# マルチスレッド環境下の参照カウントGCの為のカウンタ更新スレッド数の静的解析方法

有馬 大介* 今泉 貴史*

## 1 概要

プログラミング言語処理系の中には、動的に確保されたメモリ領域を自動的に回収する機能が用意されているものがある。この機能を**ガベージコレクション (GC)** という。GC により、プログラマは煩雑なメモリ管理から解放される。

**参照カウント GC** と呼ばれる GC では、各オブジェクト毎にカウンタを用意し、そこに被参照数を記録する。そしてカウンタが 0 になったオブジェクトを不要なデータ (**ガベージ**) として回収する。この GC 方式はガベージを即座に回収できるという利点がある。

しかしマルチスレッド環境において参照カウント GC を用いる場合、カウンタの増減にコストの高いアトミック操作等を用いる必要がある [1]。これは複数のスレッドが同時に 1 つのオブジェクトのカウンタを更新する可能性があるからである。しかし実際には複数のスレッドからカウンタを同時更新されないオブジェクトも存在する。それらのオブジェクトの参照カウンタに対してアトミック操作を用いるのは非効率的である。

本研究ではプログラムのソースコードを静的解析することにより、複数スレッドが同時にカウンタ更新する可能性がある (カウンタ更新にアトミック操作が必要な) オブジェクトを検出する方法を提案する。この検出を行う事により、カウンタ更新にアトミック操作が必要なオブジェクトとそうでないオブジェクトが区別できる。そしてアトミック操作によるカウンタ更新を前者に限定する事で、カウンタ操作のコストを軽減することができるようになる。今回は Java 言語を対象に解析を行う。

*千葉大学大学院融合科学研究科情報科学専攻

## 2 カウンタ操作スレッド数の解析方法の概要

プログラム 1 を例に挙げ、解析方法の概要について説明する。

```
1   class Main{
2     public static void main(String args[]){
3       Obj obj = new Obj();
4       MyThread mt = new MyThread(obj);
5       …
6       mt.start();
7       …
8     }
9   }
10
11  class MyThread extends Thread{
12    Obj f1;
13
14    MyThread(Obj x){
15      f1 = x;
16    }
17
18    public void run(){
19      …
20    }
21  }
```

プログラム 1: 簡単なプログラムの例

ここで main メソッドを親スレッド、main メソッド内で生成されたスレッド mt を子スレッドと呼ぶことにする。main メソッドでは 3 行目で Obj 型オブジェクトを生成し、4 行目でそのオブジェクトをスレッド mt に渡している。

6 行目の mt.start() でスレッド mt が実行され、7 行目以降の文は 2 つのスレッドが並行動作しながら実行される。この区間内でスレッド mt に渡したオブジェクト群が参照操作をされれば、それらのオブジェクト群

は 2 つ以上のスレッドからカウンタを更新される可能性がある (カウンタ更新にアトミック操作が必要)。逆に参照操作されていなければ、それらのオブジェクト群は単一のスレッドのみからカウンタを更新されることになる (カウンタ更新にアトミック操作が不必要)。この解析を行うためには、子スレッドに渡したオブジェクト群にアクセス可能な変数を求める事が必要となる。

以上の点から、今回の解析では次の 2 点に着目する。

- 子スレッドに渡したオブジェクト群にアクセス可能な変数
- 親スレッドと子スレッドが並列動作する区間

解析の手順としては以下の通りになる。

1. **制御フローグラフの作成**

   プログラムの処理の流れを調べる。ここで求めた制御フローグラフを別名解析、カウンタ操作スレッド数解析で利用する。

2. **別名解析**

   子スレッドに渡したオブジェクト群にアクセス可能な変数を調べるために行う

3. **カウンタ操作スレッド数解析**

   親スレッドと子スレッドが並列実行する区間内で、子スレッドに渡したオブジェクト群に参照操作が行われるかどうかを調べる。その結果を元に、各変数からアクセス可能なオブジェクトのカウンタ操作スレッド数が 1 か 2 以上かを判断する。

# 3 制御フローグラフの作成

プログラムの処理の流れを知るため、コードを基本ブロック単位に分解して制御フローグラフの作成を行う。

まず、エントリポイントとなるメソッドのコード全体を 1 つのブロックと考える。そしてこのブロックを先頭から解析し、ブロックを徐々に分割して基本ブロックに分解していく。

if 文や while 文のような制御構造に関しては、通常用いられる方法を使って制御フローを構築する。

解析中にメソッド呼び出しの文を見つけたら、そのメソッドのコードをそこに展開する。この時、後の解析が行いやすくするために、架空のコードの追加やプログラム文の書き換えを行う。

