していた．我々の手法は 3 言語以上の環境にも適用可能である．本節ではその方法のスケッチを示す．

３つの言語をそれぞれ A, B, C とする．まず A, B, C 全ての言語をワールドストップする．次に言語 A の参照関係情報を収集し，収集した情報を言語 B と言語 C との両方に送信する．言語 B には言語 A を経由することで言語 B のどのプロキシからどの公開オブジェクトに到達可能であるかを，言語 C には言語 A を経由することで言語 C のどのプロキシからどの公開オブジェクトに到達可能であるかをそれぞれ送信する．

言語 B，言語 C それぞれで言語 A から受信した参照関係情報を反映する．この時点で言語 A と 言語 B をまたぐ循環参照と言語 A と言語 C をまたぐ循環参照は回収できるが，言語 B と言語 C をまたぐ循環参照，および言語 A，B，C の全てをまたぐ循環参照は回収できない．

次に言語 B の参照関係情報を収集し，収集した情報を言語 C に送信し，反映する．この時点で言語 C のコレクタは全ての参照関係を調べることができる．そのため，全ての循環参照の言語 C に属する部分が回収可能である．言語 C のローカルコレクタを起動して言語 C の到達不能オブジェクトを回収する．言語 C のローカルガベージコレクションでは言語 A と言語 B をまたぐ，言語 C を経由しない循環参照は回収されない．

次に言語 B のローカルコレクタを起動して言語 B の到達不能オブジェクトを回収する．このガベージコレクションによって全ての循環参照は，少なくとも一部が回収される．残りはローカルガベージコレクションだけで回収可能である．最後にワールドストップを解除する．

ソースコード 3　ハッシュ関数

```
1  uint64_t seed;
2  void hash(uint64_t* result, int n,
       uint64_t x) {
3    uint64_t h = x ^ seed;
4    for (int i = 0; i < n; i++) {
5      h = next_xorshift_1(h);
6      result[i] = h;
7    }
8  }
9  void update_seed() {
10   seed = next_xorshift_2(seed);
11 }
```

このように全ての言語に順序を与え，順序がより先の言語は自身より順序が後である全ての言語に参照関係情報を送信することで，３ノード以上の環境でも遠慮のかたまりを回収することが可能である．この方法では，改造が不要であるコレクタは順序が最後であるコレクタ 1 つだけである．

3.3 議論: bloom filter の偽陽性

Bloom filter [4] は，本来集合に含まれていない要素でも誤って集合に含まれると判定してしまう場合がある．この性質を疑陽性という．これは我々の手法では，本来到達不可能であるオブジェクト同士が誤って到達可能であると判定されることによるメモリリークの可能性があることを意味する．一方で，bloom filter に偽陰性はない．これは我々の手法では，到達可能なオブジェクトを誤って到達不可能であるとみなしてしまうことはないことを意味する．そのため，偽陽性のために回収できないオブジェクトは存在しうるが，偽陰性のためにオブジェクトを誤回収し，メモリエラーを起こしてしまうことはない．

偽陽の発生率は bloom filter のビット数と公開オブジェクトの数に依存する．また，公開オブジェクトとプロキシの間の参照関係が密であるほうが偽陽の確率が増す．公開オブジェクトの数が一定数を超えると遠慮のかたまりの回収効率が大幅に低下するため，アプリケーションに合わせた bloom filter のチューニングが重要である．

Bloom filter が使用するハッシュ関数をガベージコ

レクションごとにランダムに変化させることで，同じ組み合わせによる偽陽を避け，確率的に回収されるようにすることは可能である．我々が使用したハッシュ関数をソースコード 3 に示す．ハッシュ関数の内部状態を 64 ビット整数 `seed` で表現する．`n` 種類のハッシュ関数を使用する bloom filter を考える．あるオブジェクトの `n` 種類のハッシュ値はそのオブジェクトのアドレスと seed の xor を起点に，xorshift [8] を $\mathtt{n}-1$ 回適用することで取得する．ガベージコレクションが終わるたびに `seed` を xorshift で更新する．ハッシュの計算に使用するものと内部状態の更新に使用するものの 2 つの xorshift は互いに異なるパラメータのものを使用する．

