Continuation based C を用いたプログラムの検証手法

比嘉 健太[1,a)]　河野 真治[2,b)]

概要：Continuation based C 言語によって記述されたプログラムのデータ構造の性質を検証する手法を提案する。Continuation based C とは当研究室が提案している Code Segment, Data Segment という単位でプログラムを記述する言語である。Code Segment とは処理の単位であり、データの単位である Data Segment を入力と出力に持つ。プログラム全体は Code Segment どうしの接続により表現され、ある Code Segment の出力は接続された次の Code Segment の入力となる。また、メモリ管理やエラーの処理など、本来中心に行ないたい処理と異なる処理はメタ計算として分離し、Meta Code Segment, Meta Data Segment として記述する。Code Segment の接続処理を Meta Code Segment として表現し、接続部分に検証を含めることで 元の Code Segment を変更することなくプログラムの検証を行なう。本論文では Continuatoin based C によって記述された赤黒木といったデータ構造の性質を検証する。

キーワード：プログラミング言語, 検証, 赤黒木

Verification method of programs using Continuation based C

Yasutaka HIGA[1,a)]　Shinji KONO[2,b)]

Abstract: We propose a verification method for executable programs using Continuation based C language. Our laboratory develops Continuation based C language which supports programming unit called Code Segment, Data Segment. Code segments are calculation units which have input/output data segments that data unit. Programs are represented by connections among with code segments and code segments. The output data segment of some code segment is converted to the input data segment of connected one. We introduce meta computations which split main computations and complicated computations such as memory control, error handling and more. Meta computations represented to meta code segment and meta data segment, which saves main computations. In this paper, We define a meta computation which connects code segments with verifications and verify properties of data structures such as Red-Black Tree.

Keywords: Programming Language, Verification, Red-Black Tree

1. ソフトウェアの検証

ソフトウェアの規模が大きくなるにつれ、バグは発生しやすくなる。バグとはソフトウェアが期待される動作とは異なる動作を行なうことである。また、ソフトウェアの期待される動作を定義したものは仕様と呼ばれ、自然言語や論理で記述される。検証とは、ソフトウェアが定められた環境下において仕様を満たすことを保証するものである。ソフトウェアが仕様を満たすことを保証する手法として、定理証明とモデル検査がある。定理証明はプログラムが満たすべき仕様を論理式で記述し、その式が常に真であることを証明する。定理証明を行なう言語には Agda[4] や Coq? などが存在する。モデル検査を行なうモデル検査器には、 promela と呼ばれる言語でモデルを記述する spin? やモデルを状態遷移系で記述する NuSMV? 、C 言語を記号実行することができる CBMC[3] などが存在する。証明やモデル検査器で検証を行なう際、実際に動作するコードでなく証明用にコードを書き直す必要があるなど、実際の実装との乖離が存在する。よって、検証した環境では仕様

[1] 琉球大学 大学院 理工学研究科 情報工学専攻
Infomation Engineering Course Graduate School of Engineering and Science University of the Ryukyus
[2] 琉球大学
University of the Ryukyus
[a)] atton@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

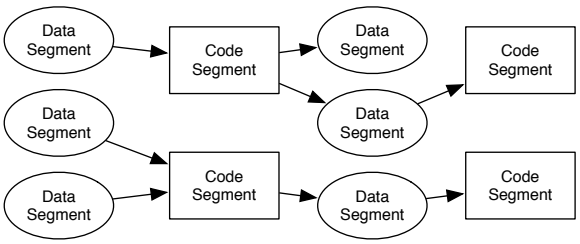


図 1 Code Segment どうしの組み合わせ

が満たされていても、実装にバグが入り込み信頼性を保証できない可能性がある。本研究は実際に動作するプログラムの信頼性を保証することを目的とする。

2. Code Segment と Data Segment

動作するコードを検証しやすいよう、本研究室では Code Segment と Data Segment を用いるプログラミングスタイルを提案している。Code Segment とは処理の単位であり、ループを含まない単純な処理のみを行なう。プログラムは Code Segment どうしを組み合わせることで構築される (図 1)。Code Segment 間における値の受け渡しには、Data Segment というデータの単位で行なう。なお、接続された Code Segment には依存関係が発生するが、依存関係が無い Code Segment は並列に実行することが可能である。

ここで、Code Segment どうしの接続処理について考える。処理を表す Code Segment どうしの接続も処理であるため、Code Segment で表現できる。このような、計算を実現するための計算をメタ計算と呼び、メタ計算を行なう Code Segment を Meta Code Segment と呼ぶ。Meta Code Segment は Code Segment 間に存在する上位の処理と考えることもできる (図 2)。