## 3.1 メソッド呼び出しの展開

メソッド $m$ を呼び出すブロックを次の 3 つに分けていく。

- メソッド $m$ 実行前のコード
- メソッド $m$ の展開コード
- メソッド $m$ 完了後のコード

メソッド $m$ の展開コードに関しては更に次の 4 つに分けていく。

- 引数への代入
- メソッド $m$ のコード
- メソッド $m$ の終了
- 返り値の代入

$n$ 個の文 $s_1, s_2, \cdots, s_n$ を持つブロック $b$ において、$s_k$ がメソッド $m(a1, a2, \cdots, an)$ の呼び出しを行っていたとする。この時、ブロック $b$ を次の 6 つのブロック $b_1, b_2, b_3, b_4, b_5, b_6$ に分解する。

**$b_1$ メソッド $m$ 開始前のブロック**

$s_1, s_2, \cdots, s_{k-1}$ からなるブロック

**$b_2$ メソッド $m$ の開始点**

メソッド $m$ の引数 $a1, a2, \cdots, an$ への代入文からなるブロック。また、メソッド m を呼び出した変数を、架空の変数 $this_m$ に代入する文を追加する。例えば文 $s_k$ が $v.m(x1, x2, \cdots, xn)$ のようなメソッド呼び出し文の時、次のようなブロックを作成する。

$$
\begin{array}{|l}
\hline
this_m = v \\
a1 = x1 \\
a2 = x2 \\
\cdots \\
an = xn \\
\hline
\end{array}
$$

**$b_3$ メソッド $m$ のコードのブロック**

メソッド $m$ のコードを持つブロック。コード内にあるフィールド $f$ は全て $this_m.f$ に置き換える。また $return$ 文を次の 2 文に置き換える。

- 架空の変数 $return_m$ に返り値を代入する文
- ブロック $b_4$ へ移動する文

**$b_4$ メソッド $m$ の終了点**

空のブロック。

**$b_5$ 返り値の代入**

文 $s_k$ が $v = u.m(x1, x2, \cdots, xn)$ のように返り値を変数 $v$ に代入する場合、$b_5$ は文 $v = return_m$ を持つ。

**$b_6$ メソッド $m$ 以降の文のブロック**

$s_{k+1}, s_{k+2}, \cdots, s_n$ からなるブロック

この 6 つのブロックに対し、エッジ $b_1 \rightarrow b_2, b_2 \rightarrow b_3, b_3 \rightarrow b_4, b_4 \rightarrow b_5, b_5 \rightarrow b_6$ を追加する。これらのブロックはまだ基本ブロックになっているとは限らないので、これらのブロックの中を解析してさらに展開・分割を行っていく。

プログラム 2 の main メソッドを例に挙げ、メソッド呼び出しの展開の例を示す (図 1)。

```
1   class C{
2     Obj f1;
3     Obj f2;
4
5     Obj method(Obj a1, Obj a2){
6       Obj temp = f1;
7       f1 = a1;
8       f2 = a2;
9       return(temp);
10    }
11  }
12
13  class Main{
14    public static void main(String args[]){
15      C c_obj;
16      Obj result;
17      … // コード群 1
18      result = c_obj.method(x1, x2);
19      … // コード群 2
20    }
21  }
```