4 関連研究

FFI 環境下で遠慮のかたまりが回収できない問題は，公開オブジェクトを参照カウント方式で管理することが原因である．Recycler [3] は参照カウント方式でありながら循環参照を回収することができるガベージコレクタである．Recycler が循環参照を回収するテクニックは試験的削除 (trial deletion) と呼ばれる．到達不可能であると思われるオブジェクト内の参照を試験的に削除し，その結果自身の参照カウントが 0 になった場合はそのオブジェクトは循環参照ゴミの一部であるとして回収するという方法である．試験的削除を FFI 環境に適用することは可能であるが，両方のコレクタの改造が必要となる．また試験的削除による参照カウントの増減は言語の境界を越えて波及するため，一度の試験的削除によって言語間の通信が複数回必要になる場合がある．言語間の通信がボトルネックであるような場合に性能が悪化するかもしれないという懸念もある．

メモリが分割された環境でのガベージコレクションという観点では，分散ガベージコレクションとも類似点が見られる．しかし我々の知る分散ガベージコレクションでは，通信のエラーがガベージコレクションを壊さないという耐障害性が問題意識として大きく扱われており，遠慮のかたまりの回収や実装の簡単化に取り組んだ研究は多くない．その中でも [11] はオブジェクトを領域間で移動させるアプローチであり，比較的我々の手法と似ている．しかしこれらの手法では部分的にしかオブジェクトを移動しないため遠慮のかたまりが回収できず，逆にメモリ消費が悪化する場合がある．

ウェブブラウザでは，Javascript エンジンとレンダラの間で遠慮のかたまりが問題となるようである．例えば Google Chrome では V8 と Blink の間で循環参照が問題となるため，cross-component tracing **?**[] という手法が提案されている．Cross-component tracing ではそれぞれのコレクタが自身が管理するメモリ領域を走査する方法を公開する．どのコレクタも他の領域を調べてオブジェクトの生死判定を行うことができるため，オブジェクトの生死判定を正確に行うことができる．この手法でもやはり両方のコレクタの改造が必要となる．

5 まとめ

本稿では FFI のためのガベージコレクションアルゴリズムを提案した．このアルゴリズムの特徴は一方の言語のガベージコレクタとして既存のコレクタを再利用し，改造を必要としないことである．ベースとなるアイデアは一方の言語のオブジェクトの参照関係からなるグラフをもう一方の言語のオブジェクトの間の参照関係として再現することで，無改造のコレクタでも全ての参照関係を調べることができるというものであった．

我々の手法は内部的に bloom filter を利用するため，全ての遠慮のかたまりを回収できない場合がある．Bloom filter によってどれだけのメモリが削減できるのか，偽陽性によってメモリ消費はどの程度増加するのか，我々のアルゴリズムはどのくらいの停止時間を要求するのかなどの実験による評価は今後の課題である．

我々の手法はストップ・ザ・ワールド方式である．ガベージコレクションが完了するまでユーザープログラムの計算を停止させる．インクリメンタル，あるいはコンカレントな手法で停止時間を削減することは挑戦的な課題である．

また我々の手法は 3 言語以上の FFI にも適用可能であるが，その停止時間は増加する．また，無改造で

済むコレクタはただ 1 つだけである．2 つ以上のコレクタを無改造に済ます方法もまた，挑戦的な課題である．

参 考 文 献

[1] : Google Code Archive - pyv8, https://code.google.com/archive/p/pyv8/.

[2] : libffi, https://sourceware.org/libffi/.

[3] Bacon, D. and Rajan, V.: Concurrent Cycle Collection in Reference Counted Systems, 08 2002.

[4] Bloom, B. H.: Space/Time Trade-Offs in Hash Coding with Allowable Errors, *Commun. ACM*, Vol. 13, No. 7(1970), pp. 422426.

[5] Collins, G. E.: A Method for Overlapping and Erasure of Lists, *Commun. ACM*, Vol. 3, No. 12(1960), pp. 655657.

[6] Foundation, P. S.: Python Documentation, https://docs.python.org/ja/3/library/ctypes.html.

[7] Johonson, S. G.: Calling Python functions from the Julia language, https://github.com/JuliaPy/PyCall.jl.

[8] Marsaglia, G.: Xorshift RNGs, *Journal of Statistical Software, Articles*, Vol. 8, No. 14(2003), pp. 1–6.

[9] Oracle: Java Documentation - Java Native Interface 仕様の目次, https://docs.oracle.com/javase/jp/8/docs/technotes/guides/jni/spec/jniTOC.html.

[10] Plainfossé, D. and Shapiro, M.: A Survey of Distributed Garbage Collection Techniques, *Proceedings of the International Workshop on Memory Management*, IWMM ’95, Berlin, Heidelberg, Springer-Verlag, 1995, pp. 211249.

[11] Vestal, S. C.: *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*, PhD Thesis, USA, 1987.