精確なガーベージコレクションのための局所変数登録誤りの発見

鵜川 始陽[1,a)]　藤本 太希[1,†1]

概要：精確なガーベージコレクション（GC）は，ルート集合となるポインタを精確に発見する必要がある．C 言語で実装された仮想機械（VM）では，C 言語の局所変数にもポインタが格納されている．そこで，GC がそれらを発見できるように，局所変数の値やアドレスを GC から参照できる表に登録する VM がある．しかし，VM のソースコード全体で間違いなく局所変数を登録したり解除する必要があり，間違いを起こしやすい．本研究では，局所変数が登録や解除できているかを VM のソースコードの制御フローグラフに対するパタンマッチによって検査した．検査は，我々が開発している JavaScript のサブセットの VM に対して行った．その結果，多くの登録漏れや冗長な登録を発見でき，この方法が有用であることが分かった．

キーワード：ごみ集め，バグ発見，プログラム解析

Finding Errors in Registrations of Local Variables for Accurate Garbage Collection

Tomoharu Ugawa[1,a)]　Taiki Fujimoto[1,†1]

Abstract: Accurate garbage collection (GC) has to find all pointers belonging to the root set. In a virtual machine (VM) implemented in C language, local variables of C language may have pointers. Thus, some VM registers the values or the addresses of local variables to a table that is visible to GC. However, this approach is error-prone because it requires to register and unregister local variables correctly though the entire source code of the VM. In this research, we checked if local variables are registered and unregistered correctly by pattern matching against control flow graphs of source code of the VM. We applied this check to the VM of a subset of JavaScript that we are developing. As a result, we found this approach is effective because it found many registrations missing and redundant registrations.

Keywords: garbage collection, bug finding, program analysis

1. はじめに

コンパクションやコピー方式のガベージコレクション（GC）のようにオブジェクトを移動させる GC では，オブジェクトを移動させた時，そのオブジェクトを指しているポインタを全て移動先を指すように更新する．そのため，GC のルートのアドレスを精確に特定する必要がある．通常，GC を持つ高級言語のインタプリタや仮想機械（VM）を C 言語で実装する場合，ルート集合にはヒープ中の GC で管理されたデータへのポインタ（以下，単にヒープへのポインタ）を持つ局所変数も含まれる．

しかし，このような局所変数を精確に列挙できるようにプログラミングするのは手間がかかり，間違いも生じやすい．局所変数は実行スタック上に作られるが，GC は実行スタックのどこに局所変数があるかや，その値がヒープへのポインタかどうかは判断できない．そこで，スタックマップのようなコンパイラによる支援の仕組みを作るか，VM のプログラム自身で明示的に局所変数をルート集合に登録，登録を解除する方法がとられる．

我々は組込みシステム向けの JavaScript VM である

[1] 高知工科大学
Kochi University of Technology
[†1] 現在，株式会社トスコ
Presently with TOSCO Corp.
[a)] ugawa.tomoharu@kochi-tech.ac.jp

```
1  get_object_prop
2  (Context *ctx, JSValue o, JSValue p) {
3    JSValue ret = NULL;
4    GC_PUSH(o);GC_PUSH(p);GC_PUSH(ret);
5    if (!is_string(p))
6      p = to_string(ctx, p);
7    do {
8      if (get_prop(o, p, &ret) == SUCCESS) {
9        GC_POP(ret);GC_POP(p);GC_POP(o);
10       return ret;
11     }
12   } while (get___proto__(o, &o) == SUCCESS);
13   GC_POP(ret);GC_POP(p);GC_POP(o);
14   return JS_UNDEFINED;
15 }
```

図 1 素朴に GC_PUSH と GC_POP を挿入した eJSVM のソースコード
Fig. 1 Source code of eJSVMinserted GC_PUSH and GC_POP straightforwardly.

eJSVM [1], [2] を開発している．現在の eJSVM にはオブジェクトを移動しないマークスイープ GC を実装しているが，将来オブジェクトを移動する GC を実装することを計画している．そのために，VM のソースコードの中でヒープへのポインタを持つ局所変数のアドレスを個別にルート集合に登録，解除している．例えば，図 **1** は eJSVM のソースコードの一部を説明用に改変したものである．ここで，GC_PUSH は引数のアドレスをルート集合に登録するマクロ，GC_POP は解除するマクロであり，JSValue 型の変数がヒープへのポインタを持つ．関数の先頭で引数を含む全ての JSValue 型の局所変数を登録し，関数を抜ける直前で全て解除している．return 文の直前でも解除する必要がある．

しかし，GC_PUSH と GC_POP を正しく追加しながらプログラミングするというのは，通常のプログラミングにはない規則に従ってプログラムを書くことであり，間違いを起こしやすい．そのうえ，以下のような理由で，間違いの余地は大きくなっている．

- プログラマは不要と思った処理を省いたりしがちで，特に eJSVM では VM のフットプリントを小さく抑えるという不要な処理を省く動機もある．例えば実際の eJSVM のソースコードでは図 **2** に示す位置に GC_PUSHGC_POP が挿入されている（詳細は 2.3 節参照）．
- プログラムを修正する際にも GC_PUSH と GC_POP が正しく行なわれるように修正する必要がある．
- eJSVM では，VM 開発者以外もハードウェアにアクセスするための組込み関数を C 言語で記述して追加することを想定している．

さらに，GC_PUSH と GC_POP が正しく追加されていなかったとしても，実行時に誤動作する頻度やその再現性は低く，

```
1  get_object_prop
2  (Context *ctx, JSValue o, JSValue p) {
3    JSValue ret;
4    if (!is_string(p)) {
5      GC_PUSH(o);
6      p = to_string(ctx, p);
7      GC_POP(o);
8    }
9    do {
10     if (get_prop(o, p, &ret) == SUCCESS)
11       return ret;
12   } while (get___proto__(o, &o) == SUCCESS);
13   return JS_UNDEFINED;
14 }
```

図 2 eJSVM のソースコード
Fig. 2 Source code of eJSVM.