プログラム 2: メソッド呼び出しを行うプログラム

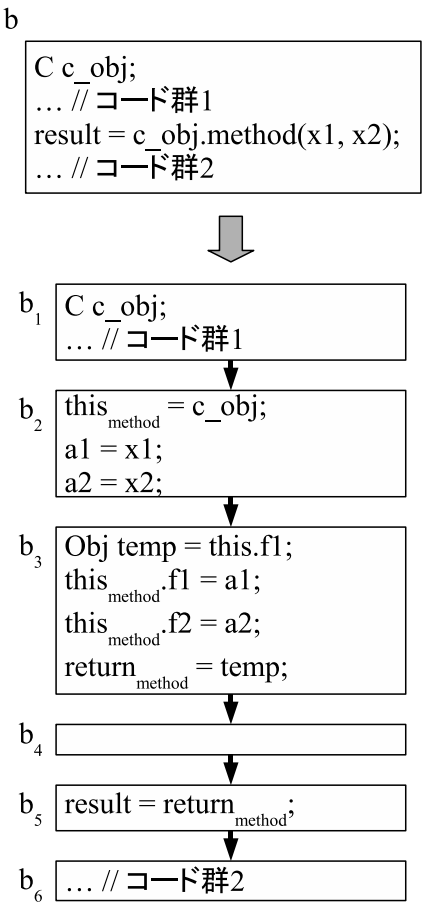


図 1: プログラム 2 メソッド呼び出しの展開

## 3.2 コンストラクタの展開

クラス $C$ のコンストラクタが $C(a1, a2, \cdots)$ の形で定義されているとする。

ブロック内に $v = new\ C(x1, x2, \cdots)$ がある時、このコンストラクタ実行をメソッド呼び出しと同じように展開する。その際、ブロック $b_2$ と $b_5$ に関しては次の変更を加える。

**$b_2$ コンストラクタの開始点**

変数 $v$ に新たなオブジェクトが割り当てられる事を示すため、架空のコード $v = MALLOC()$ を追加する。また、メソッド呼び出しの時と同様に $this_C = v$ と $a1 = x1, a2 = x2, \cdots$ の文を持たせる。

**$b_5$ 返り値の代入**

架空のコード $v = this_C$ の文を持つ。

### 3.3 メソッド $m$ の再帰呼び出しの展開

メソッド $m$ のコード内 (ブロック $b_3$) に同一のメソッド $m$ の呼び出しがある場合について述べる。これを先ほどと同じ方法で展開してしまうと、$b_3$ のブロックが無限に増え続けてしまう。

このような事態を避けるため、再帰呼び出しメソッドを展開する際は新たに $b_3$ ブロックを作らずに呼び出し元のメソッドの $b_3$ ブロックを共有させる事にする。また、呼び出し元のメソッドと再帰呼び出しのメソッドとでは、呼び出し時の引数や返り値を受け取る変数等が異なる。そこで引数の設定を行う $b_2$ ブロックと返り値を返す $b_5$ ブロックに関しては、呼び出し元メソッドと再帰呼び出しメソッドとで共有させない。再帰呼び出しメソッド用の $b_2, b_5$ ブロックを用意する。

再帰呼び出しがあるブロックを $b'$ とし、$b'$ は文 $s'_1, s'_2, \cdots, s'_m$ を持つとする。文 $s'_j$ においてメソッド $m$ の再帰呼び出しがある時、ブロック $b'$ を次のように展開する。

**$b'_1$ メソッド $m$ 開始前のブロック**

$s'_1, s'_2, \cdots, s'_{j-1}$ からなるブロック

**$b'_2$ メソッド $m$ の開始点**

メソッド $m$ の引数 $a1, a2, \cdots, an$ への代入文からなるブロック。また、メソッド $m$ を呼び出した変数を、架空の変数 $this_m$ に代入する文を追加する。

**$b'_5$ 返り値の代入**

文 $s'_j$ が $v = u.m(x1, x2, \cdots, xn);$ のように返り値を変数 $v$ に代入する場合、$b_5$ は文 $v = return_m;$ を持つ。

**$b'_6$ メソッド m 以降の文のブロック**

$s'_{j+1}, s'_{j+2}, \cdots, s'_m$ からなるブロック

最後にエッジ $b'_1 \rightarrow b'_2, b'_1 \rightarrow b'_6, b'_2 \rightarrow b_3, b_4 \rightarrow b'_5, b'_5 \rightarrow b'_6$ を追加する。

### 3.4 スレッド開始文の展開

スレッド $t$ の実行開始となる文 $t.start()$ に関しては、$t.run()$ のメソッド呼び出しと解釈して展開する。後のカウンタ操作スレッド数解析で並列実行が開始される点を把握できるようにするため、ブロック $b_1$ の末尾に架空の命令文 $THREADSTART(t)$ を追加する。

この時、親スレッドのコードであるブロック $b_6$ と、子スレッド t のコードであるブロック $b_2, b_3, b_4, b_5$ が並列に動作する事になる。そこでエッジ $b_1 \rightarrow b_2, b_2 \rightarrow b_3, b_3 \rightarrow b_4, b_4 \rightarrow b_5, b_1 \rightarrow b_6$ を追加する。