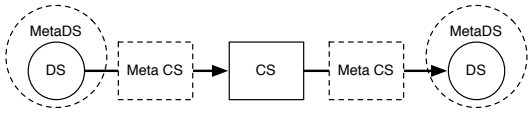


図 2 Meta Code Segment と Meta Data Segment

また、メタ計算に必要なデータは Meta Data Segment に格納し、通常の処理に必要な Data Segment も内包する。プログラムの性質を検証する機能をメタ計算として提供することで、ユーザが書いた Code Segment を変更することなく、メタ計算を追加するだけでプログラムの信頼性を上げる。

```
__code addTen(int a) {
    int b = a+10;
    goto twice(b);
}

__code twice(int x) {
    int y = 2*x;
    goto showValue(y);
}
```

表 1 Code Segment の接続 (10 加算して 2 倍する)

3. Continuation based C

Code Segment と Data Segment を用いたプログラミングスタイルで記述言語に Continuation based C[1] 言語が存在する。Continuation based C (以下 CbC) は OS や組込みシステムなどの記述を行なうことを目標に作られた、アセンブラと C 言語の中間のような言語である。CbC における Code Segment は C 言語における関数に、関数呼び出しが末尾のみである制約を加えようなものである。Code Segment どうしの接続は goto による軽量継続で表される (表 1)。軽量継続とは呼び出し元の環境を持たずに次の処理へと移動することであり、呼び出し元のスタックフレームを破棄しながら関数呼び出しを行なうようなものである。なお、C 言語の資産を利用するために通常の関数呼び出しを行なうことも可能である。

CbC における Code Segment は、C 言語の関数宣言の返り値の型の部分に `__code` を記述して定義する。Code Segment 内部には変数の宣言や if 文による分岐といった C 言語の文法を用いて処理を記述する。表 1 の例では、2 つの Code Segment `addTen` と `twice` を定義している。`addTen` では int の値を受けとり、10 加算して `twice` を実行する。`twice` では受けとった int の値を 2 倍して `showValue` を実行する。

また、CbC における Data Segment は C 言語における構造体と共用体を用いたデータ構造である。各 Code Segment で必要な値を構造体で定義し、それらの共用体として Data Segment を定義する (表 2)。

```
union Data {
    struct Count {
        int x;
    } count;
    struct Port {
        unsigned int port;
    } port;
};
```

表 2 Data Segment の例

表 2 では Data Segment として int を持つ count と un-

signed int を持つ port の 2 つを定義している。Code Segment 内部では演算やポインタ演算は行なわず、メタ計算部分でポインタへの演算を行なう。これにはメモリ管理をメタ計算部分に分離することで、プログラムを検証しやすくするねらいがある。Code Segment がどの Data Segment にアクセスするかといった指定も Meta Code Segment で行なう (表 3 における `twice_stub`)。CbC における Meta Code Segment は Code Segment と Code Segment 間に存在する Code Segment である。

```
// Code Segment
__code addTen(union Data* ds, int a) {
    int b = a+10;
    goto twice_stub(ds);
}

// Meta Code Segment
__code twice_stub(union Data* ds) {
    goto twice(ds->count.x);
}

// Code Segment
__code twice(int x) {
    int y = 2*x;
    goto showValue(y);
}
```

表 3 Data Segment を指定する Meta Code Segment

CbC における Meta Data Segment は Data Segment を内包する構造体として定義できる (表 4 における Context)。また、goto する際に必ず通る Meta Code Segment を定義することで、 Code Segment どうしの接続もメタ計算として定義できる (表 4 における `meta`)。`meta` を切り替えることで Code Segment を変更することなくメタ計算のみを変更することができる。

メタ計算の例として、メモリ管理の他にも例外処理や並列実行やネットワークアクセスなどがある。通常の計算を保存するようメタ計算を定義することで、例外処理などを含む計算に拡張することができる [5]。

4. CbC で記述した赤黒木

CbC で記述されたデータ構造に赤黒木がある。赤黒木とは木構造のデータ構造であり、各ノードに赤と黒の色を割り当て、その色を用いて木の高さのバランスを取る。赤黒木は通常の二分探索木としての条件の他に以下の条件を持つ。

- 各ノードは赤または黒の色を持つ。
- ルートの色は黒である。
- 赤ノードは 2 つの黒ノードを子として持つ (よって赤ノードが続くことはない)。

```
// Meta Data Segment
struct Context {
    union Data *data; // Data Segment
    unsigned int a; /* メタ計算に必要なデータ*/
};

// Meta Code Segment
__code meta(struct Context* context,
            enum Code next) {

    /* 接続時に行なうメタ計算を記述*/
    switch (next) {
        case AddTen:
            /* 特定の Code へのメタ計算 Segment */
            goto addTen_stub(context);
        case Twice:
            goto twice_stub(context);
    }
}

// Code Segment
__code addTen(struct Context* context, int a) {
    x = x+10;
    goto meta(context, Twice);
}

// Code Segment
__code twice(struct Context* context, int x) {
    x = x*2;
    goto meta(context, ShowValue);
}
```