通常のテストによるバグの発見やデバッグは難しい．

そこで，GC_PUSH と GC_POP が正しく追加されているかを検査する方法を検討した．その中で，Coccinelle [3], [4] という，プログラムの制御フローグラフに対してパタンマッチするツールを使ったところ，高い精度で GC_PUSH 漏れを発見できた．Coccinelle は Linux カーネルのリファクタリングやバグ発見に用いられているツールで，ユーザがドメイン特化言語（DSL）で書いた制御フローグラフのパタンを使ってパタンマッチする．

検査の精度を調べるために，これまで開発者が手動で追加していた GC_PUSH と GC_POP を全て取り除き，検査によって取り除いた箇所を指摘できるか実験した．その結果，本来は不要であった箇所を除いて全て指摘することができた．また，手動で GC_PUSH が追加されていなかった箇所も指摘され，これらは 1 箇所を除いて全て本来は GC_PUSH が必要な箇所だった．

本論文では，局所変数の登録と解除の処理を忘れるバグを Coccinelle を使って探し，効率的にデバッグが行えた経験を，システムの開発経験として報告する．さらに，本手法を別のプロジェクトのソースコードにも適用した実験の結果も報告する．

2. eJSVM でルート集合への登録と解除

2.1 基本的な仕組み

eJSVM は，ルート集合として登録された局所変数のアドレスを，スタック（以下，GC ルートスタック）で管理している．図 **3** はある関数 f が関数 g を呼び出した時の制御スタックと GC ルートスタックの様子を示している．局所変数は制御スタック中の関数フレームに割り当てられる．そのため，ルート集合として登録される局所変数もスタック構造で管理すると効率ががよい．

GC やコンパクションによってオブジェクト A が A' に

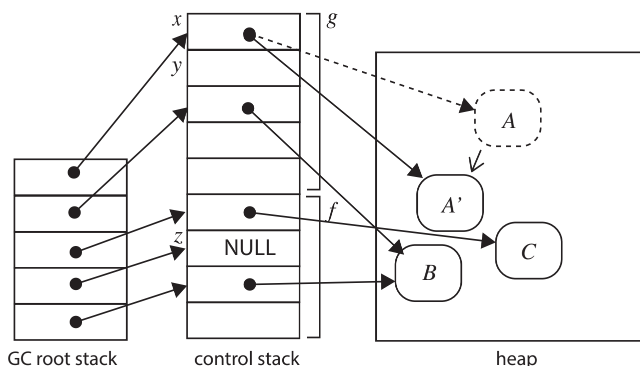

図 3 GC ルートスタック
Fig. 3 GC root stack.

移動すると，GC は変数 A を指していた変数（図では x）の値を移動先を指すように更新する．これが可能なように，GC ルートスタックには局所変数のアドレスをプッシュする．

GC は GC ルートスタックから指される変数をデリファレンスする．そのため，未初期化の変数は GC ルートスタックにプッシュしてはいけない．例えば，この先実行が進んで変数 y にポインタが代入されることが分かっていたとしても，y が未初期化の状態で y のアドレスをプッシュすると，GC が起こった時にたまたま入っていた値をポインタとしてデリファレンスしてしまう．変数 z のように NULL で初期化しておけば，GC はそれをデリファレンスしない．

VM のプログラムは，GC_PUSH と GC_POP マクロを使って，局所変数を GC ルートスタックにプッシュしたり GC ルートスタックからポップしたりする．eJSVM には GC_PUSH と GC_POP の対応がとれているかを実行時にチェックするデバッグ用のビルドオプションがあり，そのために GC_POP も変数を引数として受け取る．この変数はデバッグビルドでしか使われない．

2.2 ヒープへのポインタを持つ変数

eJSVM では，ヒープへのポインタを持つかどうかは型で判断できる．具体的には，JavaScript の値を格納する JSValue 型と，hidden class [5] など，いくつかの VM が内部的に使う構造体へのポインタ型だけがヒープへのポインタを持つ．JSValue 型の変数は整数値などポインタ以外の値を保持することもあるが，下位ビットに付けたタグで GC がポインタかどうかを判定できるようになっている．VM が内部的に使う構造体については，型によってヒープに作られるか，それ以外のアドレスに作られる（例えば静的に確保される）かを区別できるようになっている．coccinelle を使った検査では，JSValue 型とヒープに作られる構造体は完全に同じように扱うため，以降ではこれらをまとめて単に JSValue 型と書く．

2.3 無駄のない GC_PUSH と GC_POP の追加

図 1 では，関数の入口で三つある全ての JSValue 型の全ての変数を GC_PUSH し，2 箇所ある関数の出口の両方で，全てを GC_POP している．さらに，関数内で宣言された局所変数 ret は，プッシュの前に NULL で初期化している．これで正しく動作するが，無駄に GC_PUSH や GC_POP をしている箇所がある．これらの無駄な GC_PUSH や GC_POP は実行のオーバヘッドになるだけでなく，VM のフットプリントが大きくなる．組込みシステム向けの eJSVM では，このような無駄は避けたい．