スレッド開始文の展開の様子を図 2 に示す。

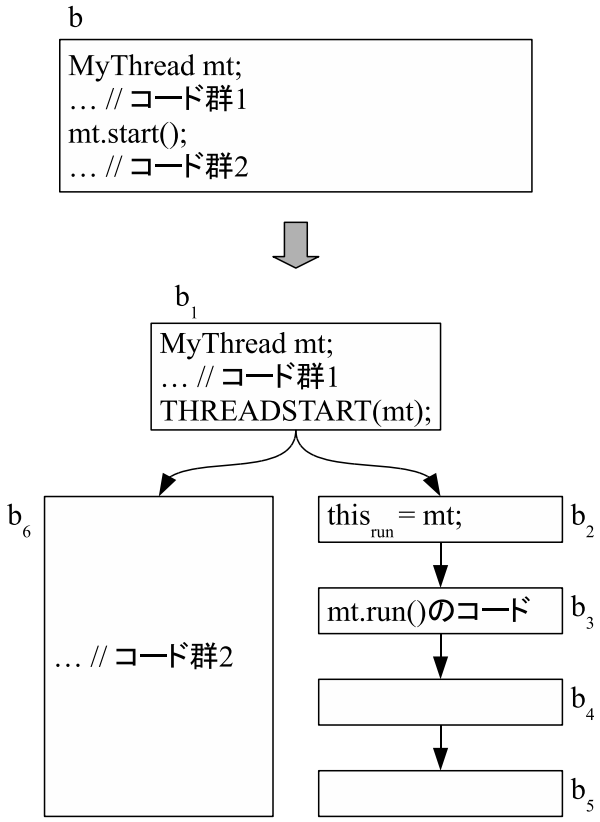


図 2: スレッド開始文の展開

## 4 別名解析

制御フローグラフを元に別名解析を行う。今回はフロー依存の解析方法 [2] と Connection analysis [3] を組み合わせて別名解析を行う。

### 4.1 変数の別名

変数 $a$ からアクセス可能なオブジェクト群と変数 $b$ からアクセス可能なオブジェクト群の中に共通するオブジェクトがある時、変数 $a$ と $b$ は別名関係があるとする [3]。この別名関係の集合を $A$ とおく。変数 $a$ と変数 $b$ が別名関係にあるなら $A$ は要素 (a, b) を持つものとする。また変数 $a$ が何らかのオブジェクトを指している場合は要素 $(a, a)$ も $A$ に含まれるとし、変数 $a$ が何のオブジェクトも指していない場合は要素 $(a, a)$ は $A$ に含まれないものとする。

別名解析の対象とする参照型変数・フィールドについて述べる。[2, 3] の解析方法では変数間の別名関係を求めており、変数が指すオブジェクトのフィールドの別名関係は求めていない。しかし今回はスレッドのフィールドが指すオブジェクト群が他スレッドで用いられるかを知りたいので、これらのフィールドの別名関係を求める必要がある。よって今回は次の変数・フィールドを解析対象とする。

- Thread クラスを継承していないクラスのオブジェクトを参照する変数
- Thread クラスを継承しているクラスのフィールド

これらの変数・フィールドをギリシャ文字 $\alpha, \beta, \gamma, \delta, \cdots$ で表す事とし、これらの変数・フィールドの集合を $V$ とおくことにする。

### 4.2 フロー依存の解析方法

[2] のフロー依存の別名解析方法を用い、制御フローグラフの各基本ブロック b に対して次の 2 つの別名関係の集合 $A_{IN(b)}, A_{OUT(b)}$ を反復を用いて求める。

- $A_{IN(b)}$

  ブロック b の入口で成り立っている別名関係の集合
- $A_{OUT(b)}$

  ブロック b の出口で成り立っている別名関係の集合

これらの集合が求まれば、文 s 実行後に成り立つ別名関係の集合 $A_s$ が定まる。

$A_{IN(b)}$ の計算方法は次のようになる。

$$A_{IN(b)} = \bigcup_{P \in pred(b)} A_{OUT(P)} \tag{1}$$

次に $A_{OUT(b)}$ の計算方法について述べる。$A_{OUT(b)}$ は次の 2 つを元に算出する。

- $A_{IN(b)}$
- ブロック b 内の文 $s_1, s_2, \cdots, s_n$

ブロック内の文を実行する度に別名関係は変化していく。文 s による別名関係の変化を関数 $T_s$ を用いて表すこととする ($T_s$ の計算規則に関しては [3] の計算規則を利用する)。$A_{OUT(b)}$ はこの関数を用いて、次のように算出される。

$$A_{OUT(b)} = T_{s_n} \circ \cdots \circ T_{s_2} \circ T_{s_1}(A_{IN(b)}) \tag{2}$$