表 4 Code Segment を接続するメタ計算メタ計算として接続した例

- ルートから最下位ノードへの経路に含まれる黒ノードの数はどの最下位ノードでも一定である。

これらの条件により、木をルートから辿った際に最も長い経路は最も短い経路の高々 2 倍に収まる。

赤黒木の実装として、当研究室で CbC を用いて開発している Gears OS [2] における非破壊赤黒木がある。非破壊赤黒木とは木への挿入や削除を行なった際に一度構築した木構造を破壊することなく新しく木構造を生成する赤黒木である。非破壊赤黒木の実装の基本的な戦略は、変更したいノードへのルートノードからの経路を全て複製し、上に存在しないノードは変更前元の木構造と共有することで実現できる (図 3)。

関数呼び出しが可能な言語では戻り値で経路を辿ることか可能だが、CbC は goto による軽量継続のみで記述する必要があるため、経路を辿るにはノードに親への参照を持たせるか挿入・削除時に辿った経路を記憶するしかない。ノードが親への参照を持つ非破壊木構造は構築出来ないため、辿った経路を記憶する方法を用いる。辿った経路を記憶するため Meta Data Segment にスタックを持たせる (表 5 における Context)。なお、このスタックは軽量継続

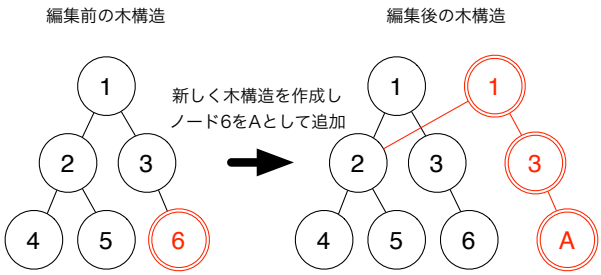


図 3　非破壊赤黒木の編集

ではなく関数呼び出しで利用している。赤黒木で利用する Data Segment を表 5 に表す。Data Segment は各ノードの情報を持つ `Node` と赤黒木を格納する `Tree` と、挿入などの操作中の一時的な木を持つ `Traverse` の共用体で表される。

```
// Data Segment for Red−Black Tree
union Data {
    struct Tree {
        struct Node* root;
    } tree;
    struct Traverse {
        struct Node* current;
        int result;
    } traverse;
    struct Node {
        int key;
        union Data* value;
        struct Node* left;
        struct Node* right;
        enum Color {
            Red,
            Black,
        } color;
    } node;
};

// Meta Data Segment
struct Context {
    stack_ptr node_stack;
    union Data **data;
};
```

表 5　赤黒木の Data Segment

赤黒木の赤が続かないという制約をチェックする Code Segment は表 6 のようになる。まず、親の情報が必須なために経路を記憶しているスタックから親の情報を取得する。親の色が黒である場合は木が平衡であるために処理を終了し、ユーザ側の Code Segment へと `goto` する。親の色が赤である場合は経路情報を再現するためにスタックへと親を挿入し、次の条件判定へと `goto` する。

ここで、赤黒木に対する処理を Code Segment とした場合、赤黒木のメモリ管理などは Meta Code Segment である。赤黒木を利用する Code Segment からは赤黒木の処理は意識する必要がないため、赤黒木の処理は Meta Code Segment のように見え、赤黒木のメモリ管理は Meta Meta Code Segment と考えられる。このようにメタ計算は階層構造を持つため、任意の Code Segment に対してメタ計算を適用することが可能である。

```
// Meta Code Segment
__code insertCase2(struct Context* context, struct
      Node* current) {
    struct Node* parent;
    stack_pop(context−>node_stack, &parent);
    if (parent−>color == Black) {
        stack_pop(context−>code_stack, &context−>
            next);
        goto meta(context, context−>next);
    }
    stack_push(context−>node_stack, &parent);
    goto meta(context, InsertCase3);
}

// Meta Meta Code Segment
__code insert2_stub(struct Context* context) {
    goto insertCase2(context, context−>data[
        Traverse]−>traverse.current);
}
```

表 6　赤黒木の赤が続かないという制約の判定

5. メタ計算を用いたデータ構造の性質の検証

CbC で記述された赤黒木と、それに対する処理の性質を実際に検証する。また、非破壊であるため木にデータを挿入、削除した際に過去の木は変更されない。非破壊赤黒木に求められる性質には、挿入したデータを参照できること、削除できること、値の更新ができること、操作を行なった後の木はバランスしていること、などが存在する。本論文では任意の順番で木に要素を挿入しても木がバランスしていることを検証する。