JavaScript のプログラムはシングルスレッドなので，マルチスレッドのプログラムのように，他のスレッドによって GC が起動されるということはない．eJSVM では GC をするには VM の実行コンテキストが必要になる．そのため，GC が起こる可能性がある箇所は，Context 型のポインタを引数として渡している関数呼出しに限られる．GC ルート集合は，GC の時に精確であればよいので，変数の生存区間を考えて GC_PUSH や GC_POP を削ることができる．

例えば図 1 では，6 行の to_string 関数の呼出ししか GC を起こす可能性がない．この呼出しをまたいで生存区間がある変数は o だけである．ret に最初に値が格納されるのは，8 行目なので，GC_PUSH する必要はない．8 行目では，get_prop 関数にポインタ引数として渡して値を格納している．

p についてはもう少し複雑である．p には引数として値が渡ってきて，例えば 8 行目などで使う．しかし 6 行目の to_string の戻り値を書き込んでいるため，生存区間はここで分かれている．仮に 6 行目の中で GC が起こり，p に引数として渡ってきたポインタが指すオブジェクトが移動していたとしても，p は to_string の戻り値（有効なオブジェクトを指すはず）で上書きされる．そのため，p を GC_PUSH しておく必要はない．

変数 o を GC_PUSH するのは，関数の入口でなくともよい．6 行目の to_string が実行されるのは p が文字列データでない時に限られる．この if 文の then 節の中で GC_PUSH と GC_POP をすれば，GC_POP を 1 箇所にまとめられる．さらに，図 1 で定義している get_object_prop 関数はオブジェクトのプロパティを取得する関数であり，プロパティ名の p には文字列データが与えられる可能性が高い．したがって，この if 文の条件が成立する可能性は低く，実行速度の面でもメリットがある．

以上のことを考慮すると，無駄なく GC_PUSH と GC_POP を追加したプログラムは図 2 のようになる．

2.4 実行経路を考慮した GC_PUSH の追加

GC_PUSH が必要かどうかは，プログラムの実行経路によって判断しなければならない．図 4 に組込み関数 Array.prototype.concat のプログラムを示す．このプロ

```
 1  JSValue e;
 2  ...
 3  e = args[i];
 4  ...
 5  while (k < len) {
 6    if (has_array_element(e, k)) {
 7      subElement =
 8        get_array_prop(ctx, e,
 9                       cint_to_fixnum(k));
10      set_array_prop(ctx, a, cint_to_fixnum(n),
11                     subElement);
12    }
13    n++;
14    k++;
15  }
```

図 4　実行経路を考える必要があるプログラムの例
Fig. 4　Example of program that we need to consider execution path.

グラムで JSValue 型の変数 e は，8 行目の GC を起こす可能性がある関数呼出しより前にルート集合に登録しておく必要がある．なぜなら，その直前の 6 行目で使っているからである．この行はプログラムテキストでは上の行になっているが，while 文の中にあり，8 行目よりも後にも実行される可能性がある．

3. Coccinelle

プログラムの実行経路に対するパタンマッチをするために，Coccinelle [3], [4] というツールを使った．Coccinelle は，SmPL という DSL で記述されたルールを受けとり，プログラムの制御フローグラフに対してパタンマッチする．さらに，パタンマッチした箇所を書き換える機能もあり，ルールはセマンテックパッチと呼ばれている．Linux カーネルの開発プロジェクトではソースコードのリファクタリングやバグの発見に使われている．

SmPL では C 言語のプログラムの断片や「...」といった省略の記号を使ったパタンでルールを記述する．Coccinelle はこのパタンを，メタ変数で拡張された計算木論理式（CTL-V 式）に変換し，制御フローグラフをクリプキ構造とみなしてモデル検査によってパタンマッチする．Coccinelle のクリプキ構造では，状態はプログラムの文（statement）であり，その属性はルール記述した文の条件に合致するかどうかになっている．文の条件には，メタ変数を含む文や，ある式を含む文という条件が指定できる．内部表現が CTL なので，「...」などの省略記号には exists や forall を指定できる．

SmPL では複数のルールを記述し，先行するルールのパタンがマッチした変数を後続のルールのパタンマッチで利用することができる．この機能を使ってマッチする条件を追加したり，偽陽性を取り除いたりできる．

```
@MissingPush depends on use && !falseuse && !immass@
identifier pre.push, decl.v;
expression e, gc.gc_fun;
position gc.gc_p, decl.decl_p;
type decl.T;
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)
... when != push(&v)
    when exists
gc_fun@gc_p
```

図 5　GC_PUSH 漏れを探すルール
Fig. 5　Rule to find missing GC_PUSH.

図 5 のルールを例に SmPL の文法を説明する．ルールの動作は 4 章で説明するが，このルールは GC_PUSH 漏れの検査のためのルールである．ルールの先頭には，ルール名 MissingPush と，このルールが依存しているルール use，falseuse，immass を書く．これによって，use にマッチし，falseuse と immass にマッチしないようなメタ変数の組合わせについて，このルールのパタンマッチが試みられる．

以降の「@@」までの行はメタ変数を定義している．decl.v のような記述で，先行するルールのメタ変数を利用できる．メタ変数には，プログラムの構文要素の他に，プログラム中の出現位置を持つ position 型がある．