解析の初期状態では全てのブロック $b$ に対して $A_{IN(b)} = A_{OUT(b)} = \phi$ とする。そしてエントリポイントのブロックから解析を始め、エッジに沿ってブロックを辿りながら別名関係を求めていく。

### 4.3 $T_s$ の計算規則

$T_s$ の計算規則について述べる。[3] では、別名関係 A が与えられた状態で文 $s$ が実行した際に得られる別名関係 $T_s(A)$ を次の計算規則で求めている。

- **文 s が $\alpha = MALLOC()$ の場合**

  $\alpha$ は以前指していたオブジェクト群を指さなくなり、新たなオブジェクトのみを指す。よって

  $$T_s(A) \quad = \quad A - \{(\alpha, \gamma) | \gamma \in V\} \cup \{(\alpha, \alpha)\}$$

  となる。
- **文 s が $\alpha = null$ の場合**

  $\alpha$ は以前指していたオブジェクト群を指さなくなる。よって

  $$T_s(A) \quad = \quad A - \{(\alpha, \gamma) | \gamma \in V\}$$

  となる。

- **文 s が $\alpha = \beta$, $\alpha = \beta.f$, $\alpha = \beta[i]$ の場合**

  $\alpha$ は以前指していたオブジェクト群を指さなくなり、$\beta$ と同じオブジェクト群を指す。よって

$$\begin{aligned} T_s(A) &= (A - \{(\alpha, \gamma) | \gamma \in V\}) \\ &\quad \cup \{(\alpha, \delta) | (\beta, \delta) \in A\} \cup (\alpha, \alpha) \end{aligned}$$

  となる。

- **文 s が $\alpha.f = \beta$ の場合**

  $\alpha$ は以前指していたオブジェクト群を指したまま、$\beta$ と同じオブジェクト群を指す可能性がある。また、同様に $\beta$ も以前指していたオブジェクト群を指したまま、$\alpha$ と同じオブジェクト群を指す可能性がある。よって

$$T_s(A) = A \cup \{(\gamma, \delta) | (\alpha, \gamma) \in A, (\beta, \delta) \in A\}$$

  となる。

今回の解析ではこれらの計算規則に次の計算規則を加え、別名関係を求める。変数 $t1, t2$ はスレッドを指す変数とする。

- **文 s が $\alpha = \beta.f1.f2.\cdots$ の場合**

  文 $s$ が $\alpha = \beta.f$ の場合と同様に

$$\begin{aligned} T_s(A) &= (A - \{(\alpha, \gamma) | \gamma \in V\}) \\ &\quad \cup \{(\alpha, \delta) | (\beta, \delta) \in A\} \cup (\alpha, \alpha) \end{aligned}$$

  とする。

- **文 s が $\alpha.f1.f2.\cdots = \beta$, $\alpha.f1.f2.\cdots = \beta.f'1.f'2.\cdots$, $\alpha.f1.f2.\cdots = \beta[i]$ の場合**

  文 $s$ が $\alpha.f = \beta$ の場合と同様に考え、

$$T_s(A) = A \cup \{(\gamma, \delta) | (\alpha, \gamma) \in A, (\beta, \delta) \in A\}$$

  とする。

- **文 s が $\alpha[i] = \beta$, $\alpha[i] = \beta.f1.f2.\cdots$, $\alpha[i] = \beta[j]$ の場合**

  文 $s$ が $\alpha.f = \beta$ の場合と同様に考え、

$$T_s(A) = A \cup \{(\gamma, \delta) | (\alpha, \gamma) \in A, (\beta, \delta) \in A\}$$

  とする。

- **文 $s$ が $t1 = MALLOC()$ の場合**

  変数 $t1$ が指すスレッドの全てのフィールド $f$ に新しい領域が割り当てられると考え、

$$\begin{aligned} T_s(A) &= (A - \{(t1.f, \gamma) | t1.f \in V, \gamma \in V\}) \\ &\quad \cup \{(t1.f, t1.f) | t1.f \in V\} \end{aligned}$$