まず、検証を行なうために満たすべき仕様を定義する。仕様はデータ構造が常に満たすべき論理式として表現し、CbC のコードで記述する。検証する仕様として、とを CbC で定義する (表 7)。

```
if (akashaInfo.maxHeight >
    2*akashaInfo.minHeight) {
    goto meta(context, ShowTrace);
}
```

表 7　木の高さの仕様記述

表 7 で定義した仕様に対し、挿入する値と順番の全ての組み合わせに対して確認していく。また、仕様を満たさな

い反例が存在する場合はその具体的な組み合わせの値を出力する。

当研究室で開発している検証用メタ計算ライブラリ akasha と、C 言語の有限モデルチェッカ CBMC[3] を用いてこの仕様を検証した。- akasha では検証用の仕様と検証用の処理を Meta Code Segment に定義する。挿入順の数え上げには深さ優先探索を用いる。CbC には関数呼び出しが存在しないため、深さ優先探索を行なう際にスタックフレームに相当する Meta Data Segment を自ら用意する。各深さ毎の木を保存しておくことで、ある深さまでの探索が終了した際に木を再現ことができ、高速に探索を行なうことができる。要素数 13 個までは任意の順で挿入を行なっても仕様が満たされることを akasha を用いて検証した。また、赤黒木の処理内部にバグを追加した際には akasha は反例を返した。

同じ仕様を C 言語の有限モデルチェッカ CBMC で検証する。CbC は C とほぼ同様の構文を持つため、単純な置換で C 言語に変換することができる。CbC で記述された赤黒木のコードを置換で C 言語に変換し、CBMC で検証を行なった。CBMC における仕様は bool を返すコードとして記述するため、akasha と同じ仕様定義を用いることが可能である。また、ポインタへの不正アクセスといった一般的なバグへの検証コードが機能として存在する。挿入順の数え上げには CBMC の機能に存在する非決定的な入力を用いた。しかし、CBMC で検証できる範囲内では Gears の赤黒木の仕様を検証することはできなかった。有限モデルチェッカでは探索する状態空間を指定し、それを越える範囲は探索しない。CBMC で実行可能な最大の状態空間まで探索しても、バグを含んだ赤黒木に対する反例を得ることはできなかった。

よって、CBMC では検証できない範囲の検証を akasha で行なうことが確認できた。

6. まとめと今後の課題

CbC で記述したプログラムに対する仕様の検証を行なうことができた。CbC は Code Segment と Data Segment を用いてプログラミングするため、検証用にコードを変更することなくメタ計算の切り替えで検証を行なうことができた。

今後の課題として、検証できる範囲の拡大や効率化、値の抽象化などがある。CBMC ではポインタへの不正アクセスを行なう実行例を検出することができる。akasha でも一般的なエラーに対するメタ計算などを定義したい。本論文では入力の組み合わせを全探索するため、過去に探索した形状の木に対しても探索を行なっていた。木の形状を抽象化することで探索範囲を抑えて高速に検証ができると思われる。また、抽象度を上げることで有限回でなく無限回の操作も扱いたい。さらに、検証の際に証明を併用することで抽象化や状態数の削減が行なえると考えている。

参考文献

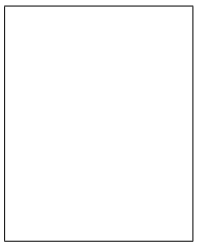
[1] 徳森海斗, 河野真治: Continuation based C の LLVM/clang 3.5 上の実装について, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2014)

[2] 伊波立樹, 東恩納琢偉, 河野真治: Code Gear、 Data Gear に基づく OS のプロトタイプ, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2016)

[3] Clarke, Edmund and Kroening, Daniel and Lerda, Flavio: A Tool for Checking ANSI-C Programs, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)

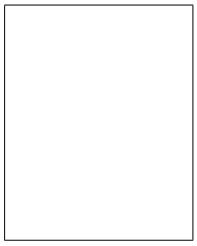
[4] The Agda Wiki - Agda 入手先 ⟨http://wiki.portal.chalmers.se/agda/pmwiki.php⟩ (参照 2016-07-05).

[5] Moggi, Eugenio: Notions of Computation and Monads, Inf. Comput (1991).



比嘉健太

2015 年琉球大学工学部情報工学科卒業。2015 年琉球大学大学院理工学研究科情報工学専攻入学。



河野真治

1960 年生．1982 年情報処理大学理学部情報科学科卒業．1984 年同大学大学院修士課程修了．1987 年同博士課程修了．理学博士．1987 年情報処理大学助手．1992 年架空大学助教授．1997 年同大教授．オンライン出版の研究に従事．2010 年情報処理記念賞受賞．電子情報通信学会，IEEE，IEEE-CS，ACM 各会員．