「@@」以降ではパタンの本体を定義する．「(|)」は選択を，「...」は途中に任意の数の文を含むことを表す[*1]．「...」には条件を付けることができ，「when != push(&v)」は，push(&v) を含む文が存在しないことを表す．「when exists」は先行する文に続くいずれかの実行経路で，以降のパタンが満たされることを表す[*2]．exists は実行経路の「...」に対応する部分だけに対して存在することを指定しているのではなく，関数の最後までの実行経路に対して以降のパタンにマッチする経路が存在することを指定している．例えば図 6 のプログラムように複数の実行経路がある場合に，そのいずれか一つ以上の実行経路で「gc_fun@gc_p」のパタンにマッチすることを指定しており，一つの実行経路でしかマッチしない図 6 のプログラムにも図 5 のルールはマッチする．exists の指定がなければ，図 6 のプログラムにはマッチしなくなる．

各構文要素には「@decl_p」のように，出現位置を表すメタ変数を付けることができる．例えば，「gc_fun@gc_p」は，メタ変数の宣言と合わせると，gc ルールで発見した関数 gc_fun を，gc ルールで発見した位置で呼び出している文を表すことになる．なお，セミコロンを付けると文を表

[*1] 「...」がなければ，後続の文は時相演算子 X（next）で連結されるが，「...」があれば U（until）で連結される．
[*2] 時相演算子 EU に変換される．

```
1 JSValue x = NULL;   /*T v@decl_p = e; にマッチ*/
2 if (flag == 1)
3   cause_gc(context);/*gc_fun@gc_p にマッチ*/
```

図 6 片方の経路でしか GC しないプログラム
Fig. 6 Program that causes GC on one path but the other.

```
1     JSValue x, y;
2     y = cause_gc(ctx);
3     x = arg[1];
4     return x == y;
```

図 7 JSValue 型変数への代入があるプログラム
Fig. 7 Program that has assignment to JSValue-type variable.

すことになり，セミコロンを付けなければ，その式を含む文を表すことになる．

Coccinelle は，OCaml や Python を使ってルールの一部を記述できる．本研究では，この機能を使ってメタ変数の条件を Python で記述したり，ルールの本体を Python で記述して，バグを発見した際の画面出力したりしている．

4. バグ検出ルールの作成

本研究では，Coccinelle を利用して次のバグを探すルールを作った．

- GC_PUSH 漏れ
- JSValue 型変数のアドレスの取得と変数への保存
- 二重の GC_PUSH
- GC_POP 漏れ
- GC_PUSH 前の GC_POP
- 早過ぎる GC_POP
- 二重の GC_POP
- JSValue 型以外の変数の GC_PUSH

「JSValue 型変数のアドレスの取得と変数への保存」は，それ自身はバグではない．しかし，取得したアドレスを変数に保存しておき，その変数が GC_POP された後に使う可能性があり，そうなるとバグになる．現在の eJSVM のソースコードには，JSValue 型変数のアドレスを取得して他の変数の初期値にしたり代入している箇所はなく，今後もこのようなプログラムを禁止しても不都合はなさそうなので，保守的で単純なルールにした．

「早過ぎる GC_POP」は，それ以降で GC が起こる可能性が残っているのに，GC 後に使われる変数を GC_POP してしまうバグである．

以降では，最も複雑な GC_PUSH 漏れを探すルールの設計と実装を説明する．他のルールの設計と実装も同様の手順で行った．また，GC_PUSH 漏れを探すルールは変数が引数の場合と局所変数として宣言されている場合でルールを分けて作ったが，この二つは本質的には同じなので，変数が局所変数として宣言されている場合に絞って説明する．

4.1 GC_PUSH 漏れバグの条件

図 2 から GC_PUSH を消したプログラムや図 4 のプログラムにマッチするようなルールを作った．まず，GC_PUSH が必要になるような条件を列挙した．

- **gc**: GC を起こす可能性のある関数 *gc_fun* の呼出しがあること．この位置を *gc_p* とする．
- **decl**: JSValue 型の変数 v が宣言されていること．この位置を *decl_p* とする．
- **use**: v は *gc_fun* の呼び出し後のどこかの位置 *use_p* で利用されていること．

この条件が全て満たされていると，*decl_p* と *gc_p* の間で v の GC_PUSH が必要になることがある．例えば図 4 では，条件 **gc** には 8 行目や 10 行目が，条件 **decl** には 1 行目が該当する．そして，条件 **gc** にどちらの行が該当した場合でも，条件 **use** に 6 行目が該当する．したがって，**gc**，**decl**，**use** の条件の全てに該当しており，*decl_p* は 1 行目，*gc_p* は 8 行目や 10 行目になる．実際，このプログラムでは 8 行目が実行されるより前に GC_PUSH が必要になる．

しかし，例えば，図 **7** のプログラムでは実際には GC_PUSH は必要ないが，**gc**，**decl**，**use** の条件には全て該当してしまう．変数 x の GC_PUSH が必要ないのは，2 行目の GC 後，4 行目で変数 x を使うまでに 3 行目で変数 x に代入しているからである．また，変数 y は，2.3 節で説明した図 2 の変数 p と同じ理由，つまり，2 行目の代入は関数呼出しよりも後で行われるため，GC_PUSH する必要がない．これらの場合を無視するために，次の条件を追加を満たす場合はバグとして検出しない．以降，このように否定的に用いられる条件には肩にマイナス記号を付ける．

- **assign**$^-$: *gc_p* と *use_p* の間には v への代入がある．
- **immass**$^-$: *gc_p* の呼び出しと同じ文で，*gc_p* より後に v への代入がある．

eJSVM には図 2 の 10 行目のように，変数 ret のアドレスを関数に渡して，その変数経由で関数から値を戻すようなプログラムもある．これも効果は代入だが，呼び出される関数が変数に値を代入しない可能性もあるので，保守的に GC_PUSH 漏れのバグとして扱う．