  とする。

- **文 s が $t1 = null$ の場合**

  変数 $t1$ が指すスレッドの全てのフィールド $f$ に $null$ が代入されたと考え、

$$T_s(A) = A - \{(t1.f, \gamma) | t1.f \in V, \gamma \in V\}$$

  とする。

- **文 s が $t1 = t2$ の場合**

  変数 $t1$ が指すスレッドのフィールド f に $t2.f$ が代入されると考え、

$$\begin{aligned} T_s(A) &= (A - \{(t1.f, \gamma) | t1.f \in V, \gamma \in V\}) \\ &\quad \cup \{(t1.f, \gamma) | t1.f \in V, (\gamma, t2.f) \in A\} \\ &\quad \cup \{(t1.f, t1.f) | t1.f \in V\} \end{aligned}$$

  とする。

文 $s$ が前述の全ての場合に当てはまらない場合は $T_s(A) = A$ とする。

# 5 カウンタ操作スレッド数解析

別名解析で解析対象とした変数・フィールドに対し、2 つ以上のスレッドからカウンタ操作されるかどうかの真偽を調べていく。変数 $\alpha$ からアクセス可能なオブジェクト群の中に、2 つ以上のスレッドからカウンタ操作されるオブジェクトが存在するか否かの真偽値を $MTOT(\alpha)$ と表す事にする。カウンタ操作スレッド数解析では各変数に対してこの $MTOT$ 値を求めていく。

別名解析と同様に反復を用い、各ブロックの入口と出口における各変数の $MTOT$ 値を求める。ブロック入口における $MTOT$ 値が決定すれば、ブロック内の $MTOT$ 値も定まることになる。

ブロック b の入口における $MTOT_{IN(b)}(\alpha)$ の計算方法を次の通りに定める。

$$MTOT_{IN(b)}(\alpha) = \bigcup_{p \in pred(b)} MTOT_{OUT(p)}(\alpha) \quad (3)$$

ブロック b の出口における $MTOT$ の算出方法に関しては別名解析と同様の方法を取る。ブロック入口における各変数の $MTOT$ の値と、ブロック内の文によって算出を行う。解析の初期状態では、全ての変数・フィールド $\alpha$ に対して $MTOT_{IN(b)}(\alpha) = false$ とする。エントリポイントのブロックから解析を始め、エッジに沿ってブロックを辿りながら各変数の $MTOT$ 値を算出していく。

ブロック内の k 番目の文 $s_k$ を実行した後の変数・フィールド $\alpha$ の $MTOT$ 値を $MTOT_{s_k}(\alpha)$ とおく。この時、$MTOT_{s_k}(\alpha)$ の計算規則を以下のように定める。

- **文 $s_k$ が $\alpha = MALLOC()$ の場合**

  $\alpha$ は文 $s_{k-1}$ の時点で指していたオブジェクト群を指さなくなる。

  - $\alpha$ がローカル変数なら、新しく確保されたオブジェクトは文 $s_k$ 実行時、単一スレッドのみからアクセス可能である。よってこの場合、

    $$MTOT_{s_k}(\alpha) \quad = \quad false$$

    とする。
  - $\alpha$ がスレッドのフィールドの場合、文 $s_{k-1}$ の時点でフィールド $\alpha$ 自体が 2 つ以上のスレッドから利用されているなら、新しく確保されたオブジェクトもまた 2 つ以上のスレッドから利用される可能性がある。逆に文 $s_{k-1}$ の時点でフィールド $\alpha$ 自体が 2 つ以上のスレッドから利用されていないなら、新しく確保されたオブジェクトは単一のスレッドのみから利用される。よってこの場合、

    $$MTOT_{s_k}(\alpha) \quad = \quad MTOT_{s_{k-1}}(\alpha)$$

    とする。

  $\alpha$ 以外の変数・フィールド $\beta$ に関しては

  $$MTOT_{s_k}(\beta) \quad = \quad MTOT_{s_{k-1}}(\beta)$$

  とする。

- **文 $s_k$ が $\alpha = null$ の場合**

  文 $s_k$ が $\alpha = MALLOC()$ の場合と同様に、$\alpha$ は文 $s_{k-1}$ の時点で指していたオブジェクト群を指さなくなる。よってこの場合、文 $s_k$ が $\alpha = MALLOC()$ の時と同じ算出方法を用いる。