4.2 GC_PUSH 漏れを探すルールの設計

GC_PUSH 漏れを探すルールは図 **8** に示す小さなルールに分けて実現した．

4.1 節で列挙した条件 **gc**，**decl**，**use**，**immass**$^-$ はそれぞれ個別のルールで実現した．さらに，これらのルールで検出した変数宣言の位置 *decl_p* と GC を起こす可能性がある関数呼出し *gc_p* の間に GC_PUSH がない場合にバグを検出するルール **MissingPush** を作った．**assign**$^-$ の条件は，**use** の条件を実現するルールに実行経路の条件として

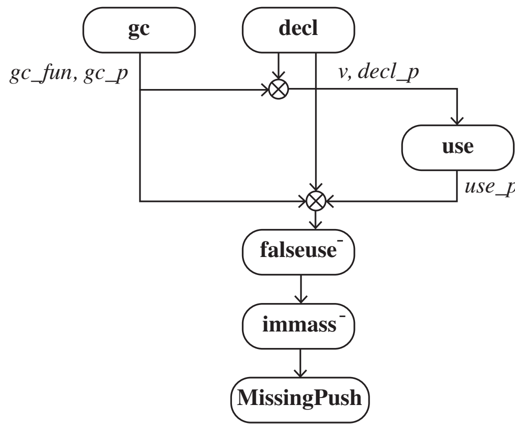

図 8　パタンの依存関係
Fig. 8　Depenency of patterns.

加えた．

use のルールを作る際に，「変数 v への代入以外の変数 v の利用」を精確に表現できず，変数 v に代入している箇所も利用として **use** が検出してしまった．また，GC_PUSH や GC_POP の引数に変数 v を渡す文も v の利用として検出してしまった．これらの偽陽性を取り除くために，次のルールを追加した．

- **falseuse**⁻: use_p の利用は考慮する必要がない．

4.3　GC_PUSH 漏れを探すルールの実装

この節では我々が作ったルールを説明する．まず **MissingPush** 以外のルールを図 **9** に示して説明する．その後，図 5 で示した **MissingPush** のルールを図 **10** に再掲して説明する．

gc のルールは，コンテキストを引数にとる関数の呼出しにマッチする．メタ変数の定義中にある

```
type pre.Context;
Context ctx;
```

は，まず，Context*型にマッチするように pre ルールの中で Python で定義した型 Context を，メタ変数として利用することを宣言している．それを使って，その型を持つメタ変数 ctx を導入している．**gc** のルールは，この ctx を引数にとる関数呼出しにマッチする．関数ポインタを使った場合でもマッチするように，gc_fun は識別子にマッチする identifier 型ではなく，式にマッチする expression 型にしている．

decl のルールは，JSValue 型やヒープ中に作られる VM 内部の構造体へのポインタ型の変数宣言にマッチする．

```
type T:script:python(){is_pointer(T)};
```

は，メタ変数 T に Python のプログラム is_pointer(T); が真になるような型だけにマッチするという条件を付けている．is_pointer は，引数が JSValue 型か，VM 内部の構造体へのポインタ型のときに真になる．**decl** のルールは，このような型の変数宣言にマッチする．変数宣言の初期化子はオプショナルなので，両方のパタンを書いている．

```
@gc@
expression gc_fun;
position gc_p;
type pre.Context;
Context ctx;
@@
gc_fun@gc_p(..., ctx,...)

@decl@
identifier v;
expression e;
position decl_p;
type T:script:python(){is_pointer(T)};
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)

@use depends on gc && decl@
identifier decl.v;
expression e, gc.gc_fun;
position gc.gc_p, use_p;
@@
gc_fun@gc_p
... when != v = e
    when exists
v@use_p

@falseuse depends on use@
identifier pre.push, pre.pop, decl.v;
expression e;
position use.use_p;
@@
(
 v@use_p = e
|
 push(&v@use_p)
|
 pop(&v@use_p)
)

@immass depends on use@
identifier decl.v;
expression gc.gc_fun;
position gc.gc_p;
@@
v = <+... gc_fun@gc_p ...+>
```

図 9　GC_PUSH 漏れを発見するルール
Fig. 9　Rules to find missing GC_PUSH.

use のルールは **gc** と **decl** のルールで発見した変数 v と関数呼出し gc_p に対して，関数呼出し gc_p の後に v が使われるプログラムにマッチする．gc_p と v の利用の間の「...」には「when != v = e」の条件を付けて，**assign**⁻ の条件を満たす場合はマッチしないようにしている．なお，関数呼出し gc_p の後に分岐がある場合，そのいずれかの実行経路上で使われている変数は全て見つけたいので，

```
@MissingPush depends on use && !falseuse && !immass@
identifier pre.push, decl.v;
expression e, gc.gc_fun;
position gc.gc_p, decl.decl_p;
type decl.T;
@@
(
 T v@decl_p;
|
 T v@decl_p = e;
)
... when != push(&v)
    when exists
gc_fun@gc_p
```

図 10　GC_PUSH 漏れを探すルール（再掲）
Fig. 10　Rule to find missing GC_PUSH (same as Fig. 5).

「...」には「when exists」も付けている．

falseuse⁻ のルールは，**use** のルールで発見した変数 v の利用が変数 v への代入や，変数 v を GC_PUSH や GC_POP に渡しているときにマッチする．

immass⁻ のルールは gc_p の関数呼出しと同じ文の中で，gc_p の関数呼出しより後に変数 v に代入されるプログラムにマッチする．

```
v = <+... gc_fun@gc_p ...+>
```

の左辺は gc_fun@gc_p を含む式を表している．

図 10 に再掲した **MissingPush** のルールでは，これまでに説明したルールを図 8 のように組合わせるために，

```
depends on use && !falseuse && !immass
```

という依存関係を書いている．**gc** と **decl** は **use** を介して間接的に依存しているので，ここには書いていない．**MissingPush** のルールは，宣言から GC を起こす関数呼出しの間に GC_PUSH を含まない文だけからなる経路が存在するプログラムにマッチする．これを表現するために，「...」には「when != push(&v)」の条件と exists を付けている．

5. 実験と事例研究

4 章で作ったルールを使って eJSVM のソースコードを検査した．eJSVM のソースコードにはマクロが使われているので，プリプロセッサで処理した後のソースコードを検査した．このとき，GC_PUSH や GC_POP は関数呼出しとして残るようにした．

5.1 精度の調査

4 章で作成したルールの精度を調べるために，eJSVM に開発者が追加していた GC_PUSH と GC_POP を全て取り除いて，取り除いた箇所を指摘できるか調べた．

eJSVM のソースコードでは合計で 32 箇所で GC_PUSH が使われていた．4 章で作成したルールでは，このうち 26 箇所を，GC_PUSH 漏れとして発見した．残る 6 箇所は実際には GC_PUSH が必要ない箇所だった．実際には必要がなかった GC_PUSH を含むプログラムを図 **11** に示す．1 行目で nexth と oh が GC_PUSH されているが，対応する GC_POP までの間に GC が起きる可能性はない．

```
 1 GC_PUSH(nexth); GC_PUSH(oh);
 2 r = hash_put_with_attribute(
 3         hidden_map(nexth),
 4         name, index, attr);
 5 if (r != HASH_PUT_SUCCESS) {
 6   GC_POP(oh); GC_POP(nexth);
 7   return FAIL;
 8 }
 9 hidden_n_entries(nexth)++;
10 hash_put_with_attribute(
11     hidden_map(oh), name,
12     (HashData)nexth,
13     ATTR_NONE | ATTR_TRANSITION);
14 GC_POP(oh); GC_POP(nexth);
```

図 **11**　不要な GC_PUSH
Fig. 11　Redundant GC_PUSH.

以上のように，開発者が追加していた GC_PUSH のうち必要な GC_PUSH は全て発見することができ精度が高いことが確認できた．

本研究で探したバグのうち GC_PUSH 漏れ以外のバグ以外は GC_PUSH か GC_POP がなければ成立しない．そのため，この実験では GC_PUSH 漏れ以外のバグは見つからなかった．

5.2 eJSVM のバグの発見

4 章で作成したルールを eJSVM に適用し，バグを探した．その結果を表 **1** に示す．「手動」は開発者が追加していた GC_PUSH のうち本当に必要だったものの数，「漏れ」は 4 章で作ったルールで発見した本当の GC_PUSH 漏れの箇所の数，「偽陽性」は 4 章で作成したルールで検出されたものの，本当は不要だった箇所の数を示している．なお，GC_PUSH 漏れ以外のバグは発見されなかった．

4 章で作成したルールは，かなりの数の GC_PUSH 漏れを報告した．報告された箇所を全て確認したところ，1 箇所を除いて全て，本当に必要な GC_PUSH が漏れていたことが分かった．その中には図 4 に示した箇所も含まれる．

1 件の偽陽性は制御変数の値により 2 箇所ある if 文の分岐先の組合わせが制限されるようなプログラムだった．該当箇所を図 **12** に示す．このプログラムの lowerValue と upperValue が JSValue 型である．検査ではどちらも GC_PUSH 漏れとして報告された．しかし，実際には upperValue は GC_PUSH する必要がない．upperValue が報告された理由は

- 3 行目の set_array_prop 関数の呼出しが GC を起こす可能性がある関数として **gc** の条件に適合し，

表 1 GC_PUSH 漏れの数
Table 1 Number of missing GC_PUSH.

ファイル	行数	手動	漏れ	偽陽性
allocate.c	228	3	0	0
builtin-array.c	797	3	23	1
builtin-boolean.c	73	0	3	0
builtin-global.c	277	0	2	0
builtin-math.c	257	0	1	0
builtin-number.c	175	0	3	0
builtin-object.c	110	0	2	0
builtin-regexp.c	273	0	0	0
builtin-string.c	692	6	6	0
call.c	251	0	0	0
codeloader.c	761	0	0	0
context.c	168	1	0	0
conversion.c	669	0	6	0
gc.c	1066	0	0	0
hash.c	474	0	0	0
init.c	114	0	0	0
main.c	429	0	0	0
object.c	1030	14	18	0
string.c	189	0	0	0
unix.c	10	0	0	0
vmloop.c	1426	0	0	0
合計	9469	27	64	1

表 2 AVL 木のプログラムに対する GC_PUSH 漏れの数
Table 2 Number of missing GC_PUSH in AVL-tree program.