- **文 $s_k$ が $\alpha = \beta$, $\alpha = \beta.f1.f2.\cdots$, $\alpha = \beta[i]$ の場合**

  $\alpha$ は以前指していたオブジェクト群を指さなくなり、$\beta$ と同一のオブジェクト群を指す事になる。

  - $\alpha$ がローカル変数なら、

    $$MTOT_{s_k}(\alpha) \quad = \quad MTOT_{s_{k-1}}(\beta)$$

    とする。
  - $\alpha$ がスレッドのフィールドの場合、$\alpha$ 自体が 2 つ以上のスレッドから利用可能である可能性がある。この時、$\beta$ や $\beta$ と別名関係にある変数が指すオブジェクト群も複数のスレッドから操作されるようになる可能性がある。よってこの場合、$(\alpha, \gamma) \in A_{s_k}$ を満たす変数 $\gamma$ に対し、

    $$MTOT_{s_k}(\gamma) \quad = \quad MTOT_{s_{k-1}}(\alpha) \vee MTOT_{s_{k-1}}(\beta)$$

    とする。

  MTOT 値が変化しなかった他の変数・フィールド $\delta$ に関しては

  $$MTOT_{s_k}(\delta) \quad = \quad MTOT_{s_{k-1}}(\delta)$$

  とする。

- **文 $s_k$ が $\alpha.f1.f2.\cdots = \beta$, $\alpha.f1.f2.\cdots = \beta.f'1.f'2.\cdots$, $\alpha.f1.f2.\cdots = \beta[i]$ の場合**

  $\alpha$ と $\beta$、そしてそれらと別名関係にある変数、フィールドが同一オブジェクト群を指す可能性がある。$MTOT_{s_{k-1}}(\alpha)$ または $MTOT_{s_{k-1}}(\beta)$ が true なら、これらの変数、フィールドはカウンタ操作スレッド数が 2 以上のオブジェクトを指す可能性がある。よって $(\alpha, \gamma) \in A_s$ を満たす変数 $\gamma$ に対し、

  $$MTOT_{s_k}(\gamma) \quad = \quad MTOT_{s_{k-1}}(\alpha) \vee MTOT_{s_{k-1}}(\beta)$$

  とする。

MTOT 値が変化しなかった他の変数・フィールド $\delta$ に関しては

$$MTOT_{s_k}(\delta) \quad = \quad MTOT_{s_{k-1}}(\delta)$$

とする。

- **文 $s_k$ が $\alpha[i] = \beta$, $\alpha[i] = \beta.f1.f2.\cdots$, $\alpha[i] = \beta[j]$ の場合**

  文 $s_k$ が $\alpha.f1.f2.\cdots = \beta$ の場合と同様に考え、同じ計算規則を用いる。

- **文 $s_k$ が $THREADSTART(t)$ の場合**

  親スレッドが子スレッドに渡したオブジェク群のうち、親スレッドと子スレッドが同時にカウンタ操作する可能性があるものを調べる。そしてそれらのオブジェクトにアクセス可能な変数の $MTOT$ 値を $true$ に設定していく。

  文 $s_k$ を含む基本ブロックの後続ブロックのうち、親スレッドの動作を示すブロックを $b_p$ とおく。まず $b_p$ と $b_p$ から到達可能な基本ブロックからなる集合 $B_p$ を求める。この集合は親スレッドの動作を表すブロック群である。次にスレッド $t$ の各フィールド $t.f1, t.f2, \cdots$ に対し、次の操作を行う。

  - **$MTOT$ 値が false のフィールド $t.f_f$ の場合**
    文 $s_k$ の時点で $t.f_f$ と別名関係にある変数が $B_p$ 内で参照操作をされているかを調べる。参照操作されていた場合、$t.f_f$ が指すオブジェクト群は複数のスレッドからカウンタ操作されることになる。よってこの時、$(t.f_f, \gamma) \in A_{s_k}$ を満たす変数 $\gamma$ に対して

    $$MTOT_{s_k}(\gamma) \quad = \quad true$$

    とする。参照操作されていなかった場合、

    $$MTOT_{s_k}(t.f_f) \quad = \quad false$$

    とする。
  - **$MTOT$ 値が true のフィールド $t.f_t$ の場合**
    $t.f_t$ が指すオブジェクト群は既に複数スレッドからカウント操作される事が分かっているので