ファイル	行数	手動	漏れ	偽陽性
avltree.c	638	28	7	0

- 9 行目の `upperValue` の利用が **use** の条件に適合し，さらに
- その間に `upperValue` への代入が存在しない経路（5 行目の `if` 文の条件が成立しない場合）が存在する

からである．しかし，5 行目の `if` 文の条件が成立しないのは `upperExsits` が偽の場合で，その場合はその後に 9 行目などの `upperValue` を使う分岐には入らない．Coccinelle は制御フローグラフに対するパタンマッチを行うので，このようにデータに依存した制御は保守的に扱うしかなく，偽陽性の可能性がある．しかし，そのようなプログラムはプログラマにとっても見通しが悪く，あまり書かれることはないと考えている．

5.3 eJSVM 以外のプログラムの検査

eJSVM 以外のプログラムに対しても，4 章と同様の方法で Coccinelle のルールを作り，GC_PUSH 漏れのバグを探した．対象には，他の研究グループが開発している GC ライブラリの，テスト用のプログラムを用いた．このプログラムは C 言語で書かれた AVL 木のプログラムで，節点を GC で管理されたヒープに起く．節点へのポインタを持つ局所変数は，eJSVM の GC_PUSH と同様のマクロを使って，アドレスを GC ルートスタックにプッシュする．このプログラムでは，デバッグ用の関数以外の全ての関数呼出しの中で GC が起こる可能性がある．

実験の結果を**表 2** に示す．この結果には，開発者が追加していた GC_PUSH のうち実際には不要だった 10 箇所は含んでいない．この実験で用いたプログラムには，多くの GC_PUSH が保守的に挿入されていた．それにもかかわらず，7 箇所の GC_PUSH 漏れを検出した．これら 7 箇所は，開発者に確認したところ全て必要な GC_PUSH が漏れていたものと分かった．

6. 議論

6.1 制御フローグラフに対するパタンマッチの限界

Coccinelle では，制御フローグラフに対してパタンマッチする．そのため，式の値に依存して実行経路が制限されるような場合にはバグを誤検出する可能性がある．eJSVM に対する適用では，このような例は図 12 の 1 箇所だけだった．この例では二つの `if` 文の分岐先の組合わせが制限される．他にも，ループの継続条件とヒープへのポインタを持つ変数が使われるかどうかが関連している場合なども考えられる．

eJSVM ではデバッグ用に GC_POP にも GC ルートスタックの先頭にあることが期待される変数を渡している．デバッグビルドすると実行時にはこれを使って GC_PUSH と GC_POP の対応を検査する．Coccinelle でこの検査をすることは難しい．例えば，変数 `a`，`b` の順で GC_PUSH され，同じ順で GC_POP されている場合はバグとして検出したいが，そのようなルールを一般化して作ることはできない．

6.2 GC_PUSH の自動挿入

Coccinelle ではパタンにマッチした箇所のプログラムを書き換えるようなルールも使える．それを使えば，例えば変数宣言の直後に GC_PUSH を自動的に挿入することもできる．しかし，それでは冗長な変数の初期化が行われる可能性がある．

最適な GC_PUSH の挿入位置は，そのプログラムの意味まで踏み込んで考える必要があり，それを自動化することは難しい．意味まで考えなくとも，例えば「制御フローグラフ上で，GC を起こす可能性がある関数呼出しの全てを支配する節点のうち，関数の入口から遠いものの中から選ぶ」という規則でもかなり効率のよいプログラムは作れるかもしれない．しかし，このような複雑なルールを SmPL で記述すると，ルールのバグの可能性が増える．

本研究では Coccinelle を用いたが，コンパイラの内部表現の上で GC_PUSH 漏れを探したりを自動挿入するという方法も考えられる．しかし，既存ツールを利用するのに比べると手間がかかる．

```
1  for (lower = 0; lower < mid; lower++) {
2    if (lowerExists)
3      lowerValue = get_array_prop(context, args[0], cint_to_fixnum(lower));
4    upperExists = has_array_element(args[0], upper);
5    if (upperExists)
6      upperValue = get_array_prop(context, args[0], cint_to_fixnum(upper));
7
8    if (lowerExists && upperExists) {
9      set_array_prop(context, args[0], cint_to_fixnum(lower), upperValue);
10     set_array_prop(context, args[0], cint_to_fixnum(upper), lowerValue);
11   } else if (!lowerExists && upperExists) {
12     set_array_prop(context, args[0], cint_to_fixnum(lower), upperValue);
13     delete_array_element(args[0], upper);
14   } else if (lowerExists && !upperExists) {
15     set_array_prop(context, args[0], cint_to_fixnum(upper), lowerValue);
16     delete_array_element(args[0], lower);
17   } else {
18     /* No action is required */
19   }
20 }
```

図 12　偽陽性の箇所

Fig. 12　Program causing a false positive.

6.3　eJSVM の開発への利用

5.1 節で行ったように GC_PUSH と GC_POP を全て消して検査すると GC_PUSH が必要が箇所が分かる．eJSVM の開発プロジェクトでは，本研究の後，実際に全ての GC_PUSH と GC_POP を消して検査し，必要な GC_PUSH と GC_POP だけを追加した．このとき，GC_PUSH の挿入箇所は開発者が考えながら挿入したが，本論文の著者ではない教員 1 名と，著者を含む学生 3 名で作業して 3 時間程度で終わった．

eJSVM では，本研究で開発したルールを適用するメイクファイルのターゲットを作って，ソースコードを変更したらすぐにバグを検査できるようにしている．これによって，本研究でのデバッグ後にも，追加されようとしたバグを発見した．検査には通常のデスクトップ計算機でも 1 分程度かかるが，これは eJSVM で使っている回帰テストの時間より短かい．そのため，実用上，検査時間が問題になることはない．

7.　関連研究

7.1　ルート集合の管理

eJSVM では局所変数のアドレスを明示的に GC ルートスタックにプッシュする方法でルート集合を特定したが，GC ルートとなる局所変数を探す方法として，いくつか他の方法も知られている．コンパイラを変更できる場合は，VM のコンパイル時に，関数フレームのどこにポインタがあるかを記録したスタックマップを生成する方法が一般的に行われている．

Henderson [6] は，関数毎にヒープへのポインタを持つ変数をまとめた構造体を作り，それをスタックに置く方法を提案している．その構造体をリストつないでおけば，GC はそのリストをたどって構造体を探すことができる．Henderson はプログラム変換によってこのような構造体を導入することを提案している．開発者が手でこのような構造体を導入するのは，関数毎に構造体定義を作る必要があり，現実的ではない．

この他にも，ヒープへのポインタは制御スタックとは別のスタックにも積んでおくシャドウスタックと呼ばれる技法も知られている．また，Java Native Interface（JNI）や V8 JavaScript エンジンでは，局所変数がヒープへのポインタを保持する必要があるときはハンドルに変換するという方法をとっている．この方法では，ハンドルを解放するのを忘れるバグの可能性がある．ただし，C++ではスコープを抜ける際にデストラクタで解放するという技法を使うことができ，V8 ではそのようにしている．

最後に，スタック上の全てのワードの値の中で，オブジェクトの先頭へのポインタのように見えるものは全てポインタだとして扱う保守的 GC もよく知られている [7]．しかし，保守的 GC ではオブジェクトを移動させることはできない．

7.2　C 言語を対象にしたパタンマッチによるバグ発見

Coccinelle は Linux カーネルプロジェクトのために開発されていた．主に Linux のバグ発見やリファクタリングに使われている [8], [9]．OS 以外のオープンソースソフトウェアにも利用されており，OpenSSL のバグ発見に使っ

た研究もある [10]．また，Coccinelle の開発プロジェクトは，NULL ポインタの参照を発見するルールなど，多くの例を紹介している[*3]．しかし，GC のバグ発見に応用した例は我々の知る限り存在しない．

Nishiwaki ら [11] は，JNI の参照の扱いに関するバグを発見するために，SEAN という C 言語のパタンマッチによるバグ発見ツールを開発した．SEAN は，構文木の特定のパタンにマッチする．中村ら [12] は，それを一般化して C 言語のソースコードからユーザが指定した構文木のパタンを探すツール ASTgrep を開発した．Quinlan ら [13] の Rose コンパイラインフラストラクチャを使った構文木のパタンマッチもバグ発見を目的にしている．

8. まとめ

本研究では，eJSVM に精確な GC を実装するために，ヒープへのポインタを持つ局所変数をルート集合に登録したり解除したりするコードのバグを探した．そのために，制御フローグラフに対するパタンマッチを行うツール Coccinelle を利用した．この方法の精確さを調べるために，開発者が追加していたルート集合への登録処理を全て取り除いたソースコードに対してこの方法を適用したところ，取り除いたうち必要な登録処理を全て見つけることができた．さらに，多くの登録処理の漏れを発見し，eJSVM のデバッグに役立った．また，他の研究プロジェクトが作ったソースコードに対しても同様の方法を適用し，いくつかのバグを発見した．偽陽性はほとんどなかったが，制御フローグラフに対するパタンマッチを行っているため，式の値に依存して実行経路が制限されるようなプログラムでは偽陽性があった．

参考文献

[1] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Computer Languages*, Vol. 51, pp. 261–279 (2019).

[2] Kataoka, T., Ugawa, T. and Iwasaki, H.: A Framework for Constructing JavasSript Virtual Machines with Customized Datatype Representations, *In Proc. SAC 2018*, ACM, pp. 1238–1247 (2018).

[3] Brunel, J., Doligez, D., Hansen, R. R., Lawall, J. L. and Muller, G.: A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking, *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, ACM, pp. 114–126 (2009).

[4] Padioleau, Y., Lawall, J., Hansen, R. R. and Muller, G.: Documenting and Automating Collateral Evolutions in Linux Device Drivers, *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*, ACM, pp. 247–260 (2008).

[5] Chambers, C., Ungar, D. and Lee, E.: An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes, *Proc. on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*, ACM, pp. 49–70 (1989).

[6] Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proceedings of the 3rd international symposium on Memory management (ISMM '02)*, ACM, pp. 150–156 (2002).

[7] Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software – Practice and Experience*, Vol. 18, No. 9, pp. 807–820 (1988).

[8] Lawall, J. L., Muller, G. and Palix, N.: Enforcing the Use of API Functions in Linux Code, *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACM, pp. 7–12 (2009).

[9] Palix, N., Thomas, G., Saha, S., Calvès, C., Muller, G. and Lawall, J.: Faults in Linux 2.6, *ACM Transactions on Computer Systems*, Vol. 32, No. 2, pp. 4:1–4:40 (2014).

[10] Lawall, J., Laurie, B., Hansen, R. R., Palix, N. and Muller, G.: Finding Error Handling Bugs in OpenSSL Using Coccinelle, *2010 European Dependable Computing Conference*, pp. 191–196 (2010).

[11] Nishiwaki, H., Ugawa, T., Umatani, S., Yasugi, M. and Yuasa, T.: SEAN: Support Tool for Detecting Rule Violations in JNI Coding, *IPSJ Transactions on Programming*, Vol. 5, No. 3, pp. 23–28 (2012).

[12] 中村真也，鵜川始陽，馬谷誠二：規則違反コードの構造を反映した木パタンを用いるコード検査器，情報処理学会論文誌プログラミング（PRO），Vol. 9, No. 4, pp. 1–15 (2016).

[13] Quinlan, D. J., Vuduc, R. W. and Misherghi, G.: Techniques for Specifying Bug Patterns, *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, ACM, pp. 27–35 (2007).

[*3] http://coccinelle.lip6.fr/rules/