    $$MTOT_{s_k}(t.f_t) \quad = \quad true$$

    とする。

MTOT 値が変化しなかった他の変数・フィールド $\delta$ に関しては

$$MTOT_{s_k}(\delta) \quad = \quad MTOT_{s_{k-1}}(\delta)$$

とする。

- **文 s が上記以外の文の場合**

  全ての変数・フィールド $\alpha$ に対し、

  $$MTOT_{s_k}(\alpha) \quad = \quad MTOT_{s_{k-1}}(\alpha)$$

  とする。

### 5.1 参照カウンタ更新時のアトミック操作の必要性の判定

文 $s_k$ における $MTOT$ 値が $true$ である変数を $\alpha_t$、$false$ である変数を $\alpha_f$ とおく。$\alpha_t$ からアクセス可能なオブジェクト群は複数のスレッドから参照カウンタを更新される可能性がある。逆に $\alpha_f$ からアクセス可能なオブジェクト群は単一のスレッドのみから参照カウンタを更新される。よって文 $s_k$ においてオブジェクトの参照カウンタを更新する際、次のように考える事ができる。

- $\alpha_t$ からアクセス可能なオブジェクト群に関してはアトミック操作が必要である
- $\alpha_f$ からアクセス可能なオブジェクト群に関してはアトミック操作は必要無い

## 6 考察

今回の解析は複数のスレッドが同時にカウンタ操作を行うかどうかを求め、それを元にカウンタ操作にアトミック操作が必要かどうかを判断している。そのため、各スレッドがそれぞれ独立した処理を行うようなプログラムにおいては、多くのオブジェクトのカウンタ操作にアトミック操作は必要無いと判定できると考えられる。ここで明らかに共通のオブジェクトを持たないような、次の 2 つの条件を持つスレッドについて考える。

- フィールドを持たない

- 別のスレッドを生成・実行しない

今回の解析方法で初めに $MTOT$ 値が $true$ となる変数・フィールドは、子スレッドのフィールドと、そのフィールドと別名関係にある変数である。そしてそれらのフィールド・変数が別の変数・フィールドに代入される事で、$MTOT$ 値が $true$ の変数が増える。フィールドを持たず、別スレッドを生成しないスレッドに関しては、最初に $MTOT$ 値が $true$ となる変数・フィールドが存在しない。また、以降の文の実行によって変数の $MTOT$ 値が $true$ になることもない。よって前述の 2 つの条件を持つスレッドに関しては、カウンタ更新にアトミック操作が全く必要無いと確実に判定できると考えられる。

次に親スレッドと子スレッドの並列実行箇所について考える。今回の解析方法ではカウンタ更新スレッドが複数存在するオブジェクトを求めるため、親スレッドが子スレッドと並列実行する箇所を解析している。しかし今回は保守的に、親スレッドが子スレッドを開始する文以降全てを並列実行箇所と見なしている。その為、親スレッドが早く子スレッドの実行開始を行ってしまうと並列実行箇所と判断されるブロックが増える。このように並列実行箇所が増えてしまうと、カウンタ更新スレッド数が複数だと判定される変数が増えてしまう。カウンタ更新スレッドが複数かどうかをより正確に判定するためには、より厳密に並列実行箇所を求める必要があると考えられる。Java 言語では並列実行中の子スレッドの完了を待つ $join()$ メソッドが用意されている。このメソッドを考慮する事により、親スレッドと子スレッドの並列実行が完了するタイミングを把握することができれば、解析の精度を上げる事ができると考えられる。

# 7 まとめ

複数スレッドから同時に参照カウントの値を更新をされる可能性があるオブジェクトとそうでないオブジェクトを区別する方法を提案した。この解析結果から、参照カウンタ更新の際の不必要なアトミック操作を減らす事が可能となる。

今回は親スレッドと子スレッドの並列実行箇所の求め方が保守的であった。そこで今後はより正確に並列実行箇所を求める方法について考えていき、さらに解析の精度を上げていくことを目指す。

# 参考文献


[1] Richard Jones and Rafael D. Lins. Garbage Collection. WILEY, 1996.

[2] 中田育男. コンパイラの構成と最適化. 朝倉書店, 第 2 版, 2009.

[3] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: a practical interprocedural heap analysis for c. Int. J. Parallel Program., Vol. 24, No. 6, pp. 547–578, 1996.